Proceedings of *RAGtime 10–13, 15–17/20–22/15–17/14–16 September, 2008/2009/2010/2011, Opava, Czech Republic*
Z. Stuchlík, G. Török and T. Pecháček, editors, Silesian University in Opava, 2014, pp. 143–154

Programming models used on Many-Core architectures

Jan Novotný

Institute of Physics, Faculty of Philosophy & Science, Silesian University in Opava,
Bezručovo nám. 13, CZ-746 01 Opava, Czech Republic

ABSTRACT
The time in which we live is characterized by an ever-increasing amount of data that we are able to explore and acquire. In all fields of science we could find some examples. Processing large volumes of information thus brings the requirement for engaging computational science. With increasing demands on data processing is advantageous to use new technology and start using parallel computation. Effective use of current technology requires from programmers new knowledge and skills. They meet with the countless new programming models and tools. In this article, we summarize the most commonly used programming models and points which good programming model should meet. The article also try to highlight the reasons why one should use a structured parallel programming.

Keywords: patterns – parallel computing – many-core systems – heterogeneous systems – programming models – parallel software development

1 INTRODUCTION

There is constantly increasing number of sold devices (smartphones, tablets, etc.) which enable parallelized applications. Not only availability of these systems is on the rise, also computational demands of applications are increasing that leads to the development and usage of certain methods. We are at a time when multi-core or many-core architectures are becoming mainstream.

In physics, the need to begin using the accelerated calculations appears in several areas. For example, the arrival of new types of radio telescopes such as the SKA (Square Kilometre Array) requires use of High Performance Computing (HPC). HPC is also often needed in areas of research such as cosmology (mostly real-time processing area) and galaxy evolution, pulsars, star and planet formation, etc.

Computational science where a massive amounts of calculations are processed is no longer a domain of supercomputers or distributed platforms. For scientific computations is now possible to use besides CPU specified coprocessors (e.g. Xeon Phi) and the graphic cards. Each of these devices has its own advantages and disadvantages. The choice either to use the CPU or many-core coprocessors is not easy and is often dependent on a computing problem.

It turns out that to get better performance, it is necessary to abandon the serial programming and start from the beginning using the parallel program development approach. Generally, it appears as the best to use a so-called heterogeneous approach. That is a division of the algorithm into several parts and calculations, which are distributed among different computational units (hosts and devices).

Nevertheless, the programmer should keep in mind that his main goal is to write scalable code, i.e. code should be able to use any amount of available parallel hardware.

Software engineers designing parallel programs have several options how to proceed. Their choice is often dependent on their skills, experience and intended objective of the application itself.

One of the easiest way is to rewrite few lines of the existing serial program or add specific lines (e.g. Directives) and use compilers ability of auto-parallelization. The other way is to use structured patterns of parallelism. The use of these patterns has it's advantages, such as ease of readability, scalability, extensibility and so on. Most patterns avoid non-deterministic behaviour as well as the serial programming, which is deterministic by nature.

Using current programming tools may nevertheless lead to worse results because of unnecessary serialization. Among programmers, this is known as a serial trap. Recognition and avoidance is perhaps the hardest part of parallel programming.

For example imagine you have two workers and four tasks. One worker take the first two tasks and the second worker the rest. The two workers can run in parallel as long as there is no dependency between the four tasks. But if there is some, like the third task need results from the first task, the system need to run serially. Moreover sometimes you can find that each task can take different time to process. If you cannot divide further then the total running time of the program is the sum of the dependent tasks or the task that takes longest. This is the *span* of example above.

We are in an era in which to get any enhancements in application performance we must use *parallel thinking*. Together with the above mentioned patterns finding serial traps is deemed as a first step to think parallel.

