Parallel Computing
in the Area of Data Analysis

Habilitation Thesis

Ing. Petr Gajdoš, Ph.D.

Department of Computer Science
VŠB – Technical University of Ostrava
Czech Republic
Contents

List of figures vii
List of code samples vii
List of algorithms vii
Acknowledgement ix

1 Introduction of the research area 1
   1.1 Motivation ........................................ 1
   1.2 Research areas described in this work ........... 2
      1.2.1 Parallel approach at the level of algorithm design .... 2
      1.2.2 Scalability and higher occupancy ............... 3
      1.2.3 Parallel computing and visualization ........... 4

2 Parallel computing 5
   2.1 Parallel computer architectures ................. 5
      2.1.1 Flynn’s taxonomy ................................ 5
      2.1.2 Parallel architectures from the synchronization point of view ... 6
      2.1.3 Streaming multiprocessors ....................... 8
      2.1.4 Classification of parallelism in terms of software .......... 9
      2.1.5 API for parallel programming .................... 10
   2.2 Selected APIs ...................................... 11
   2.3 The assessment of parallel approach efficiency ........ 13
      2.3.1 Speedup factor ................................ 14
      2.3.2 Theoretical prerequisites of speedup ............. 16

3 Research directions in the context of parallelism on GPU 19
   3.1 Research direction I:
      Parallel approach at the level of algorithm design .......... 19
      3.1.1 Loops elimination and higher vectorization ........... 19
      3.1.2 Application domains and achieved results ............ 24
   3.2 Research direction II: Improving scalability and occupancy .... 26
3.2.1 Parallel implementation with regard to scalability:
  Fixed size of a task ................................................. 27
3.2.2 Parallel implementation with respect to scalability:
  Variable size of a task .............................................. 32
3.2.3 Application domains and achieved results ...................... 34
3.3 Research direction III:
  Parallel computing and visualization ............................. 39
  3.3.1 Interoperability between computing and visualization ...... 39
  3.3.2 Application domains and achieved results ................... 41

4 Summary 45
  4.1 Future work .................................................... 45

5 Author’s Publications 47

6 Included Papers 55

Bibliography 154
List of Figures

3.1 The practical comparison of simulation on GPU ........................... 25
3.2 Speedup of the final implementation of the selected method .......... 25
3.3 An illustrative result of a model of symbolic regression. .............. 28
3.4 Block/Warp computation of a single schedule in the flow-shop problem. 36
3.5 Symbolic regression and environment pollution analysis. ............ 36
3.6 The scheme of EEG signal analysis with the use of GPU. ............ 37
3.7 GPU based classification of patients ...................................... 38
3.8 Accelerated graph layouting .............................................. 42
3.9 Indexing based on Hilbert curve in MDDE .............................. 43

List of code samples

3.1 An example of calling method from the CUBLAS library ............. 23
3.2 A code snippet for OpenCL real-time compilation ..................... 31
3.3 CUDA kernel for evaluation of a single schedule ..................... 35

List of Algorithms

3.1 Updating the distances: iterative approach ............................. 20
3.2 Updating the distances: matrix approach ............................. 21
3.3 Updating the distances during training phase using matrix operations. 22
3.4 Pseudocode for symbolic regression ................................. 32
3.5 Updating coordinates of nodes using Fruchterman-Reingold algorithm 40
3.6 Visualization of the graph structure ................................. 41
Acknowledgement. I would like to thank my wife Jana for her love and support, as well as to my small children for their energy and pure view of the world. I would also like to thank Professor Václav Snášel for his leadership and help. Last but not least, I would like to express my special gratitude to all colleagues with whom I have had the honor to work.

January 2017, Petr Gajdoš
1.1 Motivation

One of the indisputable and continually growing trends in the field of data processing and analysis is the application of parallelism and the use of technologies that enable such an access. At the beginning, classic processors with support of multithreading were used and the accessibility of other computer architectures was not common at that time as it is nowadays. However, at the end of the first decade of a new millennium, particularly in 2006, it became obvious that the worldwide trend would lead to a massive parallelism indicating that even a personal computer would have multi-core processors or graphics cards with thousands of cores. Soon after finishing my doctoral studies I started to be wholeheartedly interested in new technologies and the fast developing use of graphics processors not only because of working with image data, but also due to the possibility to further implement algorithms from the field of data analysis. Our research team succeeded in obtaining one of the first graphics cards NVIDIA Tesla [130] and we had VŠB-Technical University of Ostrava registered as one of the institutions using the programme of NVIDIA Teaching Centre. This programme supported the preparation in subjects where a new computing platform NVIDIA CUDA started to be taught. Currently, this computer architecture plays one of the key roles in the field of parallel computing. Despite of this fact, it cannot be proved that this is the only one dominating computer architecture.

