Chapter 1
Introduction to Scientific Computing Technologies for Global Analysis of Multidimensional Nonlinear Dynamical Systems

Nemanja Andonovski, Franco Moglie and Stefano Lenci

Abstract To determine global behaviour of a dynamical system, one must find invariant sets (attractors) and their respective basins of attraction. Since this cannot be made extensively with analytical methods, the numerical global analysis is currently the subject of intensive research, especially for strongly nonlinear, multidimensional dynamical systems. Numerical analysis in dimensions higher than four present a challenge, since it requires significant computing resources. Numerical methods used in global analysis that can benefit from high-power computing are those that can parallelize either data or task elaboration on a large scale. Mass parallelization comes with large number of difficulties, restrictions and programming hazards. When not implemented in compliance with hardware organization, data and instruction management can lead to severe loss of parallel algorithm performance. Systematic and methodical approach to design parallel programs is, therefore, critical to get the most from expensive high-power computing systems and to avoid unrealistic speed-up expectation. Considering these difficulties, the goal of this chapter is to introduce readers to the world of high-power computing systems for science and global analysis of strongly nonlinear, multidimensional dynamical systems. Topic covered are classification and performance of hardware and software, classes of computing problems and methodical design of programs. Two major hardware platforms used for scientific computing, clusters and systems with computational GPU are considered. Functionality of widely utilized software solutions (OpenMP, MPI, CUDA and OpenCL) for high-power computing systems is described. Performance of individual computer components are addressed so that the reader can understand advantages, disadvantages, efficiency and limits of each hardware platform. With

N. Andonovski (✉) · F. Moglie · S. Lenci
Polytechnic University of Marche, Ancona, Italy
e-mail: n.andonovski@pm.univpm.it

F. Moglie
e-mail: f.moglie@univpm.it

S. Lenci
e-mail: lenci@univpm.it

© Springer Nature Switzerland AG 2019
J. C. Jauregui (ed.), *Nonlinear Structural Dynamics and Damping*,
Mechanisms and Machine Science 69,
https://doi.org/10.1007/978-3-030-13317-7_1

this knowledge users can judge if their computation problem is suitable for mass parallelization. If this is the case, which hardware and software platforms to use. To avoid many traps of parallel programming, one of the methodical design approaches is covered. Topic is closed with example applications in science and global analysis.

1.1 Introduction

Well explored analytical methods for linear phenomena is not viable to fully determine dynamics of systems that model real life application. Lack of analytical methods have led to development of various numerical methods for global analysis of nonlinear dynamical systems. Numerical methods offer possibility to solve problems that do not have analytical solution in closed form, sacrificing generality of solution—any change in system parameters require new computations. To determine global behaviour of nonlinear systems, the amount of computations increases dramatically and require large computing resources, especially in higher dimensions [1, 2].

Historical roots of numerical computations are found in rudimentary mechanical calculating devices that over long number of years have evolved to contemporary electrical supercomputers. Major point of calculating machines remains the same—to aid people in solving various, mostly mathematical, problems. Precursors of electrical computers are mechanical tools like simple adding devices, abacus, mechanisms for drawing integrals of graphical functions or mechanical machines that integrate differential equations [3].

Transition from various mechanical computers to electrical ones were made with implementation of concept machine capable to compute anything that is computable (Turing machine). First electrical, vacuum tube computer, EINIAC (Electronic Numerical Integrator and Calculator), marks beginning of new era that will lead to computational capacity that surpasses anything ever imagined. Technology evolved to such scale that functionality of mechanical calculating machines are reproduced with integrated circuits, approx. 10 nm in size each [3–5].

High complexity of massively parallel computing systems used in science and engineering often cause difficulties during program design, particularly for those not educated in information technologies. In this chapter we aim to introduce readers to the capabilities of modern day computer systems and systematically explain all concepts needed to start making efficient program for large-scale numerical computing of global behaviour for high-dimensional nonlinear dynamical systems. Before tackling problems of high-power computing, it is necessary to analyse performance of basic computer components from which supercomputers are made of. Organization of components are described through architectures that act as logical concepts capable to deal with various computational problems. Families of problems that benefit from mass parallelization are explained along with architectures and computer implementations able to efficiently compute related tasks. Systematic design method is crucial to avoid dangers of parallel execution that do not exist in sequential approach. First step is to decide which hardware platform is most efficient for computing of the problem to be addressed. Following steps of design methodology focus on how to make program that exploits optimally or almost optimally the computer resources. Brief introduction is offered also for most common software platforms used in scientific applications.

Topic is closed with examples massively parallel computations in global analysis and science. For in-depth understanding and functionality of software and hardware platforms or global analysis methods, readers are referred to abundant resources provided by scientific literature, manufacturers and user community.

For clarity and brevity, number of definitions and formulas are omitted and principles are explained as concisely as possible. Readers that find certain topic useful for their analysis should refer quoted literature for more details and further hints on practical implementations.

1.2 Analysis of Dynamical Systems

Mathematical description of dynamics is not limited to a mechanical systems. Biological, economics, psychological and many other non-mechanical systems evolve dynamically, and their evolution is governed by systems of various equations [1]. Continuous-time systems are represented through ordinary differential equations and discrete with difference equations. Other representations such as cellular automata, lattice maps and other are also often used, especially partial differential equations where system evolution is dependent from both spatial organization and time [6]. The mathematical notion of dynamical systems express fact that the motions are determined by some rules or laws. Thus, this deterministic approach allows to form space of states (phase space) and to acquire system state at any time given the initial (and boundary when required) condition [6].

1.2.1 Linear Analysis

Traditionally, the analysis starts from linear approximation of nonlinear systems. Having a general solution in closed form gives a formal way to explore linear systems [6]. Family of linear systems gives qualitatively same response for all values of system parameters, making parameter analysis straightforward. With analytical methods it is fairly easy (from computational point of view) to determine stable and unstable behaviour of the solution. For these systems the attractor, if present, is unique, and thus the long term solution is “easy”. No multi-stability occurs, and thus basins of attraction are meaningless.

1.2.2 Nonlinear Analysis

Majority of natural systems are in fact nonlinear [6], but an initial clue of overall dynamics can be obtained by analyzing corresponding linear system. Similarities between linear and nonlinear system depend on the magnitude of nonlinearities, where higher nonlinearity produce more diverse collection of behaviours, such as multi-stability, quasi-periodicity, deterministic chaos, solitons, fractals, riddled basins or pattern formation. In order to determine which of those diverse behaviors are present in the system, analysis combines analytical approximation, numerical calculations and experimental data. An important part of analysis is observation of system behavior during the change of some system parameters, since in many cases it can lead to change in topology of the system (qualitative change), especially when the system is nonlinear.

Two classes of nonlinear systems that are in most cases subjected to analysis are those that can be represented through systems of partial differential equations or with the systems of ordinary differential equations. For global analysis most interesting are dynamical systems which can be reduced to a system of ordinary differential equations of first order [7]. Most important family are systems of second order differential equations, modelled from Second Newton's Law of motion, that can easily be reduced to the first order systems. Dimension of resulting system (not to be confused with degrees of freedom) is equal to the number of first order differential equations. Each dimension in this case corresponds either to the coordinate or velocity that appears in the system. This representation gives possibility to use well developed numerical techniques [8]. Solutions then can be analysed as trajectories in multi-dimensional state space.

1.2.3 Global Analysis

Nonlinear systems may have arbitrary number of steady motions, some stable some not. If trajectories converge towards certain steady state, it is called attractor and repellor if trajectories are diverging away from it. Basin of an attractor consists of the all initial conditions that converge to associated attractor in forward time. Goal of global analysis is to get a global behaviour of system, expressed in terms of attractors and their respective basins. It is usually conducted together with time series, frequency response and parameter variation (bifurcations) [1].

Resulting behaviour can be very colorful. Beside geometrically regular shapes as points or limit cycles and torus, attractors that may occur in nonlinear dynamics can be of a fractal structure (strange attractors) [6]. Fractal curves [9] are not smooth, but geometrically irregular or an uneven shape of non-integer dimension, repeated over all magnifications (from large to infinitesimally small). Attractors with fractal structure are usually associated with chaotic motion, but also strange nonchaotic attractors exist. In cases where multiple attractors coexist basins can be separated by smooth or fractal curves. Numerically, fractal boundaries can only be assumed up to the computer precision. Another possibility is that basins may be riddled [9]. It means that border between basins is not a curve (neither smooth nor fractal) and points in infinitesimally small hyper-sphere around certain initial condition do not necessarily converge to same attractor.

Although behaviour may be complex, the numerical procedures used at global analysis are able to compute fairly accurate results [10]. Difficulty comes with increase in system dimension, as number of required computations increases exponentially. Therefore, to numerically analyse dynamical systems with large dimension it is necessary to resort on powerful computational computer systems, which heavily really on mass parallelization of computation. Currently, multidimensional global analysis is focused at building basins in more than four dimensions. Six-dimensional systems are being examined contemporary while eight-dimensional present a challenge for both computation and visualization.

1.2.4 Numerical Computing Integration

Numerical integration schemes [8], used to overcome the limits of analytical methods are broadly classified as either explicit or implicit. In explicit schemes the governing equation is written at time for which all the solution variables are already known, and the difference equation is then solved for the solution at the next time step. Explicit methods are generally preferred for solution of problems where the interesting part of the solution is changing rapidly in time like in wave propagation problems or crash analyses. In implicit schemes, the governing equation is written at time $t + \Delta t$, while the solution is known for time t. Implicit methods are better suited for problems where the solution variation over time is less rapid, and relatively larger time steps can adequately resolve the problem.

Commonly used numerical methods (i.e. adaptive step Runge-Kutta of 4th order) can be already found within libraries and freely used for scientific computing without need to implement the whole numerical procedures in programming language.

1.3 Computer Architectures

Functionality of a computer or sub-system, without concern of actual implementation, is defined by the *architecture* [5, 11, 12]. Architectures are classified according certain properties of hardware (parallelism type, memory organization, etc.) and every computer system is synthesis of various architectures.

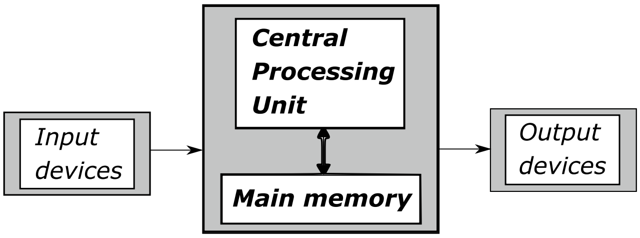

Fig. 1.1 Von Neumann concept of computer

1.3.1 Von Neumann Architecture

Basic logical concept of computers is presented through Von Neumann model [5, 11, 12] on Fig. 1.1. It consists of a processing unit (*processor*), memory and data pathways (buses). Processor interprets and executes programs which are combination of instructions and data. Processor fetches instructions and data and execute them sequentially one after another. Every instruction refers to memory address where next instruction and data are stored, meaning that flow of program is driven by instructions.