2 MOTIVATION

Hardware is parallel by nature because of several techniques like instruction level parallelism (ILP), pipelining, vector instructions, hyperthreading, etc. but CPU architects and designers made it so that CPU seemed serial on the outside. CPUs started to use implicit parallel operations long ago without programmers explicitly telling them to do so. That is so-called *serial illusion*. More about this problem could be found in literature.(McCool et al., 2008) Programmers depended on this illusion for a long time and now this approach does not lead to any significant improvement as is shown in the right bottom of the Fig. 1. Moreover the performance of serial program will not grow over time as show dark blue and green color in the figure.

To achieve improvement programmer can either rewrite the sequential code with special directives or use the compiler's automatic parallelization tools. However it is not universally working and mostly it comes with no significant improvements as is shown in the Fig. 1. The right bottom part of the figure indicates that the individual benchmark tests are rather

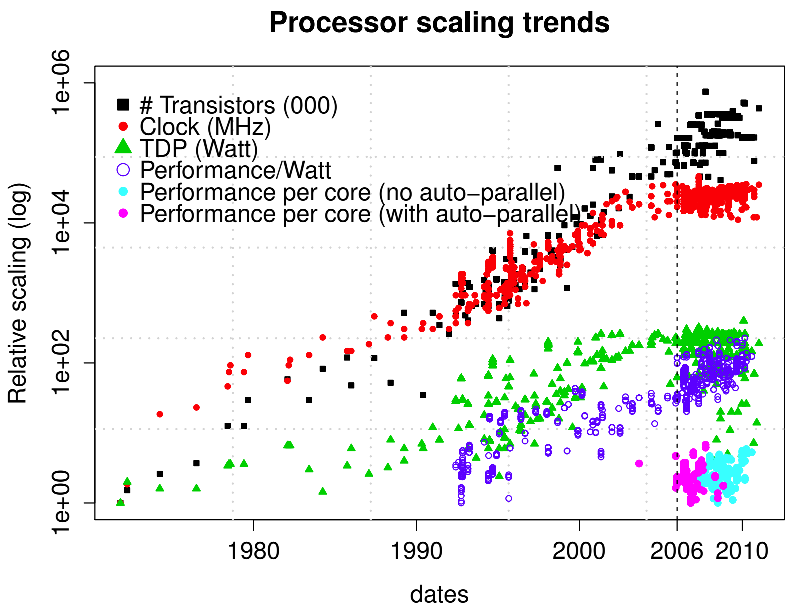

Figure 1. The figure is showing the trends over the years with emphasizing of the year 2006 when the multi-core era comes. The number of transistors is raising with accordance to the Moore law on the other hand clock rates nowadays are stalling. Data comes from CPU DB.(Stanford VLSI Group, 2014)

flat when the auto-parallelization is allowed.[1] The gain in performance per core using auto-parallelization over the years is not particularly a trend and therefore exploitation of the parallel nature of the hardware generally leads to the need of using the explicit parallel programming.(Herb Sutter, 2005) We are not saying to not use the benefits coming of the rather easy way to run application quicker. We want only stress that auto-parallelization is not always the best choice and in time scale not most effective.

Data in the Fig. 1 come from CPU DB (Danowitz et al., 2012) which gather information about CPU performance since 1973. In the figure could be observed several trends. Firstly we can see that since 1975 until 2006 the clock rate is exponentially increasing. In past the increase of clock rate was enough to improve performance of CPU until it reached so-called *power wall*, *memory wall*[2] and *ILP wall*[3]. In the figure since year 2006 can be spotted continual stagnation of clock rate, thermo-design power and performance per Watt. Still

[1] The CPU2006 Spec is an industry standard benchmark providing data on CPU performance since beginning of the multicore era around year 2006. To fully understand the nature of these benchmarks we address to read (Stanford Performance Evaluation Corporation, 2014; Subcommittee, 2006).

[2] The growth of off-chip memory is not as fast as the on-chip memory. The programmer nowadays cannot ignore the overall data rate (bandwidth) and the time between submitted and satisfied request (latency).