Over a few years there has been a massive growth of the need to analyse various data on computers, and alongside with this need, not only volumes of analysed data have increased, but also the pressure on the use of new technologies and higher computing power have been more significant. To solve this situation, new computing centres were built. For instance, the National Supercomputing Centre IT4Innovations within VŠB-Technical University of Ostrava was opened in 2013. From the standpoint of new technologies, computing centres did not rely only on the quantity of computing cores or the capacity of individual computing graphics cards, but they also started to use newly developing computing cards based on classic architecture such as Intel® Many Integrated Core Architecture (Intel MIC) [78, 128] and their cards Xeon Phi. This computer architecture played into the hands of a classic style of programming, because it supported standard and existing programming tools. More emphasis was given on the communication between processors and/or cards, so that various application programming interfaces (API) started to move to the forefront, such as Open Multi-Processing (OpenMP) [13, 14], Message Passing Interface (MPI) [110, 121] or a hybrid Open Accelerators (OpenACC) [29].
Current choice of technologies is so vast so that it is better to focus on their suitable usage with regard to the computing power, time, energy consumption, and suitability for a particular application area. This work is focused on the introduction of my research in the field of parallel data processing, particularly at several levels. The first level deals with the presentation of various views on the solution of the discussed issue with the use of parallel paradigm. A description of general issue of parallel computing is provided, as well as various aspects and approaches to the task solution with regard to the implementation of selected methods from the field of data analysis. Next level, for instance, deals with the use of parallel approach in selected application fields such as classification, regression or pattern matching. All parts are supported by selected publications. The final part of my work is devoted to the list of published papers.

1.2 Research areas described in this work

This work is focused on the presentation of achieved results when using parallel approach in the field of data analysis. The aim is to demonstrate various aspects of parallel paradigm in design and construction of algorithms and their further use in real data analyses. This paper summarizes various directions and views on the use of a selected technology in different parts of the research, and these individual directions are divided into separate parts. Each part is supported by the list of selected publications focused especially on the integration of a particular technology and the use of implemented methods. Regarding technologies, the research was predominantly focused on the use of NVIDIA graphics processors which, with respect to its architecture, can offer a higher degree of parallelization. From the standpoint of accessibility, these represent a cheap alternative for a practical use in real applications in comparison with clusters or grids of supercomputing centres.

1.2.1 Parallel approach at the level of algorithm design

The first presented part deals with a basic view on the use of algorithms to solve a particular problem in terms of parallel programming. The aim is to show that by modifying an approach during an algorithm implementation we can reach a higher degree of parallelization. Provided we consider the fact that if graphics processing units (GPU) [4, 5] are optimized to vectors and to matrix operations, we will obtain de facto the most optimal version of a selected method from the standpoint of graphics processor workload. The further use of patterns for data parallelism will only enable us to support the possibility of more extensive data processing above the frame of the capacity of one GPU. For instance, when connecting more cards via Scalable Link Interface (SLI) or easily in a cluster. The demonstrated approach is universal and it represents an accumulation of iterative calculations with the transfer to vector operations. The selected illustrative method of data analysis will be Self-Organizing Map (SOM) [88].
1.2. Research areas described in this work

Included papers are:


1.2.2 Scalability and higher occupancy

In contrast to the previous part, this chapter is focused particularly on the issue of scalability of parallel algorithm and the total workload of available computing hardware. The scalability is a key feature of algorithms and it gives the possibility of using a particular implementation of a method with regard to an increasing number of inputs and other resources. Today, scalability brings one more coherent aspect, and this is a total energy efficiency of computing. In the last decade, there was a relatively big boom of various supercomputing centres and increase in the acceptability of cloud services for parallel computing. A higher degree of parallelization required a higher need for task scalability. Due to a small possibility of scaling the task there is a decrease in total workload of available hardware. From the standpoint of programming, we speak about the so-called occupancy which means to reach the highest degree of hardware workload. An economic aspect is also taken into account currently when a total economic burden of all utilized means is calculated. The aim is not only to mass distribute parallel computing, but also to search for a balanced degree between costs of a particular computing. From the standpoint of parallelization on GPU, two different methodological approaches are being introduced to solve a particular task, which closely relate to scalability and occupancy. The aim was to solve especially nonlinear accesses to a memory and to eliminate synchronization points in a code. As an example, tree structures can be used whose single parallel implementation brings a great deal of drawbacks, especially when going through data structure and gathering constitutive calculations from processors. The possibility of using the real-time compilation units from a graphics processor will be emphasized. The utilization of compilation units in real time brings another view of algorithm construction, because a code, which will be executed, represents a dynamic result of preceding steps of an algorithm. Furthermore, the lowest level of work with data at the level of registers will be demonstrated, which leads to a great occupancy of the core of a graphics processor when using a suitable task scaling. The use of this approach was described in detail in the attached publications.
Chapter 1. Introduction of the research area