Complexity of real computer system is significantly higher than in Von Neumann model, as consequence of adding various complements to compensate restrictions of low-performing components. Concepts of high-power computing are based on combining multiple logical units in various ways [12, 13] resulting in numerous computer systems dealing with problems in science, engineering, economy, etc.

1.3.2 Parallel Architectures Parallelization Classification

As current manufacturing technology reached peak where fabrication process is getting increasingly difficult to improve, performance increase is achieved by parallelizing the computations. Performance is enhanced by parallelizing processing elements (i.e. multiprocessors, neural networks) or instructions inside processing element (pipeline, vector processing) [5, 11–13].

Pipelining and vector processing are already implemented inside modern processors and are not widely used as stand-alone parallelization concepts. For scientific computing most relevant are multiprocessor architectures, as being most performant parallelization concept. Other existing parallel architectures are rarely used because they are efficient only for problem-specific applications. It is notable to mention data flow computing, systolic processing and neural networks as concepts utilized in science, that are not instruction driven as majority of architectures based on von Neumann model.

1.3.2.1 Pipeline, Superscalar and Very Long Instruction Word (VLIW) Processors

Processor have multiple stages of executing one instruction. Main segments are instruction fetching, decoding, execution and result store. All segments can be further fragmented, resulting that processor have to do multiple steps in order to fully execute one instruction. Pipeline parallelization is achieved by simultaneously executing different segments of successive instructions. Pipeline technique is also exploited in architectures where instructions are pipelined with multiple processors.

VLIW are instructions combined from several shorter, that allow processors to have a deeper pipeline. Superscalar processors exploit pipelining concept by adding more hardware circuits, so processor can do arithmetic, logic or floating-point operations of several separate instructions parallelly.

1.3.2.2 Vector Processors

As pipelined computers execute multiple instructions simultaneously, vector computers can process entire vector of data in one instruction [5]. Fetching of data is done for whole vector not for just one piece of data. Multiple arithmetic circuits then can manipulate entire vector during one instruction period instead of processing each vector element as successive instruction. Each vector element is manipulated with same instruction.

1.3.2.3 Systolic Architecture

In systolic array [5], processing units are organized in a network, so that at each cycle part of data is calculated and forwarded to the subsequent processing elements in grid. After initial latency (number of cycles until all computing elements receive first block of data) system can efficiently compute repetitive task (matrix transformation, sorting, Fourier transform, etc.). Implementations of this architecture is effective for specialized computations, making it efficient, compact and economically convenient but inflexible.

1.3.2.4 Data Flow Architecture

In dataflow computing [5], the order of execution is managed by data availability, not by order of instructions. An instruction executes when required data arrives, as result of program flow being driven by data dependencies. Each instruction reference to next instruction, contrary to instruction ruled architectures where instructions reference to memory location of next instruction. This architecture found successful implementation only in several areas (telemetry, digital signal processing, etc.).

1.3.2.5 Neural Networks

An attempt to mimic structure and function of biological neural networks [5] is implemented by parallelly interconnecting large number of simple processing elements. Artificial neural networks can derive solutions in dynamic situations from incomplete or probabilistic data. To be able to adapt, the learning algorithm of neural network requires training multiple running of program with correct input data and known solutions. Difficulties come from possibility that network is adapting its behaviour from incorrect previous information. Great complexity of artificial neural networks limits its usage at scientific computing, although the promising potential uses.

1.3.2.6 Multiprocessors

Strict definition of multiprocessors [5, 12, 13] is not well established due to variety of implementations. General idea is to connect multiple computing entities into network that act as single system, which can run one or more programs that are divided into numerous parallel tasks. Concepts and implementations of multiprocessor computers often match the requirements necessary to compute most problems encountered in scientific and engineering applications. Remainder of this chapter is therefore, dedicated to multiprocessor architectures, implementations and design methodology.

1.3.3 Stream Concurrency Architecture Classification

Flynn's taxonomy [5, 11–13] classifies computer architectures according to number of parallel streams. Instruction stream (program) is sequence of commands/ instructions executed by processors. The flow of data that is being manipulated by commands from instruction stream is called data stream. Flow of instructions goes from memory to processing unit, while data flow is bi-directional. Flynn's taxonomy is most often used for classifying multiprocessors, since logical concepts of taxonomy reflect multiprocessor organization.

1.3.3.1 Single Instruction Stream, Single Data Stream (SISD)

Conventional serial computers with one processor, built according to the von Neumann model, work on SISD principle. Processor is able to sequentially process one stream of instructions and operate over single data set. Flow scheme of SISD architecture is depicted in Fig. 1.2.

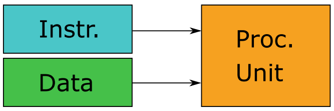

Fig. 1.2 SISD execution model

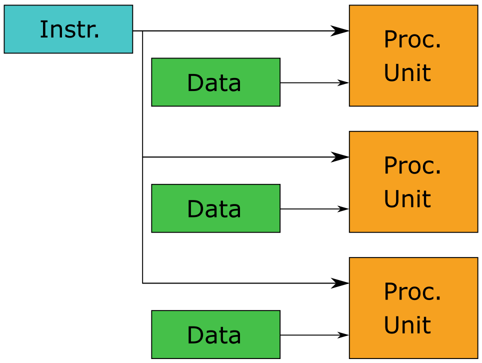

Fig. 1.3 SIMD execution model

1.3.3.2 Single Instruction Stream, Multiple Data Streams (SIMD)

Concept of multiprocessor computer that parallelly execute single instruction stream on multiple data streams. This means that each processing unit will execute same sequence of instructions, but with distinct data stream, as on Fig. 1.3. Execution is lock-stepped, which means that processing units are synchronized and all tasks will start and finish in same time. Examples are graphical coprocessors and array computers where multiple processing units work under control of single control unit.

1.3.3.3 Multiple Instruction Streams, Single Data Stream (MISD)

In MISD architecture each processing unit handles own (distinctive) instruction stream, operating over single data stream. This architecture, shown in Fig. 1.4, is

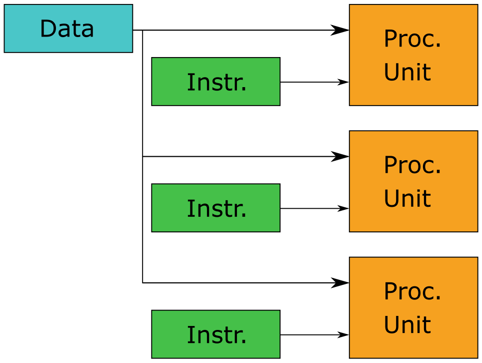

Fig. 1.4 MISD execution model

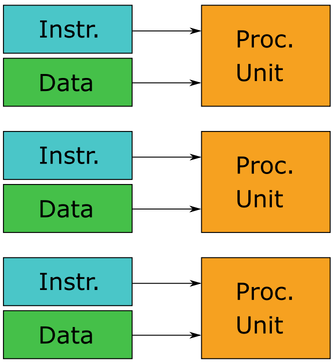

Fig. 1.5 MIMD execution model

rarely used, mostly for fault tolerance, as system must agree on the result from all instruction streams. Space shuttle flight control computer is an example.

1.3.3.4 Multiple Instruction Streams, Multiple Data Streams (MIMD)

Computer system where multiple autonomous processing units simultaneously execute distinct instruction streams over separate data streams is shown on Fig. 1.5. Tasks in MIMD architecture are asynchronous and execution can start and finish at any time independently on other processing unit tasks. Almost all modern systems are built according to MIMD architecture, from computers with one multi-core processor to clusters and cloud super-computers.

1.3.4 Memory Access Level Classification

To achieve massive performance increase, multiple processors are connected into single system (networking). Coupling between processors in network might be implemented on various ways, resulting in computer systems spanned from single case to geographically dispersed systems [11, 14, 15]. Programs that are run on high-power computing systems are divided into numerous parallel *tasks*. Common name for parallelly executing tasks in shared memory systems are *threads* and on distributed memory systems—*processes*.

1.3.4.1 Shared Memory Architecture

Memory hierarchy of shared memory systems [11, 14, 15] allows all processors to access central system memory. Memory that is shared between tightly coupled processors is called *local memory*. High performance of shared memory systems comes from availability to access data directly. If synchronization of instructions and data management are not handled carefully, it can lead to various errors.

1.3.4.2 Distributed Memory Architecture

In distributed memory systems [11, 14, 15] every processor has its own local memory. Systems architecturally organized like these are loosely coupled and processors can operate independently. Local memory of one processor is inaccessible by remote processors and data is exchanged through communication channels. Part of system consisting of a processor and its local memory is named *node*. Theoretically there is no limit in how many nodes can be connected to form distributed memory high power computing system. Data coherency is automatically maintained since remote memory is not directly accessible in any case. Drawback is that node interconnections transfer data at much lower rates than bus connections on shared memory systems.

1.3.4.3 Hybrid Memory Access Architectures

Actual implementations are not restricted to strictly shared or distributed architecture. Reality rather correspond to MIMD architectures [13, 15]. In fact, many of the high-power systems are combination of multiple coupled sub-systems. Hybrid architectures have interconnected nodes where each node may be architecturally different from others. For example, node might consist of several processors or can be GPU accelerated.

1.3.5 Other Architecture Classifications

To avoid over encumbering reader with unnecessary information for scientific computing other classifications of parallel designs are not considered. For curiosity reader may refer to literature [11–14] for classifications based on instruction set architecture, network organization, degree of parallelization, pipelining level, etc.

1.4 Hardware Components

Basic computer components are building blocks for personal computers and supercomputers as well. In this section readers are introduced to functionality and performance of relevant components, required to efficiently design parallel software for scientific purposes. High level of attention must be paid to the performance of all components to avoid *bottlenecks*—limited capacity of computer caused by single low-performing component [11, 16].

Actual physical parts of computer are various combinations of electrical circuits [11, 12, 16]. Electrical elements (transistors, capacitors, resistors, diodes, inductors, etc.) are integrated in chipsets that perform designated operations. Chipsets can be digital that operate only on binary data (0 and 1 values) or analogue which can manipulate variable continuous signals.

Personal computers are relatively small sets, while high-power computing systems are created from considerably larger number of components, combined to operate as coordinated system. To grasp how complex systems behave it is necessary to understand functionality and performance of each of the individual components. To interact with user, computers need input/output devices (keyboard, mouse, monitor, printer, etc.). Modern computers also cannot be imagined without a Graphical Processing Unit (GPU) that enhance visual output of computers. Other numerous components and devices such as TV cards, sound reproduction systems or gaming controllers do not play role in scientific applications and although are regular parts of modern computers, are not considered.

