The multi-dimensional model of grid computing

Viera Šípková
Ústav informatiky, Slovenská akadémia vied
845 07 Bratislava, Dúbravská cesta 9

Abstract. Recently processors have added explicit parallelism in the form of multiple cores in a chip, and the number of cores is predicted to be increasing exponentially over time. Along this line processors will continue to scale in performance, though processors themselves are no longer significantly scaling their clock rate. Hence, software applications that are not parallelised maximum as possible will not benefit. Program codes are required to exploit multiple heterogeneous cores on a chip, multiple processors within a cluster, and additional processors and clusters available across a network and in a grid. In order for applications to take advantage of this hybrid hardware the investigation of new algorithms and programming models that are capable to abstract the underlying architecture, is fundamental and crucial. In this paper we focus on different software techniques and programming models that we found to be most useful in the development of parallel and distributed applications designed to run on heterogeneous hardware resources like multicore machines, clusters, and grids.

1 Introduction

Recently processors have added explicit parallelism in the form of multiple cores in a chip, and the number of cores is predicted to be increasing exponentially over time. The forecast is to include hundreds or thousands of them within a few years. What is more, in the near future chips will be of highly heterogeneous organisations, with complex memory hierarchies and different types of cores specialised for particular tasks (like floating point accelerators, graphics processing units, etc.). Along this line processors will continue to scale in performance, though processors themselves are no longer significantly scaling their clock rate. Hence, software applications that are not parallelised maximum as possible will not benefit. Program codes are required to exploit multiple heterogeneous cores on a chip, multiple processors within a cluster, and processors and clusters available across a network and in a grid. Each core may also include an additional parallelism at the instruction level. The second important observation is that managing memory hierarchies is critical too, memories are to be managed so that cores are not starved for data. Terms concurrency, locality, modularity, scalability, and portability become fundamental pre-requisites in the algorithm and application design. In the process of the software development the selection of an appropriate programming model which is capable to abstract the underlying architecture, is crucial. New methodologies and mechanisms enabling the highly parallel execution of applications need to be investigated. And even more, tools and frameworks providing the automatic code parallelisation, and performing data transfers between different levels of memories, all in a very dynamic fashion, should be available. Regarding the grid computing, even though we consider a world
abounded with more powerful local computer systems, the demand for grids and grid software will still remain, because of data sources and communities will continue to be distributed. These forthcoming grids will consist of more complex nodes, but the community will be surely competent enough to conceive applications that will call for these great computing facilities.

This work outlines some of the best practices applied in the development of parallel applications; they originated mostly from the area of high performance computing, but they can be combined with modern technologies to obtain a powerful solution satisfying user requirements for correctness and performance.

2 Principles of Parallel Programming

According to opinions of S. Akhter and J. Roberts [1] (both from Intel): “The entire concept of parallel programming centers on the design, development and deployment of threads within an application and the coordination between threads and their respective operations”. Threads enable multiple operations to proceed simultaneously. Moving from the traditional sequential model to a parallel programming model calls for rethinking the idea of process flow: those activities that can be executed in parallel must be detected and the program should be expressed as a hierarchical set of tasks with specifying dependencies between them. Breaking a program down into individual tasks including also the definition of task dependencies is known as program decomposition. In general, there exist many ways how to decompose a program, following different criteria and strategies, though basic forms are the next:

- **Decomposition by task** - many threads, where different activities are assigned to different threads. The problem is broken down by the functions that it performs. Tasks which can run concurrently are scheduled to do so.

- **Decomposition by data** - multiple threads performing the same work just on different data items. Data decomposing, also called data-level parallelism, breaks down a problem by the data it works on.

- **Decomposition by data flow** - many threads, where one thread’s output serves as an input to another thread. The problem is broken down by how data flow between tasks. An example of this type is the producer-consumer problem.

Different decompositions provide different benefits and have various implications. The choice of the decomposition strategy is dictated mostly by the problem domain, some jobs are much better suited to one type of decomposition, and another jobs have no clear bias. Consequently, the empirical evaluation plays a more significant role in the design of parallel programming than it does in standard single-threaded one. The use of threads enables to improve performance substantially, however, managing the simultaneous processes and their possible interactions is far from a simplicity. Major challenges which are required going into, are the following:

- **Communication** - relates to the bandwidth and latency issues associated with exchanging data between threads.