Included papers are:


1.2.3 Parallel computing and visualization

The last mentioned research direction is represented by a group of tasks which combine a parallel computing and visualization. Nowadays, this field involves especially the use of graphics processing units, because from a historical point of view they were designed in this way, and applications used GPUs exclusively for visualizations. With the arrival of programming pipeline and shader units, the graphics processors also became purely computing units. Today, it depends on a particular application domain or task which define if a card is used for computing, visualizing, or it is a combination of both. Current constructions of GPUs have enough computing cores and shaders for the production of an image. The benefit of a combined approach in the use of graphics processors is especially in Video Random Access Memory (VRAM) which is shared by computing and visualization parts of an application. With the unitary storage there is no need for the data transfer, which finally speeds up computing. On the other hand, we have to take into account that the memory of the best graphics cards varies somewhere in the region of tens of gigabytes, what brings some restrictions. When speaking about possible research tasks, we will mention a real-time visualization of graph layouts and indexation of spatial data. These tasks belong to the field of analysis of graphs, networks, or to the field of a basic analysis of data and algorithms, such as k-Nearest Neighbors (KNN), etc. The obtained results are described in the following papers.

Included papers are:


The following chapter deals with the overall description of a parallel programming domain, basic principles and classification of computer architectures in the way that we know them today. Each research area will be closely linked with a selected computer architecture, and all the achieved results, which are in the form of applications of parallel methods, will be focused on the utmost use of a selected technology. The application framework will be supported by selected publications.

2.1 Parallel computer architectures

Generally speaking, the parallel computing involves a group of computing elements which cooperate to solve a particular problem. Always, it is necessary to solve the question of performance and effectiveness in the meaning of computing time and used sources such as operating memory, computing units or a total used energy. A common goal of using parallel technologies is to reach a maximum speedup. Provided that some task or its part seems to be serial sometimes, it can be possible to change the approach leading to its solution and find a new way which will be, on the other hand, suitable for parallel computing.

As was implied, there is no single definition of the concept of a parallel approach especially because we have to primarily consider what level of a particular computer architecture we are using. For instance, the unrolling and subsequent workload of a processor caused by more instructions would be characterized as a part of Instruction Level Parallelism (ILP), and it would rather be considered as a code optimization. On the other hand, nowadays we can look at common computers as variants of parallel devices, and thus match the concept of parallel approach with the Flynn’s taxonomy [6]. The following overview is only a brief introduction of individual views on the parallel computing and it will serve as a clarifying text for the following chapters.

2.1.1 Flynn’s taxonomy

Flynn’s taxonomy is considered as a key classification of parallel computer architectures from the point of view of simultaneous processing of instructions which divides processor units into:

**Single instruction stream single data stream (SISD):** It is a type of classic computer architecture used almost within a single uni-core processor that can execute a single
instruction stream operating on data stored in a single memory.

**Single instruction stream, multiple data streams (SIMD):** This time, all parallel computing units share the same instruction set and perform the same operation on multiple data points simultaneously. This computer architecture has become very popular in digital image processing. It is sometimes known as a vector architecture which laid foundation especially of modern graphics processing units. From the point of view of parallel approach, this architecture is connected with the so-called Data Level Parallelism (DLP).

**Multiple instruction streams, single data stream (MISD):** From the perspective of used computer architectures it is not a commonly used architecture, and when applied on one computing unit, MISD practically doesn’t occur. It is a type of computer architecture where many functional units perform different operations on the same data. One example can be, for instance, the so-called Systolic array which represents a homogenous network of closely connected processor units known as nodes. Every node executes its own set of instructions on data obtained from a parent stream. It is very disputable whether to classify systolic array as MISD or not from the perspective of Flynn’s taxonomy.