[3] The hardware is parallel in nature and for example if two close instructions do not depend on each other, they can be ran parallelly. However the useful limit for most real problems is around six instructions.

according to Moore law the amount of transistors in CPU is growing exponentially and that is why the performance per Watt is slightly going up as demonstrate dark blue color in the Fig. 1.

Secondly, we can see that around year 2004 the increase in thermal design power (green dots) gets to its maximum that can be effectively air-cooled (power wall). That was the most severe issue which appeared and led to creation of multi-core architectures.

Nevertheless, the theoretical performance continues to grow. It is clear that a programmer to obtain better results need to move from just rewriting the serial algorithm (*refactoring*[4]) and using the auto-parallelization. Nowadays to improve the code and be able to scale it in time developers must explicitly specify parallel algorithms.

In many cases the way to *think parallel* and use new programming models seems the best solution. However with that more problems arises: knowledge of new models, finding the most time consuming chain of tasks which needs to be run serially (*span*[5]), etc.

Finally learning parallel programming side by side with serial programming is not the best approach. Teaching how to think parallel needs to be done from the beginning and alone to avoid certain serial assumptions. Structured approach to parallel programming is essential and it is one of the best strategies for writing effective scalable program. Using a set of patterns with standard names help to design such programs and also aid in readability.

3 PARALLEL SOFTWARE DEVELOPMENT

The process of software development today provides various methodologies that have been proven over time. Their structure often helps to set a few questions that are advised to answer before starting programming, or before refactoring. (Figure 2 shows the three most commonly used methods).

These questions may specify for example, exactly what do we want from the application? On which architectures will it run or which hardware is available (this decision is mostly set during the design phase, see 2)? Where are the input data and what type of data have we available? These are the types of questions that will assist us in the software development process and possibly help us find the span of the program.

During the implementation phase, it is also preferable to use development environments and tools such as profilers, debuggers, etc. This fact is highlighted in documentation from nVidia (best practice guide).(nVidia, 2014)

In its essence nVidia best practice guide introduces four elements: assess, parallelize, optimize and deploy (APOD cycle) which is very similar to the spiral methodology. There should be emphasized that this is a cyclical process and it is always necessary to verify the results with a control test in the verification phase.

Finally, McCool et al. (2008) highlight problems that must be discussed in process of parallel algorithm design:

[4] It is true that this is sometimes sufficient, and certainly one of the simplest ways to parallelize code.(Fowler et al., 1999)

[5] Span refers to the longest set of tasks that must run serially. This chain in computer science is known as a critical path.

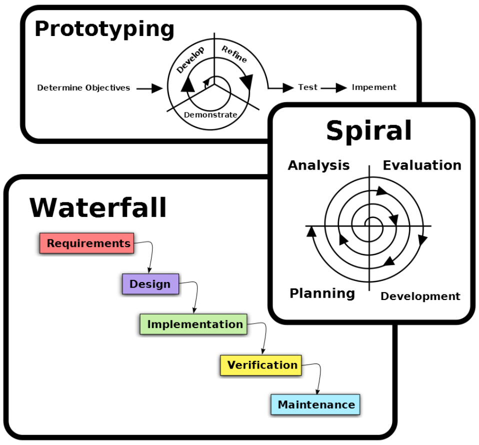

Figure 2. The three basic approaches applied to software development methodology frameworks.(Wikimedia Foundation, 2014)

- "Total amount of computational work.
- Critical path (span).
- Total amount of communication."

In question of amount of computational work we believe in heterogeneous approach, i.e. systems with different kind of units. Every programmable unit has its own specifics and it is favourable to use certain kind depending on its abilities.

Critical path refers to recognition and finding chains of instruction which needs to be run serially as mentioned above in Section 2.