1.4.1 Central Processing Unit (CPU)

The set of electronic components that operate and manage data is integrated on one chip called *central processing unit* or *processor* [11, 12, 16]. It fetches instructions from main memory, decodes them and perform indicated operations over correct data set. *Datapath* is part of processor that execute instructions and manipulate data. It is a network of arithmetic and logic units connected to internal memory. Logical circuits can perform i.e. bitwise logical operations, while arithmetic circuitry can add, subtract, increment, etc. Modern processors have also floating-point units, the circuitry specialized to perform arithmetic operations over floating-point numbers faster than arithmetical can. The control unit is part of processor that manages scheduling of instructions, data transfer from/to memory and coordinates other components. Multi-core processor is one integrated chip consisting of two or more independent processors named *cores* in this case.

System clock [5, 11, 12] is part of processor that regulate update time of internal components. At every *cycle* (tick of clock) states of components are changed. Cycle length must be long enough to allow the propagation of state change through all processor components. To execute one instruction, processor in most cases requires multiple cycles.

With combination of large number of integrated circuits processor can perform diverse range of operations at extremely high rate. Current high-end technology enables billions of transistors to be integrated on a single chip. For example, Qualcomm Centriq 2400® processor (one processor with 8 cores) have about 18 000 000 000 integrated circuits on chip with 398 mm^2 area [4].

1.4.2 Main Memory

Instructions and data of currently running programs are stored in *main memory* [11, 12, 16]. Every time processor is ready to execute some part of the program, it has to make an access to main memory to retrieve required information. Fetching process is slow in comparison to execution time of processor, creating frequent bottlenecks. *Memory Wall* is the name for performance restriction imposed by low performance of main memory.

Main memory can be either read-only (ROM) or read and write (random access memory—RAM). Data residing on ROM is permanently stored and mostly used to keep hardware management programs hidden from users. On the other side, RAM is volatile type of memory where all user applications are loaded on runtime. When speaking about memory in following sections, it will be referred to main memory (RAM). When mentioned, other memories will be addressed by their type—registers, cache or storage.

1.4.3 Internal Memory

Major mechanism to overcome restrictions of main memory is to use small amounts of high-speed memory integrated on processor chip. Two types of internal memories, that considerably reduce data access time, are *registers* and *cache* [5, 11, 12].

Registers, that are located directly near processing circuits, are very low latency memory with access time from one to few clock cycles. Other memory types that are located far away, require additional access control and pathways that considerably increase fetching time. Despite the high performance, processors are fabricated with low amount of register memory due to high manufacturing costs (per storage capacity) and large spatial occupancy on processor chip.

Cache memory serves as a buffer between registers and main memory. When some piece of data is processed, it is highly probable that nearby data will also be needed in near future. Therefore, processor transfers entire block from main memory to cache, instead of only data needed at the exact moment of access. Cache is organized into levels, where lower levels, closer to processor cores are faster but with lesser capacity. Levels 1 and 2 are commonly dedicated to each core and are not shared, while higher levels of cache are accessible to all processor cores. Highest level of cache sometimes can be allocated as part of main memory, while lower levels are always integrated on processor chip.

1.4.4 Mass Storage Memory

Storage (or secondary) memory [5, 12] is a component or external device where data and programs are permanently kept when computer is not powered. Examples are Hard Disk Drive, DVD disks, Flash memory or online cloud storages. It has high capacity and low manufacturing cost per storage unit. Access time is considerably slower compared to execution time of processor operations. Storage memory does not actively participate in computations, but for task with large data sets, writing and reading of information can create performance drops due to low speed of mass storage devices.

1.4.5 Data Transfer

Internal transfer of information between processor and other components is done by *bus* [11, 12, 16], a combination of wires acting as data highway. It can be a point-to-point pathway, connecting two specific components or shared (multi-point) connection between several components. Speed of the bus is affected by its length as well as by the number of devices sharing it. Bus clock manages the time interval of its state update. Bus cycle is longer than processors which makes it inefficient to supply information directly from main memory.

Computer systems created by joining multiple computing entities can exchange information over Local Area Network (LAN) or Wide Area Network (WAN) [16]. Various protocols manage how data is sent and received in network. Internet and Ethernet are protocols with low and medium data transfer performances used in loosely coupled systems. Tightly coupled systems use high-performance network protocols (i.e. InfiniBand$^{®}$) to communicate information on higher rates.

Actual links that represent communication channels can be wired (collection of digital or analogue wires, coaxial cables, Ethernet cables or optical wires) or wireless (radio signals or optical communications). Connections other than bus have considerably lower transfer capacity, causing low utilization of resources on systems that have to communicate large quantities of information.

1.4.6 Graphical Processing Unit

Graphical image processing is data intensive operation and accelerator graphic coprocessors are intensively developed to aid processor to visualize graphic content [13, 17]. When processor encounters graphically intensive part of program it forwards data and instructions to Graphical Processing Unit (*GPU*, coprocessor or accelerator for short) that have electronic circuits optimized for efficient image processing. Price for being specialized for certain tasks is that GPU is highly inflexible and unable to function on its own.

Special branch is computational GPUs (or General-Purpose GPUs) that utilize high number of simple cores (stream processors, shaders) which are able to conduct general computations, while retaining high performance in data intensive applications [13, 18]. Graphical coprocessors are fabricated on separate (graphic) card connected to processor by bus. High-performance RAM is located on the coprocessor card due to high demand for data.

In this chapter, when mentioning GPU, we will refer to the general purpose graphical coprocessors only, since computations are major concern, not the image processing acceleration.

1.4.7 ASIC and FPGA

Application-Specific Integrated Circuit (ASIC) is an integrated chip intended for specific use, while field-programmable gate arrays (FPGAs) are designed to be configured by user after manufacturing [11, 17]. Either design offers excellent efficiency when is developed for specific use. To formulate functionality of both designs elevated expertise in information technologies is required, thus not covered in this chapter.

1.4.8 Performance of Computer Components

Principal matter for high-power computing systems is how well extensive tasks can be accomplished. Performances of individual components reflect the overall ability of computer system to complete required tasks. Knowing the performance of data operations and transfer helps in understanding why various components exists and why are implemented in certain ways. From efficiency perspective, performance issues of particular components must be considered during design of parallel algorithms.

Depending from context, performance can be measured as amount of jobs done per unit of time (throughput) or amount of time required for some task to complete (execution time, latency, response time, access time or delay) [11, 17].

1.4.8.1 CPU Performance

When comparing performance of processors, usually their speed is a relevant factor. Unfortunately, term speed of processor does not have a determinate meaning. It may refer to several measures that quantify some of the processor characteristics. Reason for this is rich complexity of micro-architectures that differ even between processors of same manufacturer and family. Current technologies of manufacturing processors with multiple cores, additionally complicates comparison of different processors.

Number of instructions per second (*IPS*) [11, 17] is one of the measures used, but for processors with complex instruction set it is not an accurate measurement. Instructions have unequal size, resulting in variable execution time for each instruction. More common and precise speed measure is number of system state updates per second, namely *clock speed* or *frequency* (GHz) [11, 17]. Clock speed also does not give definite comparison quantities, since some processors may finish certain actions in fewer clock cycles than others. Thus, even equal clock speeds of different processors do not guaranty that they will do equal amount of work.

Benchmark programs are used to get somewhat accurate comparison of performances for various systems or components. They may test performance of overall systems or a single component in various workload situations. In this way processors can be compared on how well they perform at certain task. For scientific computing where data is mainly floating-point numbers, measure how well system (or single component) perform is represented through floating-point operations per second (*FLOPS*) [11, 14, 17].

1.4.8.2 GPU Performance

Graphical coprocessors are highly specialized hardware and their high performance limits the flexibility. Metrics for GPU performance are same as for processors, but direct comparison of processors versus general-purpose GPUs is vague because each is designed to be efficient at different type of tasks [17, 18]. However, certain applications are not focused on computation, such as the matrix transpose. In those cases, metric relevant to indicate throughput of GPU tasks is MBPS (megabytes per second).

GPU use extensive amounts of electrical energy for computations, that often causes necessity to measure also the power consumption [19]. *Energy* is a measure of how much electrical energy the system consumes in total, *power* is energy consumption per time unit and *power efficiency* quantifies arbitrary performance measure per power consumption.

1.4.8.3 Data Transfer Performance

Performance of data transfer is governed by two factors, *latency* and *channel width* [11, 16]. Latency is time in seconds that takes to access the data. Width is the amount of information that can be transferred at once. Combined, is a measure called *bandwidth* that quantifies transfer capacity by the amount of data that can be transferred per unit of time. Drops of overall computer performance is often caused by data transfer bottlenecks, where data is not supplied fast enough or in low quantities required for high percent of utilization of computing resources.

Table 1.1 Memory hierarchy

Mem. type	Approx. cap.	Latency
Registers	200 ps	4 kB
Cache L1	64 kB	1 ns
Cache L2	256 kB	3–10 ns
Cache L3	16–64 MB	10–20 ns
RAM	up to 256 GB	50–100 ns
HDD	up to 64 TB	5–10 ms
Flash	up to 16 TB	100–200 ms

1.4.8.4 Memory Hierarchy and Performance

Memory can be hierarchically classified based on capacity and response time [5, 12]. Table 1.1 shows performance and capacities of various memories that can be effectively manufactured with current technologies [20]. At top levels of hierarchy are internal memories with fast access speed and low capacity. Main memory is in middle, with average access time and significantly higher capacity than internal memory. Low levels of hierarchy are occupied by massive storage devices with slow response time.

1.4.8.5 Overall System Performance

Overall system performance depends on combination of components and type of task being computed. Systems with fast and numerous processors cannot work efficiently if memory and data interconnections cannot keep up with performance of processors. Drops of performance is even more evident with GPUs where efficiency is highly influenced on both hardware factors and how tasks are scheduled. Optimization of hardware according to one component is bad practice and each platform should be designed to efficiently utilize all resources. Also, to maximize performance of hardware platform it is equally important to use an appropriate programming model [11, 13–15].

Supercomputers that exist today are often combination of multiple processors and GPUs. List of current (scientific) supercomputers compared to their floating-point operation power can be found at The Top500 web-site [21]. Lately, for ecological and economic purposes another measure is often used in scientific computing—FLOPS/Watt, which measures the amount of work done per energy consumption. Supercomputers sorted by their energy efficiency are found at The Green500 List [22].

1.5 Massively Parallel Designs

Systems with large number of processing units are named massively parallel designs for high-power computing [17]. Those systems use numerous processors, connected computers or GPUs to achieve coordinated execution of program in parallel [11–13]. Depending the interconnection implementation and centralization level, computers might be developed as mainframes or connected in clusters, grids or clouds. Other types of computer designs are available, such as fog or peer-to-peer computing. Some of the nodes in network might have computational GPU, which does not change type of computer system, only the performance of the node.

Topic of massively parallel computers is too large to be covered in one chapter, also unnecessary because two designs mostly used in scientific and engineering computing are clusters and computers with computational GPU. Functionality of cluster and GPU accelerated computers (nodes) will, therefore, be explained in detail while other implementations are only listed according to [11, 13, 17].