**Multiple instruction streams, multiple data streams (MIMD):** This computer architecture belongs to one of the most developing. It involves especially processors that function on the basis of shared memory (Shared-memory multiprocessors) and the so-called Message-passing multiprocessors, including networks, clusters and grids. Here, we can find a microarchitecture Intel Larrabee whose development was finished in 2010 as a dead branch of GPGPU chips (General-purpose computing on graphics processing units). This microarchitecture was the basis for the further development of computer architecture Intel MIC and processors Intel Xeon Phi.

Flynn’s taxonomy is very comprehensive, but it neglects the synchronization between individual processors, even if it is a very important part of parallel computing.

### 2.1.2 Parallel architectures from the synchronization point of view

**Shared memory multiprocessors:** They represent an enhanced computer architecture, popular for its possibility of easy data sharing. They use shared address space where all processing units have the same access which can be realized in the following ways:

- **Uniform Memory Access (UMA):** All the processors in the UMA model share the physical memory uniformly. The access is provided by a common System Bus. The drawback of this type is a physical limitation of the number of processors. System Bus was later substituted by Interconnection network, which helped to increase the number of connected processors and total data throughput. The drawback of this approach is a requirement for the coherence of all the connected caches of all
processors. Moreover, it is necessary to solve the problems of access into the shared memory, which leads to the use of synchronization barriers.

- **Non-Uniform Memory Access (NUMA):** Processors can access the memory non-uniformly. The difference between NUMA and UMA is especially in the existence of a local memory which is available for the processor. The access to processor’s memory is direct and faster. NUMA-Cache Coherent (NUMA–CC) has an extra requirement when a cache must be coherent with the shared memory, otherwise it is not necessary.

- **Distributed Shared Memory (DSM):** Is a widely used form of computer architecture with the shared memory. Shared address space is provided by Virtual Address Space. Every processor has its own memory, and access to Virtual Address Space is executed according to the type of virtual memory.

**Single instruction stream, multiple data streams (SIMD):** This computer architecture can be included into the above described group, because it uses shared memory as well. Its concept fully corresponds with the Flynn’s taxonomy where every processor executes the same set of instructions simultaneously with other processors. As it is described below, there is a number of processors which are based on this computer architecture and extend it, as it is in NVIDIA graphics cards and its parallel architecture.

**Systolic processors:** Computer architecture is composed of processing elements (PE) which are mutually connected. Every element has its own small temporary memory. Processors are independent and they usually execute the same set of instructions, which is not indeed necessary. They are used in the fields where we can use Data Level Parallelism (DLP) such as pattern searching, data filtering, application of finite element methods, etc. Systolic processors are usually designed to execute a definite task, and a task modification means to change the whole system. The drawback is high system workload when obtaining and completing output data, which put enormous strain on the system I/O bandwidth.

**Clusters:** These are two or more connected processors, in majority of cases the whole devices in local area network (LAN). These can share local memory, usually it is hard drive computer storage RAID (Redundant arrays of inexpensive/independent disks). The significant role in synchronization is played by the so-called gather operation which obtains data from individual processors/devices, and then by the reduce operation which accumulates the result of parallel operations into one unit. The only one master processor/device is responsible for these operations.

**Grids:** Despite the above described clusters it is a computer architecture which makes use of wide area network (WAN). Today, we know this architecture under the title of “cloud”. It has a variable computing power depending on the number of connected processors and other sources such as memory, data flow volume or computer storage capacity. Synchronization is executed identically as in clusters.
2.1.3 Streaming multiprocessors

Streaming multiprocessors (SM) are closely tied with graphics processing units and they are based on SIMD or MIMD computer architecture. Every SM can execute its own set of instructions. Despite the classic CPU they have a higher number of registers, which allows higher thread independence in the sense of data processing. Registers can be matched to threads non-uniformly, which allows higher variability of code construction. The number of threads is limited when there is a higher need of registers. Processors have caches at several levels:

- Shared memory used especially for data sharing between individual threads within one unit.
- Constant cache accessible for all threads enabling fast read-only data broadcast.
- Texture cache enabling faster aggregate access to texture memory.
- L1 cache decreasing latency during global memory read-out.

Next important element, which is responsible for the difference between SM and classic processors, is Warp Scheduler. Warp is a group of threads and is the smallest unit executing the same instructions within all its threads. The concept of Warp is known in connection with hardware of NVIDIA. The competitor company AMD call these groups “wavefront”. The aim of warp scheduler is to plan the schedule of individual warps in streaming multiprocessor, whereas one streaming multiprocessor can have more warp schedulers. The knowledge of warp function mechanism plays a key role in the correct workload of a computer graphics card. From now on we are going to speak especially about NVIDIA hardware in connection with GPU [15, 87, 130, 139, 153].