The last mentioned concerns the fact that memory access and communication between cores costs time. How much time it costs depends on the location of the work unit (locality). Relatively low cost are threads which run on the same core, more cost those which share an on-chip memory and even more cost those in another socket. It may indicate that the problem is memory bandwidth limited and not computational limited.[6] Finally we can say that finding the bottleneck of the program is the most critical problem.

4 PROGRAMMING MODELS AND PATTERNS

Effective management of communication and redistribution of work is required for meaningful parallel programming. Usage of patterns should facilitate both. Programming models that enable effective implementation of patterns are also necessary.

Unfortunately none of the most widely used programming languages are adapted for needs of parallel programming. However, if we look at the amount of serial code already written, it would be a shame not to reuse it. For this reason, most parallel models are in fact an extension of the current programming practices and tools.

[6] In the case of heterogeneous systems must be also taken into account the bandwidth of the PCI Express bus.

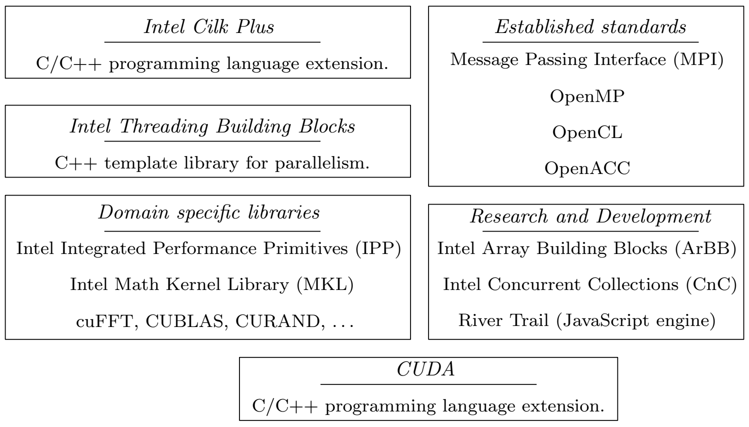
Intel Cilk Plus
C/C++ programming language extension.
Intel Threading Building Blocks
C++ template library for parallelism.
Domain specific libraries
Intel Integrated Performance Primitives (IPP)
Intel Math Kernel Library (MKL)
cuFFT, CUBLAS, CURAND, ...
Established standards
Message Passing Interface (MPI)
OpenMP
OpenCL
OpenACC
Research and Development
Intel Array Building Blocks (ArBB)
Intel Concurrent Collections (CnC)
River Trail (JavaScript engine)
CUDA
C/C++ programming language extension.

Figure 3. Examples of most frequently used programming models and libraries for parallel programming.(McCool et al., 2008)

In Figure 3 are examples of most frequently used extensions and libraries in industry and science. Despite the fact that each of them has its advantages and disadvantages, it is important to know that the models have the following properties:

- performance,
- productivity,
- portability.

For portability there is need to extend functionality and performance across operating systems and compilers. Programming languages like Java, C and C++ are portable and most of the programming models in the Fig. 3 too. Just CUDA language does not fulfil this criteria. To be able to compile the CUDA code you need to use nVidia compiler and for running the program a CUDA enabled graphics card is required.

In general, using abstractions like elemental functions or array operations is preferable. This approach is better in general than a program specifically suited to concrete hardware although such a program is very efficient. For example each Intel's processor today can support different vector instruction set extension, that is why using abstraction to specify vectorization[7] instead of vector intrinsic[8] is better. However, we must keep in mind that in some cases programming for the specific pieces of hardware is desired.

In terms of productivity models we should make it possible to debug and maintain programs as well as easily implement a range of suitable algorithms and maintain composability. By that we mean the ability to use a feature regardless to other features used in the linked library or elsewhere in the code. Consider a case where this is not true and using `for` statement somewhere in the code meant that you cannot use `if` statement anywhere. It's something we do not want, but unfortunately can happen.

[7] Vectorization is a concrete form of parallelism enabling simultaneous computing using vector hardware by instructions such as MMX, SSE, and AVX.