1.5.1 Classification of High-Power Computing Platforms

Workstation is historically broad term, firstly used for hubs where the operating person would interact with large sized computers. Nowadays it is referred to personal computers or computers from which bigger systems are managed. Importance of PC workstation is that whole design process is done on it and finished programs are executed on more expensive platforms. But, if supplied with computational GPU, regular workstation becomes a high-power computing platform.

1.5.1.1 Mainframe

Mainframe computers contain multiple processors connected at the bus level. Mostly used in business applications for transaction processing. Since the metrics used to measure performance of mainframes is set of certain tasks (update of database, disk I/O, etc.), it is not useful for science and engineering applications where large number of floating point operations are required.

1.5.1.2 Clusters

Clusters are formed by connecting nodes into network that act as a single unit, computing single task (program/application). Cluster nodes are often closely centralized and connected with local high-speed interconnections.

1.5.1.3 Grid Computers

Resemble clusters, usually larger, more geographically dispersed and are not dedicated to run only one task.

1.5.1.4 Clouds

Cloud computers are systems whose organization is hidden from end user. Primarily developed to provide application services to commercial purposes without need from user to know organization of nodes or to have expertise in information technologies. Recently cloud services are as well available to scientific computing, usually as virtual machines representing clusters.

1.5.1.5 GPU Computing

Contrary to mass parallelization done by adding more processors or nodes to the system, any computer can become high-power computing platform by adding a general-purpose graphic card. When installed, GPU cards significantly increase capabilities in data intensive tasks of any computer (or node).

1.5.2 Clusters and General-Purpose Graphic Cards

Programming models mostly used in science are based on SIMD and MIMD architectures [13, 14, 17] for HPC computing. To have efficient computations, the programming models must be efficiently mapped to hardware. SIMD type problems efficiently exploit the inherit (hardware level) parallelism of GPUs, while clusters reflect MIMD architectures.

1.5.2.1 Clusters

Clusters computers are networks of computing entities (nodes) [13, 17, 18]. Nodes are commonly stand-alone computers, typically connected in LAN, with one of the high-speed protocols. Property of clusters, that distinguish them from other networked computers is that clusters operate as single system performing single task (program).

Considering that nodes are separate computers, memory of each node is not shared, making clusters a distributed memory system. Nodes communicate data by passing messages to each other. Transferring time of messages between nodes that are not directly connected can be considerably longer than between directly connected nodes.

To avoid communication overheads, it is advisable to know network topology of the cluster (line, ring, tree, mesh, hypercube, fully connected, star or bus).

Clusters are often implemented on master/slave dogma. Master process manages existence and work of numerous slave processes that carry out actual computations. MIMD problems that are instruction intensive and do not require large amount of data communication are highly suitable for cluster computations.

1.5.2.2 GPU Computing

Development of processors through history was focused on increase in number of successive instructions it can perform per unit of time. Graphical processing as SIMD operation had low benefit from increased clock speed. It required large number of equal operations to be carried out simultaneously. Graphical coprocessors evolved from being capable only to perform graphical computations to devices that can achieve remarkable speed-up in data intensive applications [13, 17, 18].

A GPU consists of several stream processors, each one having dozens of simple computing cores (CUDA cores). For example, GPU card based on NVIDIA Fermi® architecture can have up to 16 stream processors with 32 cores each [23].

1.5.2.3 GPU Cluster Combination

To expand capabilities of clusters, GPUs are added to some of the nodes. This mixed implementation is not surprising and it is good practice since it combines computational power of both MIMD and SIMD environments [24].

1.5.3 Classification of High-Power Computing Problems

Parallelization aim to speed-up the serial execution by either dividing instructions or data between concurrent processing units. To efficiently carry out parallelization process, it is necessary to determine what type of parallelization can be achieved.

1.5.3.1 Task Parallelism

Task parallelism [14, 17] occurs in cases where sequence of instructions can be divided into multiple, parallel and independent tasks. Task may run same or different code over same or different data, communicate information during execution and start and stop at arbitrary time unless explicitly specified otherwise.

1.5.3.2 Data Parallelism

Data intensive computations are those where large number of data elements have to be processed in the same way [14, 17]. One sequence of instruction is executed over multiple parallel data elements as in SIMD architecture. In comparison with task parallelism where instruction set is split over multiple tasks, in data parallelism data set is distributed between parallel processing elements.

1.5.3.3 Combining of Data and Task Parallelism

At various degrees, majority of actual programs and computation problems are not strictly data or task parallel. Problems described with MIMD architecture often combine parallelization on both data and instruction levels [25]. Certain problems, such as Fast Fourier transform or some sorting procedures, also benefit from possibility that algorithm can switch between data and task parallel execution models during computations.

1.5.4 Classification of High-Power Computing Paradigms

1.5.4.1 High-Performance Computing (HPC)

Tasks that require large amount of computational power during short time periods (one day or less) are characterized as *high-performance computing* [26]. Measure of computational power in HPC jobs is FLOPS. HPC systems tend to focus on tightly coupled parallel tasks, and as such they must execute within a particular site with low-latency interconnects, mainly clusters or mainframes. Presuming that majority of scientific computing is done over floating-point data, HPC is in most cases appropriate computing paradigm for scientific applications.

1.5.4.2 High-Throughput Computing (HTC)

When task at hand is far larger that require months or years to compute it is not important how fast, but how many jobs can be finished per unit of time (in HTC terms—jobs per month or year) [26]. HTC systems deal with sequential jobs that can be individually scheduled (loosely-coupled tasks) on many different computing resources, such as grids or clouds.

1.5.4.3 Many-Task Computing (MTC)

Many applications are not computing extremes as HPC and HTC [26]. Paradigm that offer middle ground is MTC, where both tightly coupled and independent task can be executed, no matter if tasks are instruction or data intensive, large or small.

1.6 Performance of Parallel Designs

For high performance of computing systems, it is not enough to have parallelization on massive scale [11, 13–15, 18, 20, 27–29]. Utilization of resources depends on both hardware and software factors and each computing problem should be computed on appropriate platform. Algorithm type, programming issues and hardware or communication restrictions downgrade performance when are not addressed properly.

As many factors are involved in science and engineering, it is substantial to decide what performance metric is relevant. Performance models serve to compare how efficiently different algorithms perform specific requirements. Analysis of performance models can discover many inefficiency causes. Often when not planned in advance, program hits performance wall, in which case code re-factoring must be done. To achieve optimal performance of parallel designs it is necessary to balance software and hardware factors and in many cases, programming effort and costs also must be considered. To evaluate performance of parallel programs, number of models are proposed [15, 20, 28] to address different performance issues.

Performance measures and issues of parallel designs are addressed in advance, so that during design methodology section reader can comprehend importance of each step.

1.6.1 Scalability

Property of computer system or computing problem that describe how well it cope with resource improvement is called *scalability* [11, 14, 15, 17, 28]. *Overheads* are the segments of program code that do not benefit from improved resources since must be executed serially on one processor. Problem or computer that can manage well the increase of resources is called *scalable*. Effect of overheads is that for fixed size problems efficiency drops by increasing the number of parallel processing units. To maintain efficiency at decent levels, solution is to proportionally increase also the problem size. Basically, scalability is a measure or a property of code and system sensitivity to resource improvement. Code and computer system that are scalable will not lose efficiency with increased number of processors. Term *strong scaling* defines how solution time varies for fixed total size problem during increase of resource power. *Weak scaling* measures variation of problem solution time with resource improvement for fixed size problem per processor. Highly scalable problems, that do not have overhead are being called *embarrassingly parallel* [14]. This is in practice often rare.

Beside overheads, other hardware and programming factors reduce parallelization benefits. Actual speed-up during parallelization is less than theoretical, since the performance values are calculated only by processor utilization time, ignoring constrains of hardware components, interconnections effects, technology cost and performance or algorithm structure. Factors that affect scalability are number of processing units, variable clock speeds of components, problem size, execution time, data input/output demand, memory capacity, communication overheads, programming and hardware costs. To achieve computational objectives, some parameters may be fixed while optimizing other factors. *Scalability analysis* is engineering procedure that helps to systematically identify critical factors of any algorithm performance [15, 28].

1.6.2 Parallelization Degree

The degree of parallelism [15] reflects how much software parallelism matches hardware parallelism. During execution, parallel program can use variable number of processing units over different time periods. Number of processors utilized at each time period is defined as the *degree of parallelization*. *Parallelism profile* is the plot of parallelism degree versus time.

1.6.2.1 Average Parallelism

In systems with multiple processors (of equal computing capacity) that during computation use various degrees of parallelism it is possible to calculate total amount of work done over some time period. *Average parallelism* [15] is total amount of work done expressed as single constant degree of parallelism (during this time period).

1.6.2.2 Available Parallelism

Parallelism degree is directly tied to type of problem. *Available parallelism* [15] characterize what degree is possible to achieve with certain problem. It is reported that data-intensive applications may have degree of parallelism from 500 to 3500, in an idealized environment, while at the instruction-level parallelism is rarely higher 7. However, the degree of parallelism may be extended to thousands in some scientific algorithms where it is possible to parallelize instructions inside basic blocks (sequence of instructions with has a single entry and exit).

1.6.3 Parallelization Speed-Up

Main goal of parallelization is to speed-up the computing process by dividing computational load over multiple processors [11, 15, 17, 28, 30]. Ideally, speed-up is equal to the number of processors that share workload. In reality, this does not happen because of negative hardware and algorithmic effects of parallelization and often, some of the program code that must remain sequential. As performance depends on multiple factors, it is not easy to formulate unified speed-up measure for all problems. To evaluate speed-up number of techniques are proposed, neither fully standardized nor agreed upon, but all serve purpose to quantitatively demonstrate efficiency of parallel designs under certain conditions.

1.6.3.1 Amdahl's Law and Gustafson's Law

Amdahl and Gustafson gave formulas to calculate theoretical speed-up of parallel program versus to sequential one. Speed-up according to Amdahl is quantified as percentage of sequential processing time for one processor, versus overall parallel execution time for fixed size of computing problem, while Gustafson's law describes theoretical speed-up in cases with increased workload. As described in [30] those two laws are related and in fact are equivalent. Often it is hard or impossible to determine required serial percentage and those laws do not reflect real speed-up achieved by parallelization. Furthermore, those laws can give misleading speed-up for algorithms that change structure when parallelized.

1.6.3.2 Memory-Bounded Speed-Up

Many scientific and engineering computations are often bound by memory capacity rather than performance of processing units. *Memory-bounded speed-up model* [31] generalizes Amdahl's law and Gustafson's law by maximizing the use of both processing and memory capacities. The idea behind this model is to solve the largest possible problem, limited only by the capacity of available memory. To achieve scalable performance, this model may result in an increased execution time.