- **Synchronisation** - is the process by which two or more threads coordinate their activities.
– *Load balancing* - is the distribution of work across multiple threads so, that they all perform roughly the same amount of work.

– *Scalability* - is the case of making efficient use of a larger number of threads when software runs on more capable systems; it is important to create algorithms that will scale far beyond their current systems.

3 Computer Architectures and Programming Models

One of the earliest classification systems for computer architectures and program models was proposed by Michael J. Flynn in 1966 [2], known as Flynn’s taxonomy. It is based upon the number of concurrent streams of information available in the computer system. Two types of information flow into the processor: instructions (control) and data. The instruction stream is defined as a sequence of instructions performed by the processing unit, and the data stream is defined as the data traffic between the memory and the processing unit. According to Flynn’s taxonomy, either of the instruction or data streams can be single or multiple:

- **Single Instruction, Single Data streams (SISD)** - a conventional single-processor computer which exploits no parallelism in either the instruction or data streams.

- **Single Instruction, Multiple Data streams (SIMD)** - a computer which exploits multiple data streams against a single instruction stream to perform operations which may be naturally parallelised (e.g. an array processor or GPU).

- **Multiple Instruction, Single Data streams (MISD)** - multiple instructions operate on a single data stream. This is an uncommon architecture, generally used for the fault tolerance, where heterogeneous systems operate on the same data stream and must agree on the result.

- **Multiple Instruction, Multiple Data streams (MIMD)** - multiple autonomous processors simultaneously execute different instructions on different data. Distributed systems are generally recognised to be MIMD architectures, either exploiting a global shared memory or a distributed memory space.

Although many machines are hybrids of these categories, this classic model has survived since it is simple, easy to understand, and gives a good first approximation. It represents the most widely used scheme.

Today, almost all modern computer systems are based on a MIMD architecture, which can further be subdivided into the following two categories:

- **Single Program, Multiple Data (SPMD)** - multiple autonomous processors simultaneously execute the same program on different data, but at independent points, rather than in the lock-step fashion that SIMD imposes. SPMD is considered to be the most common style of parallel programming.

- **Multiple Program, Multiple Data (MPMD)** - multiple autonomous processors simultaneously execute at least two independent programs. One example of this approach is the *master/worker* strategy, where one node is defined to be a manager (*master*) which runs one program farming out data to other nodes (workers) that all perform simultaneously a second program.
According to the characteristics of the physical memory included, the MIMD class can be split into systems with the shared memory, distributed memory, or combination of both distributed shared memory. A system in which each element of main memory is accessible with equal latency and bandwidth is called Uniform Memory Access (UMA) architecture, unlike a system that does not have this property and is called Non-Uniform Memory Access (NUMA) architecture. In general, UMA can be achieved only by SM systems which include a memory not physically distributed. Because of the own local memory, DM systems obviously belong to the NUMA type.

3.1 Shared Memory Systems

In a shared memory (SM) system all processors have access to a global shared memory representing a single address space. Typically, a shared memory may consist of several modules organised in a form of hierarchy and are actively supported by a hardware (bus or crossbar switch). Different processors can interfere with each other when accessing the same memory module, and these race conditions may manifest in the behaviour of the program. Therefore, synchronisation mechanisms, such as barriers, locks and semaphores, have to be used to avoid inconsistencies in memory and among caches. An advantage of this model is that the notion of “data ownership” is lacking, and hence there is no need to specify explicitly the communication of data. However, understanding and managing locality becomes more difficult. The dominant programming model for the implementation of multi-threading concept in SM architectures is the OpenMP [3] interface.

OpenMP (Open Multi-Processing) - an Application Programming Interface that supports multi-platform shared memory parallel programming in C/C++ and Fortran on many architectures, including Unix and Windows NT platforms. The OpenMP specification (started in 1997) provides a portable, scalable, and explicit programming model, offering the programmer full control over the parallelisation. OpenMP consists of three main API components: compiler directives, library routines, and environment variables which influence the run-time behaviour. For parallel program execution the Fork-Join model is applied:

   Start - the program begins as a single process (master thread), which executes sequentially until the parallel region construct is encountered;

   Fork - at the point of the parallel region construct the master thread creates a team of additional threads, and the code enclosed by is then executed in parallel by all participating threads;

   Join - when the team threads complete statements inside a parallel region, they synchronise and terminate, leaving only the master thread to resume the execution.