[8] Intrinsics seems like a function in programming language but are supported directly by the compiler.

Oversubscription is another quite common problem, i.e. each use of parallelism could define a new set of threads and so exceed the number of threads that system can handle.

To obtain reasonable improvement of performance we must sustain scalability. In other words, generating more parallelism when the problem grows larger is crucial. This can be achieved by so called data parallelism and each programming model in the figure is supporting it. For example Cilk Plus has a “special” array notation extensions for C and C++ (see, algorithm 1) which explicitly specify data parallelism operations. The ArBB has even simpler solution as long as the data are stored in the appropriate containers. Such abstractions built in the models make possible to use regular data parallelism[9] explicitly and not rely only on the compilers.

Algorithm 1: Examples of vector addition in different programming models.

Serial vector addition in C:

```
for $i$ ← 0 to 100000 do
    a[i] = b[i] + c[i];
end
```

Parallel vector addition in Cilk Plus:

```
a[0:100000] = b[0:100000] + c[0:100000];
```

Parallel vector addition in ArBB:

```
input  : a
output: b, c
a = b + c;
```

Above mentioned models and usage of elementary functions or compiler directives such as `pragma`, are helping to vectorize the code and work in the context of regular data parallelism. At last, sometimes to support vectorization, change of the data layout (array of structures or structure of arrays) is preferable.

Selecting the programming model is not an easy task, despite the fact that some models overlap its functionality, but not portability. Next factor may be the hardware on which will the application run. Other may be the requirement for fine control of architecture or vice versa ease of implementation (see Fig. 4).

Structured programming using patterns is now commonly used and proven method of development of applications in various fields such as natural language learning (Kamiya, 2012), software architectures (Buschmann et al., 1996; Schmidt et al., 2000) and so on. Perhaps that is why they have became best practice tool among the software engineers.(Gamma et al., 1994)

In computer science there are three different strategies: Design Patterns, Implementation Patterns and Algorithmic Strategy Patterns. The first two mentioned have rather abstract character. The first of them can be classified as a high level and the second as a low level

[9] That is a subcategory of data parallelism which is mapped onto vector instruction of the hardware.

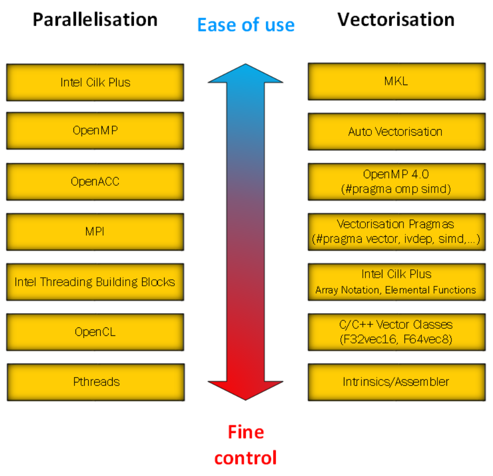
Parallelisation
Ease of use
Vectorisation
Intel Cilk Plus
OpenMP
OpenACC
MPI
Intel Threading Building Blocks
OpenCL
Pthreads
MKL
Auto Vectorisation
OpenMP 4.0 (#pragma omp simd)
Vectorisation Pragmas (#pragma vector, ivdep, simd,...)
Intel Cilk Plus Array Notation, Elemental Functions
C/C++ Vector Classes (F32vec16, F64vec8)
Intrinsics/Assembler
Fine control

Figure 4. Here is shown the relevance between easy of implementation and fine control from different models provided by Leibnitz Supercomputing Centre in their guide for users.(Leibnitz Supercomputing Centre, Bavarian Academy of Sciences and Humanities, 2014-07-21)

(tied to specific hardware). Algorithmic Strategy pattern lies somewhere between these two, and thanks to it's semantic behaviour and ease of implementation appears to be best choice for most of the cases. Because they affect how your algorithms are organized, we strongly believe that algorithmic strategy patterns are good for learning and teaching parallel thinking. These design in literature are usually so-called algorithmic skeletons.(Cole, 1989; Aldinucci and Danelutto, 2007)

Lots of Algorithmic Strategic Patterns nowadays are already implemented in the programming models and often knowledge of these patterns helps to teach programming languages. Still, knowledge and use of algorithmic skeletons exceeds certain programming languages and models in being more general.