1.6.4 Efficiency, Utilization and Quality of Parallelism

To enlarge scope of measuring performance, efficiency, utilization and quality of parallelism is measured by overall execution times instead of using the time percent. Balance between following several parameters are often good practice to achieve efficient parallel computations.

1.6.4.1 Efficiency

The speed-up factor of parallel computer system that does fixed amount of work, is expressed as execution time ratio between single-processor system and multiprocessor system. *Efficiency* [15] is for this case speed-up divided by number of processors running on parallel computer. Calculated this way, efficiency gives the degree of speed-up in comparison to theoretical (maximum) values.

1.6.4.2 Utilization and Redundancy

Redundancy [15] of system expresses the matching degree between hardware and software parallelism. Value is calculated by dividing amount of operations done by multiprocessors system with the value for single-processor system. When efficiency is multiplied by redundancy, result is overall *utilization* [15] of resources used during execution of program.

1.6.4.3 Quality

Parallelism *quality* [15] is single value used to validate speed-up of increased resources. Expression combines previous factors and is directly proportional to the speed-up and efficiency and inversely related to the redundancy.

1.6.4.4 Mean Performance

Arithmetic and *harmonic mean* performance measures [15] are used for parallel computers that execute multiple programs and in various parallelization modes (multiprocessing, vector processing, pipelining, etc.). Scientific computations are mostly organized as multiprocessor system running single programs, rendering those measures rarely significant.

1.6.5 Performance Summary

As it can be seen from previous parts of this section, models and measures that try to generalize performance of parallel computations often do not give an accurate image. To adjust performance representation to certain problems, many different approaches have been proposed. Extensive quantitative approach to performance analysis of various problems is treated in [20, 32].

At certain point scientific programmer have to make a trade-off between evaluating performance and working on actual computations. Performance models that are less accurate, but easy to examine, can often give an enough clear picture if parallelized problem computes with acceptable performance.

1.7 Efficiency Issues of Massively Parallel Designs

Parallelization consequences, hardware restrictions and programming approaches influence performance and correctness of results and program propagation. Certain issues manifest only on shared memory systems, other on distributed and some are system independent. Typical issues are overheads, dependencies and bottlenecks, caused by either hardware or algorithm.

This section classifies at what conditions common issues may occur. To achieve scalable performance, attention must be paid to avoid or resolve those problems.

1.7.1 General Issues

1.7.1.1 Overheads

Overhead [13, 17, 28] is common name for conditions that avert processor from actual computations. Algorithmic overheads or excess computations are parts of program that are difficult to or cannot be parallelized at all. In those cases, parallel algorithm can be either much more complex than sequential or it must be run sequentially on only one processor. Communication overhead (interprocess interaction) is time that processors spend on data transfer instead of computations. It includes reading, writing and waiting for data.

Idling is state of processing elements at which no useful computations are done. Reasons for idle time of processors might be synchronization requirements, overheads or load imbalance.

1.7.1.2 Bottlenecks

As seen from previous sections, data transfer often downgrades processor performance. In multiprocessor environments, memory bandwidth is aggregated with adding more nodes, but for multi-core it is harder to achieve desired level of scalability. Situation where processor speed gets limited as result of insufficient increase of memory bandwidth is called *memory wall* [17]. Computations where performance is constrained by transfer rates of memory are referred as *memory bound* [28].

Techniques to overcome memory performance problems are named latency hiding, tolerating or reducing mechanisms. Those techniques and mechanisms [15] (memory consistency, cache coherence, prefetching, cache and coherence misses and other) are actually very important, but are concern only to engineers who build multiprocessor systems. In general, (scientific) programmer should properly balance computations with data transfer and not be occupied with low-level implementation and issues.

In multiprocessor systems bottlenecks [11, 14] can also be caused by poor load balance, where whole system waits idle for one processor to finish computations.

1.7.1.3 Dependencies

Often for certain computations to advance, data from previous iterations is required. Such cases are called data dependent [14, 15]. Parallelization is interrupted by task that must wait for data to continue. Other types of dependencies exist, but are not significant from scientific programmer perspective.

1.7.2 Issues on Shared Memory Systems

1.7.2.1 Result Preservation

In parallel program, order of execution is changed in comparison to sequential one [29]. Some operations that are sequentially executed one after another, now may be executed parallelly. Order of execution can cause change of accuracy on some systems due to number truncation or rounding off on different places in program. To avoid such errors, it is required to check if results are consistent during change in order of execution.

1.7.2.2 Synchronization Errors

Very common errors manifest on shared memory systems when threads are not time synchronized [29]. Program executes correctly, since there is no actual bug, but results may be incorrect. Timing of threads impacts if those errors happen or not, even in cases when synchronization is not explicitly imposed.

First type of incorrect results may occur when one thread starts to work on data from another thread that have not yet finished with computations. A barrier can be instructed to ensure that the thread will wait for another one to finish.

Result inconsistency can also happen when a thread is scheduled to work on part of data already processed by another thread. To avoid this issue, programmer can explicitly protect access to data of another thread.

Race condition is a conflicting state when multiple threads try to simultaneously update same variable. Without access synchronization variable update may not be as it is intended. To resolve race conditions, reading and writing to a shared variable should be enclosed in critical section that permit only one thread at time to manipulate data. Critical sections are implemented as *atomic instructions*. One atomic instruction is sequence of instructions that manipulate shared variable. At any given time, only one atomic instruction is allowed to be executed by whole computer system and it must be executed entirely before any other instruction.

1.7.2.3 Variable Scope Issues

Certain variables may be declared to be shared between all threads or private to each thread. If declared incorrectly shared variable may not be updated when necessary and private may be updated by wrong thread [29].

1.7.3 Issues on Distributed Memory Systems

1.7.3.1 Deadlocks

Situation when two or more processes keep waiting for resource from each other and none of them can make any progress [13, 14, 18]. Resource in those cases can be message send/receive operation, synchronization instructions or access to remote device or memory. Conditions under which deadlocks happen are when resource is mutually exclusive (only one process may use it), when process that use a resource requests another resource (hold and wait condition), in cases when resource cannot be released without process action (no pre-emption condition) and when multiple processes in circular chain wait a message from another one.

Deadlock detection [33] is analysis conducted to reveal if deadlock conditions apply to set of resources and processes. Prevention and avoidance ensure that deadlock conditions do not hold and will not occur upon resource utilization. To prevent mutual exclusion, resource should be available to multiple processes. Hold and wait condition may be eliminated by forcing the process to release all resources upon request for another or to acquire all resources with single operation. No pre-emption is eliminated by allowing a resource to be released from process. Circular chain waiting is prevented by imposing an order of resource employment.

1.7.3.2 Livelocks

Similar situation to deadlock, where two or more processes fail to progress because all keep responding to each other request indefinitely [17, 33]. It is resolved by giving respond priority to one of the process.

1.7.3.3 Busy Waiting

Occurrence when one process sends a message to another process that continuously denies it or when process constantly tests if a condition is satisfied [33]. The first process keeps resending the message or checking the condition state and cannot continue with useful work.

1.7.3.4 Starvation

In cases where processes are scheduled by another entity, starved process [17, 33] is one that is ready to continue, but scheduler ignores it.

1.8 Parallel Program Design

In multi-core environments there is no certainty that some parallel operations will be executed at exactly same time or in exactly specified order without losing parallelization or introducing idle waiting time. For example, in parallel program one core might supply certain data to another core too late or too early. The program code does not report any error (since there is none), yet it produces incorrect output due to loss of data coherency or processors stay unnecessary idle. De-bug difficulty presents that on certain runtime program will work and on another runtime, it will go into a dead-lock even with same data, depending how threads are scheduled by operating system [13, 14, 17, 18, 27]. Step-by-step debugging and data tracing is therefore not feasible solution for checking errors and flow of a parallel program. Much more attention must be invested in methodical design to prevent dead-locks, incoherent data operations and other dangers of parallelization. Design methodology will be described in order to minimize parallel program flaws and bugs due to common bad practice during programming while maintaining decent level of simplicity and efficiency [18, 27]. More detailed approach to design of parallel algorithms may be found in [13, 14, 20]

1.8.1 Parallel Program/Algorithm Properties

1.8.1.1 Concurrency Versus Parallelism

Often confused for same, but fundamentally different procedures of instructing commands [17, 18, 27]. Concurrency enables several different threads to be open for execution by single processor. This means that while one thread is executed (sequentially), other threads stay idle and processor can arbitrary switch between execution of threads.

In parallelism, every thread is executed on dedicated processor in the same time, resulting that (ideally) there is no idle time for any thread.

1.8.1.2 Locality

Ratio of local memory accesses to remote memory accesses is defined by locality of program [27]. It is one of the major properties of parallel programs which can lead to efficiency loss due to communication constraints.

1.8.1.3 Modularity

Property of program (and in some cases hardware) to be composed of number of smaller, independent units (modules) is in general good practice in software engineering, both sequential and parallel [11, 27].

1.8.2 Design Methodology

To get from problem specification to effective parallel algorithm it is crucially important to rely on design methodology than on pure creativity of programmer. Anyway, creativity is of great importance even when following methodical approach. It allows to increase range of considered options, distinguish bad from good alternatives and to minimize backtracking from bad choices. Design flaws easily compromise parallel program performance. As most of problems have multiple parallelization possibilities, methodical design helps to characterize most favourable solution [13, 14, 18, 27, 28].

1.8.2.1 Partitioning (Problem Decomposition)

First step before starting to design a parallel program, is to determine if problem is inherently parallel by its nature or it may or may not be parallelized [14, 27, 28]. To be parallelizable an algorithm must satisfy certain conditions. Detecting methods serve to discover if some algorithm is parallel when it might not be obvious.

Partitioning serve to recognize parallelization opportunities in the problem. Data and operations are decomposed into smaller (independent when possible) tasks. If decomposed tasks are not independent, they have to communicate data according to dependencies. *Domain decomposition* is technique where data set of problem is divided into independent pieces. Next step is to associate tasks to partitioned data set. Complementary technique, the *functional decomposition*, focuses on dividing the computations into smaller disjoint tasks. In case when decomposed tasks correspond to partitioned data, decomposition is complete. By nature of many problems this is not possible. In those cases, replication of data or task set must be considered. Even when it is not necessary, it might be worth to replicate data or instructions to reduce communication. Several guidelines should be considered before proceeding to next steps of parallelization, to ensure that there are no obvious design flaws:

1. to increase flexibility in following design stages, there should be at least one order of magnitude more tasks than processors in the system and
2. consider both decomposition techniques and identify alternative options,
3. scalability can be compromised with larger problems if there are redundant computations or data input/output after partitioning,
4. it is hard to allocate equal amount of load to each processor if tasks are not comparable in size,
5. to properly scale, with increase in problem size, number of task should grow, rather than size of individual task.