OpenMP provides a relaxed-consistency and temporary view of the memory. Threads can cache their data, but they are not required to maintain exact consistency with the real memory at all times, it is the responsibility of the programmer.

3.2 Distributed Memory Systems

A distributed memory (DM) system, also referred to as message passing computer cluster, comprises of a number of independent computers (nodes), which are linked together by an interconnection network enabling nodes to interact with each other. In DM systems there is no global memory, each processing node has its own address
space, thence the computational tasks can only operate on local data. If remote
data is required, it is necessary to communicate with one or more processors and
the data transmission takes place using the message passing via the interconnection
network. In programming DM systems the key issue is how to distribute data over
the memories to avoid frequent inter-node communications. For programming DM
systems the MPI [4] has become a de facto standard. There exist more than a
dozen implementations on a great variety of computing platforms. A grid-enabled
implementation of MPI, MPICH-G2 [5], allows users to run MPI programs across
multiple computers at different sites of a grid.

**MPI (Message Passing Interface)** - a specification for an API (started in 1993)
that allows many computers to communicate with one another. The interface is
suitable for use by fully general MIMD programs, as well as those written in the more
restricted style of SPMD. The main functionalities provided by standards MPI-1
and MPI-2 are: point-to-point communication, collective operations, communication
contexts, process groups and topologies, environmental management and inquiries,
profiling interface, bindings for Fortran and C. In most MPI implementations, a
fixed set of processes is created at the program initialisation, where one process
is created per CPU (or CPU core), each process may execute a different program.
Probably MPI’s most important feature is a mechanism called a “communicator”,
which allows to define groups encapsulating internal communication structures.

### 3.3 Distributed Shared Memory Systems

Distributed shared memory (DSM) and memory virtualisation refer to a wide class
of software and hardware implementations in which the processor has access to its
own local memory and also to a large shared memory. Software DSM systems and
virtualisation technology follows from memory management architectures and vir-
tual memory techniques, which provide a transparent network interface based on
the memory abstraction - a global address space for physically distributed memory
machines. A memory virtualisation decouples memory resources from individual sys-
tems in data centres and then aggregates those resources into a virtualised memory
pool available to any computer in the cluster or networked server.

### 3.4 Graphics Processing Units

In recent years graphics processing units (GPUs) have attracted a lot of interest in
several fields struggling with massively large computational tasks. They are leaving
behind their fixed function as accelerators and growing rapidly into general pur-
pose computational devices for highly parallel tasks. GPUs are useful mainly for
extremely data-intensive applications, where similar calculations are performed on
vast quantities of data that are arrayed in a regular grid-like fashion.

In 2007, NVIDIA launched the parallel software platform **CUDA (Compute
Unified Device Architecture)** [6, 7], which is the most mature work of the emerging
programming models for GPUs. CUDA focuses on the GPU as a threaded co-
processor to the host CPU, and specifies the associated memory model as a variety
of different address spaces for communication within the GPU and with the host
CPU. CUDA includes C/C++ software development tools (special extensions and
API calls), and function libraries.
In the last year the language **OpenCL (Open Computing Language)** [8] deserves a lot of popularity. It is claimed to be: “**OpenCL is the first open, royalty-free standard for general-purpose parallel programming of heterogeneous systems. It provides a uniform programming environment to write efficient, portable code for high-performance compute servers, desktop computer systems and handheld devices using a diverse mix of multicore CPUs, GPUs, Cell-type architectures and other parallel processors such as DSPs**”. But following the opinion of some experts, for example, of Michael Wolfe (Portland Group) [9], OpenCL will not replace or supersede other parallel languages as MPI, OpenMP, Java threads, or Map/Reduce. It’s likely to be more useful as a target language for higher level programming languages, tools and environments, or to implement optimised libraries, than as a language for a more general programming community.