Examples of the three most useful and frequently used patterns are: nesting, mapping and fork-join. Nesting is often used in sequential programming and is important for a modular approach. Yet its transfer to the parallel programme is a challenge. The key to implement map pattern is a division of the problem into smaller tasks and run those parallelly. This is called embarrassing parallelism. Problems where this pattern can be used are well scalable and lead to efficient vectorization. Last pattern: fork-join divide tasks recursively into simpler ones and combines them later. In fact, it is one of the pillars of the strategy divide and conquer in computational science.

All these mentioned patterns highlight that to get good scalable code we need to pay attention to the data parallelism method. In other words, dividing problems into smaller ones with possibility to grow with increasing overall problem size.

Generally, the programmer should use a structured approach for better readability, debugability and scalability of the code. Patterns should be used as basic building blocks. They also become a common vocabulary when discussing ways of how to solve computational problems. In addition, patterns are independent of the usage of a particular programming model, programming language or even computing architecture.

5 SHORT OVERVIEW OF PARALLEL PROGRAMMING MODELS

The following paragraphs briefly describe selected programming models satisfying above mentioned properties.

Cilk Plus

Cilk Plus is an extension of the programming languages C and C++ it supports task and data parallelism. It is very easy to use and compared to sequential code it differs only in adding keywords (i.e. `cilk_sync`) and array section notation. This characteristic is mainly due to the fact that Cilk Plus is embedded into most compilers. In fact if you run Cilk Plus program with one thread it will act as the special keywords are ignored (serial illusion). Other features are:

- "Fork-join to support irregular programming patterns and nesting." (McCool et al., 2008)
- Parallel loops to support patterns such as map.
- Explicit vectorization by using array section like `pragma simd`.
- "Load balancing via work-stealing." (McCool et al., 2008)

For more information and specification see Intel (2014a).

Threading Building Blocks (TBB)

This is a C++ library under development by Intel. Because it is not a language extension it is supported by all standard (ISO C++) compilers. In order to run the blocks of code in parallel TBB requires the use of functors.

Just as Cilk Plus TBB supports parallelism based on the tasking model. In other words, the individual operations are considered as tasks and are dynamically allocated to each core using the library run-time engine and automating efficient use of the CPU cache.

TBB provides following features:

- "Template library supporting regular and irregular parallelism." (McCool et al., 2008)
- Support for certain pattern (fork-join, scan, reduction).
- Load balancing via work-stealing.

Advantage of TBB is in algorithms that are written with respect to the minimum assumption of data structure. The literature describes TBB as part of generic programming and C++ standard template library (STL) is a good example of its philosophy.

Commonly seen in practice are the use of the individual components of TBB with other programming models such as OpenMP and Cilk Plus. For more information and specification see Intel (2014b).

Open Multi-Processing (OpenMP)

OpenMP is a standard developed by a consortium of major brands in hardware and software. This is an application programming interface (API) that supports shared memory multiprocessing programming. OpenMP is based on using a set of compilation directives or pragmas in Fortran, C and C++.

In its essence OpenMP is based on multi-threading model, i.e. the main thread divide and run multiple threads sub-dividing the task among them. This property is especially beneficial for certain types of algorithms and memory hierarchy platforms often seen in high-performance computing (HPC). The main problem of the explicit threading model is mentioned oversubscription.