1.8.2.2 Communication Design

Flow of data is specified in communication stage [27] of design. In general, tasks can execute parallelly, but it is rare that they are independent. Proper communication structures are required to efficiently exchange data between parallel tasks. Goal of communication design process is to allow efficient parallel execution, by acknowledging what communication channels and operations are required and eliminating those which are not necessary. Communication channels for parallel algorithms obtained by functional decomposition correspond to the data flow between tasks. For domain decomposed algorithms, data flow is not always straightforward, since some operations might require data from several tasks.

Local communication structures are used when task communicate only with small number of neighbouring tasks. Global communication protocols are more efficient when many tasks communicate with each other. Communication networks can be structured, where tasks form a regular composition and unstructured where task are arbitrarily arranged. If identity of communication pairs varies during program execution, communication is dynamic and for unchangeable identities, it is static. In synchronous communication information exchange is coordinated, but for asynchronous communication structures, data is transferred with no mutual cooperation.

The following check-list is proposed to avoid overheads and scalability issues arising from inefficient communication layout:

1. for scalable algorithm, all tasks should perform similar number of communication operations,
2. when possible, arrange task so that global communication can be encapsulated in local communication structure,
3. evaluate if communication operations are able to proceed parallelly,
4. evaluate if tasks can execute parallely and does communication prevents any of the tasks to proceed.

1.8.2.3 Agglomeration

One of the principal requisites for efficient execution is level of matching between hardware and software. In agglomeration stage [27] algorithm from previous phases is adapted to be homologous to the computer system used for computation. It is known that it is useful to combine (agglomerate) large number of small tasks into fewer task larger in size or to replicate either data or computation. Reduced number of tasks or replication can substantially reduce communication overheads.

Revision of parallel algorithm attained by decomposition and communication design phase can be optimized by following agglomeration procedure:

1. reduce communication cost by increasing task locality,
2. verify that benefits outweigh the costs of replication or limit scalability,
3. task created by agglomeration should have similar communication costs as single smaller task,
4. evaluate if agglomerated algorithm with less parallel opportunities execute more efficiently than highly parallel algorithm with large communication costs,
5. check if granularity (size of tasks) can be increased even further, since fewer large task are often simpler and less costly,
6. evaluate modification costs of parallelization and strive to increase possibilities of code reuse.

1.8.2.4 Mapping

Final stage of design is to decide how to map task execution on processors [27]. Since there is no universal mechanism to assign set of tasks and required communications to certain processors, two strategies are used to minimize execution time. First option is to map tasks to different processors in order to increase parallelization level. Other option that increase locality, is to map tasks that communicate often to the same processor. Those strategies are conflicting and a trade-off must be made to achieve optimal performance. Favoured strategy is problem specific and use of task-scheduling or load balancing algorithms can be used to dynamically manage task execution.

1.8.3 Design Evaluation

Before starting to write actual code, parallel design should be evaluated according to few criterion. Some simple performance analysis should be conducted to verify that parallel algorithm meets performance requirements and that is the best choice among available alternatives. Also, to be considered are the economic costs of implementing and possibilities for future code reuse or integration into larger system [27, 28].

1.9 Software Solutions

Software solutions are the connection between hardware platforms and computing problems. Various libraries, frameworks and APIs (application programming interfaces) are added to backbone programming languages like C/C++ or Fortran. Functionality of parallelization software is developed to be independent from backbone languages. To exploit parallel resources, programmer only need to call (parallelization) sub-routines from one of the supported programming language [13, 18].

This section explains principal functionality (concepts) of major software solutions for science applications. Logic behind those software platforms is clarified and for actual coding tutorials readers are invited to use specialized programming literature.

1.9.1 OpenMP

OpenMP (Open Multi-Processing) [13, 18, 34] is an API that supports programming on shared memory computers in C, C++ and Fortran languages. It offers intuitive, multi-threading method of parallelization, where one main thread (master) forks when parallelizable part of code is encountered. Work is then divided among number of secondary (slave) threads. There can also be multiple levels of forking. Threads of same level execute same code over designated portion of total data. It is usually used in combination with other parallel software when is possible to parallelize work inside nodes.

1.9.2 Message Passing Interface (MPI)

MPI [13, 18, 35] is a standard that defines syntax and semantics of library routines used for writing message-passing programs in C, C++ and Fortran. It operates on variety of parallel architectures, but is major standard for programming of distributed memory systems, such as clusters. Message passing with MPI is not so intuitive approach to parallel programming and requires more attention than multi-threading approach with OpenMP.

Parallelization is achieved by creating one master and numerous slave tasks (*ranks*) at program runtime. Each rank runs own instance of MPI program. Within code it is specified what parts are executed or skipped by certain ranks. In this way, each rank has own instance of data structures that are not shared with other ranks (although the structures are declared under the same name). To access some remote data, rank have to explicitly request it.

Core of MPI is based on communication by passing messages between ranks. The simplest form of information exchange is by send/receive operations. One rank would request some data, other rank has to acknowledge this request and send required data back. First rank then has to appropriately receive the message containing the requested information.

Beside point-to-point communications as send/receive, there are collective class of communication operations. When large number of ranks have to exchange data it is much more efficient to use collective message passing.

Synchronization of execution can be explicitly imposed by instructing barriers or implicitly by using blocking communication. Neither rank is allowed to proceed until all ranks execute the explicit barrier instruction. Blocking communication prevents receiving rank to continue until message is received. Asynchronous communication can be achieved by using non-blocking message passing or by using probe instructions. With probe instructions, ranks check if there is pending message. When message is there, probing rank receives it, when not, rank continues with execution.

Communicator is a structure that defines communication privileges. It is used to specify what ranks will participate in certain communication operations.

Functionality of message passing makes it appropriate for programming of MIMD problems in HPC. Technique often used is *hybrid programming*, where MPI and OpenMP are used together.

1.9.3 CUDA and OpenCL

Compute Unified Device Architecture (CUDA) [13, 18, 23] is programming environment developed to efficiently map data parallel task to GPU structure. GPU program is separated in parts run by CPU (*host*) and data intensive functions (*kernels*) that are executed on GPU (*device*).

Beside memory allocation that hold transfer of data between CPU and GPU memories, programmer has to specify how threads are organized inside kernel. Kernel *grid* is organized in two levels. Top level is organization of thread *blocks* within the grid. On second level threads are arranged inside block. Each block of same grid has same number and structure of threads. Latest GPUs support three-dimensional organization of threads within block.

Execution configuration of kernel is further divided into smaller units *wraps*, that are collection of threads which executes at once. Mechanism called thread *scheduler* decided which wrap will be executed. This execution model efficiently exploits memory and core organization of GPU even in cases when programmer poorly organize kernel grid and memory allocation.

Kernel execution requires large amounts of data and access to it is very time-expensive. *Memory coalescing* is a technique that combines neighbouring data and copies it together from slow global to fast shared register memory. To exploit memory coalescing, programmer should organize data so that neighbouring threads in wrap use equally organized data in memory (consecutive threads should use consecutive memory locations). Technique that help to methodically arrange data according to thread execution schedule is called *tiling*. It enables also to efficiently reuse data or pre-load piece of data for faster access.

Open Computing Language (OpenCL) [13, 18, 36] is cross-platform programming environment that provide standardized support for computers with multiple processors, GPUs and other computing units. It provides methods to efficiently assign tasks and exploit all resources of heterogeneous computing platforms. Execution model of OpenCL programs are slightly more complex, but very similar to CUDA.

1.9.4 Other HPC and Scientific Software

The above mentioned software are not only solutions available on market, but are widely used and supported by user community. Other software such as job schedulers, node installation and management, integrated stacks and monitoring programs are more concern of system administrators than to programmers. For larger jobs it might be useful also to get familiar with load balancing, task scheduling and management [32].

Most HPC platforms are never dedicated to only one computing job. Resources (HPC time) are shared/distributed between large number of scientific and industry projects. It is worth to mention that large amount of HPC time is dedicated to *Computational Fluid Dynamics*. Although it is a nonlinear phenomena, it is not focused on global analysis but on modeling the motion of specific problems, such as interaction of fluids with a solid surface. Reference [37] presents an example of scientific applications available at one of the supercomputers from Top500 list.

1.10 HPC in Global Analysis

It is assumed that reader is familiar with the basic of nonlinear dynamics and is looking to parallelize his/her own computations. This section is thus, dedicated to introduction to some of main methods of global analysis, to present possible parallelization options and to provide examples of how it may be accomplished.

1.10.1 Numerical Global Analysis Methods

As computers are able to manipulate and store numbers with limited number of digits, numerical methods operate with somehow discretized continuous state space. There are two types of methods, classified according to discretization methodology. First class are methods that treat state space as collection of points. Second class of methods divide state space into number of hyper-cubes (*cells*).

1.10.1.1 Point Integration

Straightforward method to get a basin of attraction is to take a cross-section of continuous state space and approximate it with set of points. To determine basin, every point is evolved by integrating forward in time. It is assumed that orbit converges if, after certain amount of time, distance from it to one of the one of the attractors is within some small predefined tolerance [10]. This method often requires integration of trajectories for extensive amount of time, that depends on the length of the transient motion [38].

1.10.1.2 Point Mapping

Instead of looking for the continuous time history of a motion of the system, it is also possible to track the system state at a sequence of discrete time instants. Methods how to approximate continuous trajectories with point mapping depend on the type of system and nature of analysis. Result of discretization is mapping function where discrete trajectory is formed by iterating the map, starting from an initial state [1].

1.10.1.3 Basins of Attraction

Theory of point mapping allows to determine equilibrium points, periodic motions and strange attractors, associated with continuous counterpart. To determine basins for stable sets there are several methods. Direct approach investigates where each initial condition maps after certain number of iterations. Time saving method consists of starting from small known region around an attractor and then expand the boundary. For certain two-dimensional maps it is possible to separate basins by using stable manifolds of saddle points.

1.10.1.4 Grid of Starts

Integration of grid of points (or grid of starts) [39] is a method related to the point integration. Basins are determined in the same manner, by integration from initial condition to the attractor. The initial condition in this case is a cell, commonly its center point. In other words, all states residing inside the cell are approximated as single cell entity. Although this method give fairly accurate results, the drawback is high computational requirements [40] because of the long integration time needed to overcome transient.

1.10.1.5 Cell Mappings

Closely related to the point mapping, cell mapping methods also approximate continuous trajectories by a discrete map. Starting from state space discretized as in Sect. 1.10.1.4, each cell is enumerated with an positive integer number. Initial conditions are then integrated over small time period to obtain the mapping. An image (or image cell) is a cell where the initial cell is mapped after one step of a map. It is possible to use methods analogue to point mapping to determine attractors and corresponding basins. As cell mapping is less computing intensive than fore-mentioned methods, and over the years, various cell mapping methods were developed, each with its advantages and drawbacks.