## 4 Hybrid Programming Models

Current computing systems allow applying of many parallel solutions at the same time to achieve the maximum combined effect. For the last several years, most clusters have been built of a collection of multicore nodes, with a shared memory at the node level, and distributed memory between nodes. In order to exploit all the computing and memory resources effectively, a hierarchical hybrid programming model of parallelisation combining diverse task-parallel, data-parallel, service-oriented, and other concepts, appears to fit well. For instance, grid and workflow services can be used to implement a coarse grain parallelism, the MPI for communicating between nodes, and some other programming paradigms (e.g. OpenMP, Pthreads, PGAS languages, CUDA, OpenCL, etc.) within the node and multiple address spaces. Additionally, to get a top speed-up within each CPU, SIMD vector instructions, super-scalar instructions, and the multiple parallel functional units should operate.

At present, the parallelism at lower levels is usually detected and generated automatically by the compiler and handled transparently by the CPU itself. Considering a higher-level parallelism, the situation is much more complicated. There is a lack of common software tools which would span across multiple levels and different hierarchies of the architecture. Despite advances in OS, compilers and programming models, the developer still has to manage some of that hierarchy by himself. Currently, the most widely used parallel programming models are implemented in three major ways: as libraries invoked from traditional sequential languages, as language extensions, or complete new execution models.

A **library-based** solution is, in general, easy to port and can be independent of processor or compiler vendor. The MPI library is one well-known example.

A **language-based** solution exposes the semantics in the language, allowing compilers or other tools to analyse and optimise the program. For example, CAF and UPC expose MPI-like parallelism and communication in the language.

A **directive-based** solution has some of advantages of the language-based approach in that directives expose the semantics to the compiler and other tools allowing intelligent analysis and optimisation. The OpenMP is a widely available parallel programming model based on directives.

The **Berkeley View** project [10] stated that the model must allow the programmer
to balance the competing goals of the productivity and implementation efficiency. Based on this the concept of two programming layers was introduced:

- **Efficiency layer** - includes expert programmers who produce software components, libraries and services, implementing good parallel algorithms; and software frameworks that support general structural patterns of computation and communication.

- **Productivity layer** - includes domain experts and naive programmers who productively build parallel applications and application frameworks using the existing libraries and tools.

To design and evaluate parallel programming models the Berkeley View has proposed a set of **dwarfs**, instead of traditional benchmarks, where a dwarf is defined as an algorithmic method that captures a pattern of computation and communication. A dwarf is specified at a high level of abstraction that can group related but quite different computational methods and allow reasoning about their behaviour across a broad range of applications.

In the process of developing future parallel applications the major benefits in terms of programmer productivity and software portability will be gained by an appropriate intelligent software development platform that manages and optimises code to target the various parallelism mechanisms available. It is expected that sophisticated software agents will assist to automate the process of application design at several levels. Some agents will advice the composition of a program at a higher level of abstraction using miscellaneous parallel building blocks of different granularity, while others will guide the exploration of different parallel strategies and performance optimisations. The programmer may specify main strategies for code and data partitioning and the system can explore this restricted optimisation space to generate an efficient parallel program. Such **semi-automatic parallelisation** seems to be a reasonable methodology. In this way the best practices from HPC and HTC combined with modern technologies can offer new opportunities to obtain an effortless solution.

## 5 Conclusion

Many software-design communities are trying to figure out the best way of making use of new architectures.

The MPI Forum [11] reconvened in 2007, to clarify some MPI-2 issues and explore developments for a possible MPI-3. Within the MPI-3.0 standardisation effort also the support for multicore and hybrid programming is included.

The Berkeley View [10] project has investigated the influence of multicore processors in applications, hardware, programming models, and systems software for parallel computing.

The project **Future Grid** (a part of Teragrid) is aimed at supporting the development of new system software and applications in order to accelerate the adoption of new technologies in scientific computing. This will allow to research multicore computing, cloud computing, and new algorithms and new software paradigms.

The current Europe’s leading grid computing project EGEE [12] is concentrated
first of all on providing a computing infrastructure and on running distributed applications. The middleware support for parallel applications is rather poor. The new grid project EGI [13] will foster the convergence of components from different middleware providers to a single, ideally standardised interface UMD (Unified Middleware Distribution), to meet the evolving needs of application communities.

Acknowledgements. This work is supported by projects: VEGA No. 2/0211/09, VEGA No. 1/4103/07, EGEE-III FP7-222667, and DMM VMSP-P-0048-09.

References