One of the advantages over Cilk Plus and TBB is the ability to explicitly manage threads using the thread ID and the number of threads to control how the work is mapped to threads. This feature is often used by HPC programmers. On the other hand, it is a limiting factor too. It prevents system to determine the load balancing.

In summary OpenMP interface provides following features:

- "A tasking model that supports execution by an explicit group of threads.
- Creations of group of threads that jointly execute a block of code.
- Support for atomic operations and locks."(McCool et al., 2008)

For more information and specification see OpenMP Architecture Review Board (2014).

Array Building Blocks (ArBB)

ArBB is compiler independent C++ library supporting data parallelism on different architectures such as multi-core processors, graphics processing unit (GPU), Intel Many Integrated Core Architecture (MIC). The parallelization is mediated by a set of operations which operate on a group of data. It provides following features:

- "High level programming language with elemental functions and vector operations.
- Offloading to attached many-core architectures without changing source code." (McCool et al., 2008)
- Safe by default: preventing parallel programming bugs such as deadlocks and data races.

This programming model is classified as a high-level programming language. It was developed by Intel as experimental library. During October 2012 was announced discontinuation of development in favour of Cilk Plus and TBB.(Intel, 2011)

Open Computing Language (OpenCL)

OpenCL is an open standard maintained by the Khronos group and is used for writing programs that can execute on heterogeneous machines such as CPUs, GPUs, field-programmable gate arrays (FPGAs) and others.

OpenCL framework divides computing system into two parts: CPU (host) and accelerators (devices). Therefore, it includes a kernel language[10] and an API for data management and execution of the kernels on the devices from the host.

As a low-level language is primarily designed for performance programming. This fact requires more effort from programmers to specify computations into detail and often write different versions of kernel for each class of devices.

Although OpenCL is not considered as mainstream among programmers as the previous models, we believe that it has it's place among others and should not be ignored. For more information and specification see OpenMP Architecture Review Board (2014).

[10] The language is a standard C99 including certain features.

Compute Unified Device Architecture (CUDA)

CUDA is a parallel computing platform using primarily GPUs. It is developed by NVIDIA Corporation and its implemented on graphics cards of their brand.

For software engineers CUDA is programming model accessible through a set of accelerated libraries, compilation directives (OpenACC) and as an extension of C/C++ and Fortran languages with using specific compilers.

Despite this programming model does not comply the requirement for portability and works only on certain devices, is it highly extended and used in computational science. For example third party wrappers are available for Python, Java, Perl, MATLAB, and also software Mathematica have its native support.

In addition to OpenCL, CUDA is for beginning programmers a little easier and has very good documentation on the Web (nVidia, 2014-08-01). This and the above mentioned are arguments why it should be one of the options in deciding which programming model to use.

CONCLUSION

Mainstream computers and other electronic devices around us are changing in its essence and to be able to get better performance and scalability of old or new software we need to switch to a new concept of thinking. Furthermore, as we have shown in Section 2 programmers cannot rely on so-called serial illusion any-more. We reached the three walls: power wall, memory wall and ILP wall. Because of that we are driven by the need to change to the explicit parallel programming.

Today in computer science and in science in general are increasingly used heterogeneous systems, i.e. systems based on multi-core and many-core computational units. To make better use of these machines, it is good to use parallel programming models. Some examples of programming models and their properties we described in Section 5.

Furthermore, to achieve effective programming, it is the best practice to use parallel patterns. They are easily scalable, debugable and among computational scientists settle basic vocabulary. That is why we want to stress that usage of parallel patterns and models is for a computational scientist inevitable.

ACKNOWLEDGEMENTS

I would like to express my special thanks to Karel Adámek and Anna Janíková for their valuable and constructive suggestions during writing this paper and for many consultations. The presented work was supported by EU grant Synergy CZ.1.07./2.3.00/20.0071

REFERENCES

Aldinucci, M. and Danelutto, M. (2007), Skeleton-based parallel programming: Functional and parallel semantics in a single shot, *Computer Languages, Systems & Structures*, **33**(3-4), pp. 179–192, ISSN 14778424.