The *Simple cell mapping* (*SCM*) method [1] assumes that each cell can have only one image cell. If a cell maps to itself, it is considered to be a periodic cell. When certain cell maps to itself after multiple map steps, all cells in the sequence form a periodic motion. Basin of attraction is then a collection of cells that after arbitrary number of map steps get mapped to some periodic cell or motion. With SCM it is not possible to accurately approximate chaotic attractors or fractal basin boundaries, but it can give a hint if those are present in the dynamical system. E.g. a chaotic attractor in SCM can be recognized as a several periodic motions with a very long period. Advanced cell mapping methods can be used to overcome obstacles of SCM, but at certain computational costs.

The *Generalized cell mapping* (*GCM*) method [1] improves SCM by incorporating more information on system dynamics inside the map. Function that governs the map evolution is based on probability of each cell to map into one of the possible image cells. Such formulation of GCM mapping leads to the finite Markov chains for which well developed theory enables identifying the dynamics of the system. GCM is effective for discovering occurrence of chaos in the system.

The *Interpolated cell mapping* (*ICM*) [39] is a method developed to combine good characteristics of SCM and GCM. In SCM trajectory endpoint is rarely in the center of image cell. ICM records beside image cell also the actual location of endpoint inside image cell. Next iteration is calculated by taking the relative position of terminal point in respect to four surrounding cells. The end point of this iteration is obtained by interpolating between endpoints of the trajectories emerging from fore-mentioned four surrounding cells. In comparison to SCM and GCM, this method gives more accurate trajectories, but requires additional computational costs for computing new terminal positions.

Cell mapping methods are particularly suitable for systems with a periodic excitation. In this case the map is the stroboscopic one, sampled at the period of the excitation.

1.10.2 Multi-dimensional and Parallelized Numerical Global Analysis Methods

Before-mentioned methods work well in lower dimensions, but do not scale so well with increase in system dimension. To overcome dimensionality restrictions in [41] the authors developed a multi-degree of freedom (MDOF) extension to cell mapping methods. Beside dimensionality, another, difficult task is to overcome inner seriality of those methods. Survey of successful attempts to parallelize global analysis methods follows.

In [42] MDOF cell mapping algorithm is restructured to exploit most time consuming part of global analysis – the system integration. In a series of papers [43–45] the authors examined parallelization of grid of starts method on cluster computers. Successfully computed basins of attraction in both previous cases are used to determine integrity measures of dynamical systems.

To exploit massive parallelization capabilities of GPU, in [46] the authors refined the SCM method with subdivision techniques, in order to solve problems in multi-objective optimization. Another GPU parallelization approach on global analysis is tackled in [47] that combined several cell mapping methods (SCM, GCM and ICM). This combined method gives fast and resource efficient method to discover attractors, but require additional computations to determine basins. Authors provide example applications to impact model, plasma model and six-dimensional Lorenz system.

1.10.3 Example of Global Analysis

To illustrate computing of basins with SCM we used the four-dimensional system of two coupled and driven Duffing-Van der Pol oscillators considered in [48], governed by the equations:

$$\begin{aligned}
\dot{x}_0 &= x_1,\\
\dot{x}_1 &= \nu(1 - {x_0}^2)x1 - {\omega_1}^2 x_0 - \epsilon {x_0}^3 + \beta x_2 + Fsin(t),\\
\dot{x}_2 &= x_3,\\
\dot{x}_3 &= \nu(1 - {x_2}^2)x3 - {\omega_2}^2 x_2 - \epsilon {x_2}^3 + \beta x_0 + Fsin(t + 0.25),
\end{aligned} \tag{1.1}$$

with $\omega_1 = 0.000023216854686$, $\omega_2 = 0.022222854255$, $\nu = 0.25$, $\beta = 0.01$, $\epsilon = 1$, and $F = 1.4$.

Calculations are carried out with two different resolutions to demonstrate accuracy and performance of the method. For the low resolution the state space region $x_i = (-4, 4)$ is divided into 110 intervals per dimension, totaling 146 410 000 cells and size cell $h_i \approx 0.073$. Significantly higher number of cells, 1 632 240 801 is achieved by dividing each dimension into 201 intervals, here having size cell $h_i \approx 0.039801$.

Initially, the system (1.1) is integrated with a parallelization process on a small cluster using MPI standard implemented in C programming language. Then, the four-dimensional basins of attraction are determined with SCM post-processing algorithm. Low resolution case is calculated on two different computers. On the one with lower performance integration time was approximately 4 h and basin search lasted about 15 min. Then, the same calculations are carried out on computer with four times more processors. Result was that integration stage scaled well reducing computing time to around 1 h, but result of increased communication in post-processing stage increased time to build basins to 30 min.

Bad scaling of post-processing is even more evident in the high resolution case, where integration lasted 6 h and post-processing 12 h. From achieved performance it is obvious that integration stage can be considered as parallelizable, since it scales well. On the other hand, SCM post-processing is highly inadequate to be computed on distributed memory systems where large amount of communication operations drastically degrade performance.

In Figs. 1.6a and 1.7a we report the $x_0 - x_1$ and $x_2 - x_3$ basins cross-sections (other coordinates are fixed to 0) of low resolution and on Figs. 1.6b and 1.7b of high resolution case. To synthetically present usefulness of full-dimensional basins we report in Figs. 1.8 and 1.9 how the $x_0 - x_1$ cross-section of basins changes as the x_3 coordinate is varied.

Although the fastest and least resource consuming method, the drawback of pure SCM computations is that there is the possibility that some basins and attractors are assimilated as a result of the low accuracy of approximated trajectories, especially at low resolutions.

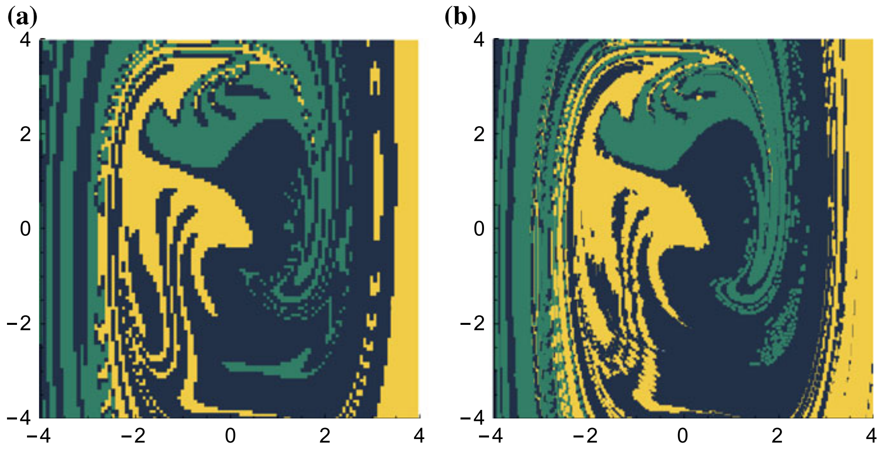

Fig. 1.6 $x_0 - x_1$ basin cross-section of **a** low and **b** high resolution cases

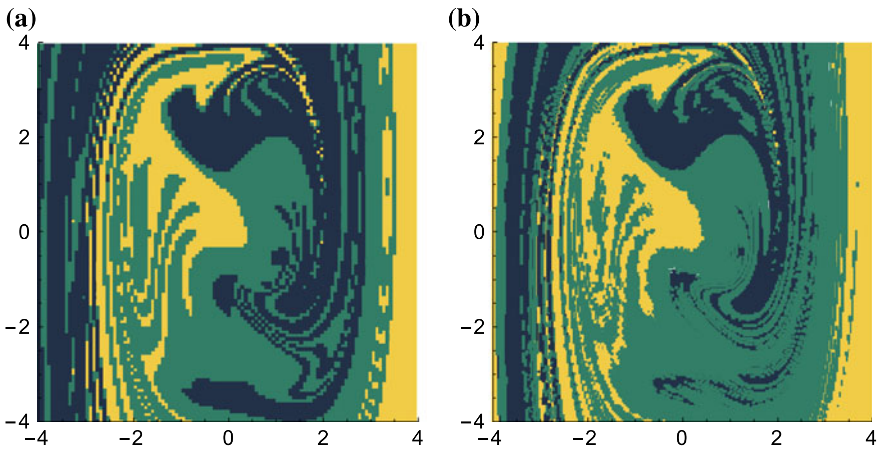

Fig. 1.7 $x_2 - x_3$ basin cross-section of **a** low and **b** high resolution cases