Buschmann, F., Meunier, R., Rohnert, H., Sommerlad, P. and Stal, M. (1996), *Pattern-Oriented Software Architecture Volume 1: A System of Patterns*, Wiley, Chichester ; New York, ISBN 9780471958697.

Cole, M. I. (1989), *Algorithmic Skeletons: Structural Management of Parallel Computation*, The MIT Press, London : Cambridge, Mass, ISBN 9780262530866.

Danowitz, A., Kelley, K., Mao, J., Stevenson, J. P. and Horowitz, M. (2012), CPU DB: recording microprocessor history, *Communications of the ACM*, **55**(4), pp. 55–63, URL `http://dl.acm.org/citation.cfm?id=2133822`.

Fowler, M., Beck, K., Brant, J., Opdyke, W. and Roberts, D. (1999), *Refactoring: Improving the Design of Existing Code*, Addison-Wesley Professional, Reading, MA, 1st edition, ISBN 9780201485677.

Gamma, E., Helm, R., Johnson, R. and Vlissides, J. (1994), *Design Patterns: Elements of Reusable Object-Oriented Software*, Addison-Wesley Professional, Reading, Mass, 1st edition, ISBN 9780201633610.

Herb Sutter (2005), The free lunch is over: A fundamental turn toward concurrency in software, *Dr. Dobbs Journal*, **30**(3), URL `http://www.gotw.ca/publications/concurrency-ddj.htm`.

Intel (2011), Intel® Array Building Blocks, URL `https://software.intel.com/en-us/articles/intel-array-building-blocks`.

Intel (2014a), Intel® Cilk™ Plus, URL `https://software.intel.com/en-us/node/522579`.

Intel (2014b), Intel® Threading Building Blocks (Intel® TBB) User Guide, URL `https://software.intel.com/en-us/node/506045`.

Kamiya, T. (2012), *Japanese Sentence Patterns for Effective Communication: A Self-Study Course and Reference*, Kodansha USA, New York, ISBN 9781568364209.

Leibnitz Supercomputing Centre, Bavarian Academy of Sciences and Humanities (2014-07-21), LRZ: SuperMIC - intel xeon phi cluster, URL `http://www.lrz.de/services/compute/supermuc/supermic/`.

McCool, M., Reinders, J. and Robison, A. (2008), *Structured Parallel Programming: Patterns for Efficient Computation*, Morgan Kaufmann, Amsterdam, 1st edition, ISBN 9780124159938.

nVidia (2014), CUDA C programming guide, URL http://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html.

nVidia (2014-08-01), CUDA toolkit documentation, URL `http://docs.nvidia.com/cuda/index.html`.

OpenMP Architecture Review Board (2014), OpenMP.org, URL `http://openmp.org/wp/`.

Schmidt, D., Stal, M., Rohnert, H. and Buschmann, F. (2000), *Pattern-Oriented Software Architecture Volume 2: Patterns for Concurrent and Networked Objects*, Wiley, Chichester England ; New York, ISBN 9780471606956.

Stanford Performance Evaluation Corporation (2014), SPEC - standard performance evaluation corporation, URL `http://www.spec.org/`.

Stanford VLSI Group (2014), CPU DB - looking at 40 years of processor improvements | a complete database of processors for researchers and hobbyists alike., cPUDBv1.2-11-30-gcbf16c8, URL `http://cpudb.stanford.edu/`.

Subcommittee, S. C. (2006), SPEC CPU2006 benchmark descriptions, *ACM SIGARCH Computer Architecture News*, **34**(4), pp. 1–17, URL `http://dl.acm.org/citation.cfm?id=1186737`.

Wikimedia Foundation (2014), Software development process, URL `http://en.wikipedia.org/w/index.php?title=Software_development_process&oldid=627519437`.