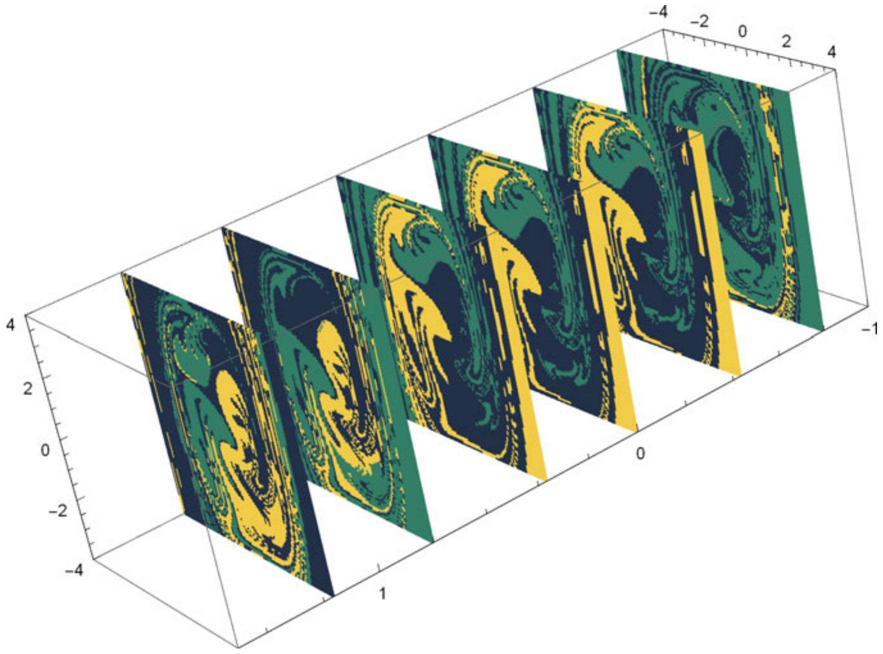

Fig. 1.8 Various $x_0 - x_1$ basin cross-sections of low resolution case

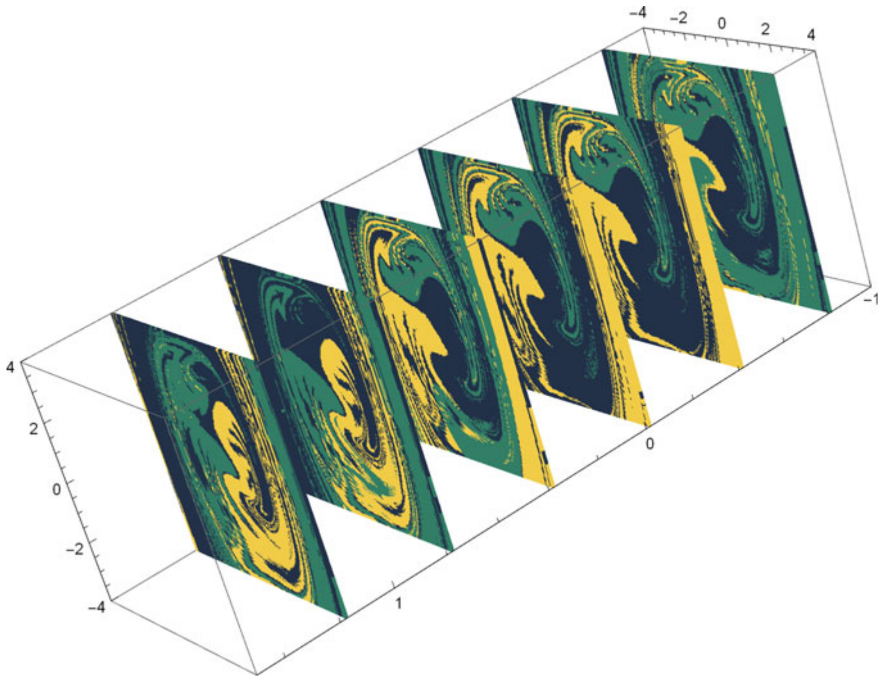

Fig. 1.9 Various $x_0 - x_1$ basin cross-sections of high resolution case

1.11 Conclusion

Vast amount of scattered literature and examples is available and newcomers often get lost. In this chapter we hope to have introduced scientists and engineers to topics of high power-computing in step-by-step fashion to all concepts, implementations and methods required to understand how to efficiently solve large computing problems.

Performance of individual components is explained in enough detail that is required to understand overall system performance. With concepts of hardware organization (architectures) and different implementations readers were introduced to world of massively parallel computing. From mentioned concepts of high-power computing, high performance computing with clusters and GPU is most relevant for science and engineering.

In design process, programmers now understand why should strive to make parallel software scalable, as local as possible and modular. It is possible to achieve this task by methodical development of algorithm through partitioning the problem, optimizing communication structures and agglomeration of communication intensive parts.

Performance issues and parallelization speed-up are discussed to prevent unrealistic expectations from parallel computing. Readers were also introduced to functionality of main software solutions in HPC, OpenMP for shared memory computers, MPI standard for distributed systems and OpenCL and CUDA for environments with computational GPUs.

Topic is closed by presenting common methods and applications of global analysis. After studying this paper, it is hoped that reader is able to identify type of computing problem, to choose proper hardware/software platform, methodically plan design process and evaluate parallel algorithm by referring to specific literature for deepening on specific topics.

Acknowledgements NA and SL would like to thank Radu Serban and Dan Negrut, University of Wisconsin-Madison, USA, for help with HPC and for the kind hospitality during the visit of NA.

References

1. Hsu, C.S.: Cell-to-Cell Mapping: A Method of Global Analysis for Nonlinear Systems. Springer, New York (1987)
2. Sun, J.-Q., Luo, A.C.J. (eds.): Global Analysis of Nonlinear Dynamics. Springer, New York (2012)
3. Category: History of computing hardware—Wikipedia, the Free Encyclopedia. Accessed 15 Nov 2017
4. Disrupting the datacenter: Qualcomm Centriq$^{\mathrm{TM}}$ 2400 processor. Accessed 15 Jan 2018
5. Null, L., Lobur, Julia: The Essentials of Computer Organization and Architecture. Jones and Bartlett Publishers, Sudbury, Mass (2003)
6. Hilborn, R.C.: Chaos and nonlinear dynamics: an introduction for scientists and engineers, 2nd edn. Oxford University Press, Oxford (2000)
7. Strogatz, S.H.: Nonlinear Dynamics and Chaos: With Applications in Physics, Biology, Chemistry, and Engineering. Mass, Wokingham, Addison-Wesley Pub, Reading (1994)
8. Jain, M.K., Iyengar, S.R.K., Jain, R.K.: Numerical Methods for Scientific and Engineering Computation. Wiley Eastern Ltd., New Delhi etc. (1986)
9. Aguirre, J., Viana, R.L., Sanjuán, M.A.F.: Fractal structures in nonlinear dynamics. Rev. Mod. Phys. **81**, 333–386 (2009)
10. Engelina Nusse, H., Hunt, B.R., John Kostelich, E., Yorke, J.A.: Dynamics: numerical explorations. In: Applied Mathematical Sciences, 2nd edn. Springer, New York (1998)
11. Comer, D.: Essentials of Computer Architecture, 2nd edn. Chapman and Hall CRC (2017)
12. Elahi, A.: Computer Systems: Digital Design. Fundamentals of Computer Architecture and Assembly Language. Springer, Cham (2018)
13. Czarnul, Pawel: Parallel Programming for Modern High Performance Computing Systems. CRC Press, Taylor and Francis Group (2018)
14. Aubanel, E.: Elements of Parallel Computing. Chapman and Hall CRC (2016)
15. Hwang, K.: Advanced Computer Architecture: Parallelism, Scalability, Programmability. McGraw-Hill Series in Computer Engineering. McGraw-Hill, New York (1993)
16. Clements, A.: Principles of Computer Hardware, 4th edn. Oxford University Press, Oxford, New York (2006)
17. Padua, D. (ed.): Encyclopedia of Parallel Computing. Springer US (2011)
18. Rauber, T., Rnger, G.: Parallel Programming for Multicore and Cluster Systems, 2nd edn. Springer, Berlin, Heidelberg (2013)
19. Jiao, Y., Lin, H., Balaji, P., Feng, W.: Power and performance characterization of computational kernels on the gpu. In: Green Computing and Communications (GreenCom), 2010 IEEE/ACM International Conference on Cyber, Physical and Social Computing (CPSCom), pp. 221–228, Dec 2010
20. Hennessy, J.L., Patterson, D.A., Asanovi, K.: Computer architecture: a quantitative approach, 5th edn. Morgan Kaufmann/Elsevier, Amsterdam, Boston (2012)

21. The Top500 List—November 2017. Accessed 11 Feb 2018
22. The Green500 List—November 2017. Accessed 11 Feb 2018
23. CUDA zone. Accessed 15 Jan 2018
24. Sourouri, M., Langguth, J., Spiga, F., Baden, S.B., Cai, X.: Cpu+gpu programming of stencil computations for resource-efficient use of gpu clusters. In: 2015 IEEE 18th International Conference on Computational Science and Engineering, pp. 17–26, Oct 2015
25. Chakrabarti, S., Demmel, J., Yelick, K.: Modeling the benefits of mixed data and task parallelism. In: Proceedings of the Seventh Annual ACM Symposium on Parallel Algorithms and Architectures, SPAA '95, pp. 74–83. ACM, New York, NY, USA (1995)
26. Raicu, I., Foster, I.T., Zhao, Y.: Many-task computing for grids and supercomputers. In: 2008 Workshop on Many-Task Computing on Grids and Supercomputers, pp. 1–11, Nov 2008
27. Foster, Ian: Designing and Building Parallel Programs: Concepts and Tools for Parallel Software Engineering. Addison-Wesley Longman Publishing Co., Inc, Boston, MA, USA (1995)
28. Grama, A.: Introduction to Parallel Computing, 2nd edn. Addison-Wesley, Harlow, England, New York (2003)
29. Solihin, Y.: Fundamentals of Parallel Multicore Architecture, 1st edn. Chapman-Hall, CRC (2015)
30. Shi, Y.: Reevaluating Amdahl's law and Gustafson's law (1996)
31. Sun, X.-H., Ni, L.M.: Institute for computer applications in science, and engineering. In: Scalable Problems and Memory-Bounded Speedup. ICASE Report. Institute for Computer Applications in Science and Engineering, NASA Langley Research Center, Hampton, Va. (1992)
32. Pllana, S, Xhafa, F. (eds.) Programming Multi-core and Many-core Computing Systems, 1st edn. Wiley Publishing (2014)
33. Stallings, W., Paul, G.: Operating Systems: Internals and Design Principles, 7th international edn. Pearson, Boston, Mass, London (2012)
34. The OpenMP API specification for parallel programming. Accessed 15 Jan 2018
35. Message Passing Interface (MPI) Forum. Accessed 15 Jan 2018
36. The OpenCL^{TM} specification. Accessed 15 Jan 2018
37. Cineca SCAI application software for science. Accessed 01 June 2018
38. Medio, A., Lines, M.: Nonlinear Dynamics: A Primer. Cambridge University Press (2001)
39. Tongue, B.-H., Gu, K.: Interpolated cell mapping of dynamical systems. J. Appl. Mech **55**(2), 461–466 (1988)
40. Ge, Z.-M., Lee, S.-C.: A modified interpolated cell mapping method. J. Sound Vib. **199**(2), 189–206 (1997)
41. Spek, J.A.W., van der, D.H. Campen, V., Kraker de, A.: Cell mapping for multi degrees of freedom systems. In: Bajaj, A.K. (ed.) Nonlinear and Stochastic Dynamics: Presented at 1994 International Mechanical Engineering Congress and Exhibition, Nov 6–11, 1994, pp. 151–159. Chicago, Illinois, AMD, ASME (1994)
42. Eason, R.P., Dick, A.J.: A parallelized multi-degrees-of-freedom cell mapping method. Nonlinear Dyn. **77**(3), 467–479 (2014)
43. Belardinelli, P., Lenci, S.: A first parallel programming approach in basins of attraction computation. Int. J. Non-Linear Mech. Dyn. Stab. Control Flexible Struct. **80**, 76–81 (2016)
44. Belardinelli, P., Lenci, S.: An efficient parallel implementation of cell mapping methods for mdof systems. Nonlinear Dyn. **86**(4), 2279–2290 (2016)
45. Belardinelli, P., Lenci, S.: Improving the global analysis of mechanical systems via parallel computation of basins of attraction. In: Procedia IUTAM, IUTAM Symposium on Nonlinear and Delayed Dynamics of Mechatronic Systems, vol. 22, pp. 192–199 (2017)
46. Fernndez, J., Schtze, O., Hernndez, C., Sun, J.-Q., Xiong, Fu-Rui: Parallel simple cell mapping for multi-objective optimization. Eng. Opt. **48**(11), 1845–1868 (2016)
47. Xiong, F., Qin, Z.-C., Ding, Q., Castellanos, C.H., Fernandez, J., Schetze, O., Sun, J.Q.: Parallel cell mapping method for global analysis of high-dimensional nonlinear dynamical systems, vol. 82 (2015)
48. Battelino, P.M., Grebogi, C., Ott, E., Yorke, J.A., Yorke, E.D.: Multiple coexisting attractors, basin boundaries and basic sets. Phys. D **32**(2), 296–305 (1988)