The last well-known computer architecture of streaming multiprocessors is known under the term of GP100 from NVIDIA Pascal series [107]. One SM contains up to 64 cores for float computing and 32 cores for computing with doubles. If this is applied on the whole card, which has 60 SM, then the card has 3584 cores for float computing and 1792 cores for double computing, which enables a mass data and instruction parallelism even on one GPU.

The current concept of computer architectures is, up to a certain extent, very vague, and it is not always possible to place a particular architecture into a particular class. In connection with below mentioned architectures NVIDIA and their graphics processing unit we very often speak about Single Instruction Multiple Thread (SIMT). SIMT is placed somewhere between a classic SIMD and Simultaneous Multithreading (SMT). Simultaneous Multithreading is a parallel execution of threads with their own set of instructions. From the point of view of instruction execution, SIMT is more flexible than SIMD. In the case of NVIDIA graphics processing units, the execution of instructions is divided among streaming multiprocessors which will execute the same set of instructions on their threads.
Three crucial features that cannot be found in the classic SIMD architectures should be mentioned here:

1. Single instruction and multiple register sets: Threads within SM have a set of registers and they can process different data. For instance, one SM in NVIDIA Pascal architecture (GP100) has 64 cores for float computing and every core can run a thread. Within the programme it is possible to variably change the number of active threads and actively influence the scope of processed data.

2. Single instruction and multiple addresses: Threads can have an access to different address spaces. Accesses are independent, but they can compete if different threads try to read out or record to the same address. Similar behaviour can be simulated even in SIMD architecture, but causing a high latency. This feature is, in the case of SIMT architecture, frequently used during necessary random access to data (permutation, look-up tables, shuffling, etc.)

3. Single instruction and multiple flow paths: Threads at the level of streaming processor can be further executed. Before the execution, threads are logically divided into the so-called Warps, which represent a group of 32 threads that will execute given instructions. Nevertheless, due to a code which has a kind of contingent branching, threads are automatically executed (threads are stopped or synchronized for further mutual running, individual threads can be prematurely finished, etc.).

Finally, different streaming multiprocessors of one card can execute different sets of instructions, and we can encounter them under the term of Multiple SIMD (MSIMD). This is valid for NVIDIA Kepler architecture and others of higher rank.

2.1.4 Classification of parallelism in terms of software

As was implied above, the concept of parallelism can be considered in terms of hardware as well as software. The second one, software view is always connected with a particular application area and a particular task. The below mentioned levels of parallelism do not have to function independently. On the contrary, in a majority of well-known or freely accessible applications they merge, and the suitable use and combination influence, to a certain extent, the quality of final application in terms of efficiency and utilization of hardware.

Data-level parallelism (DLP): The easiest form in terms of transfer from serial approach to parallel approach is data-level parallelism. This is suitable for tasks where input data are independent and their processing does not significantly influence them. The example can be a parallel sum of two vectors of $M$ dimension, where every element of a vector is processed independently.

Instruction-level parallelism (ILP): This level is closely linked with the possibilities of a given hardware and represents the execution of more than one independent instruction by one processor at a given run time. The effect of ILP is significantly demonstrated at the
moment when there are more I/O operations coming into memory within data processing, which is much slower than the execution of arithmetical instructions. The example can be an easy sequence of mathematical terms:

1. \( X = A + B \) takes \( \sim 20 \) cycles to execute
2. \( Y = A + C \) is independent, can start anytime
3. \( Z = X + D \) is dependent, must wait for completion

An ideal case would be when one independent line was executed in \( \sim 20 \) cycles (clc) in NVIDIA architecture \[107\], which requires total instruction scheduling in the warp with a prerequisite that all the data will be in registers. In our case not all the lines are independent. The line 1 and 2 are executed in parallel due to ILP and their sequence will last \( \sim 20 \) clc. The execution of the third line will last other \( \sim 20 \) clc, because the computing is dependent on line 1. In case of ILP, it does not depend only on the possibilities of a processor, but also on a suitable schedule of instructions.

**Thread or Task-level parallelism (TLP):** The thread is the smallest sequence of instructions whose execution can be scheduled separately within one programme. This level of parallelism represents a form of data processing when more threads execute different instruction sets on different data. In terms of software, it is not important in what processor or core the thread is executed. The most important point is then the synchronization of threads in the case of concurrent output. As an example we can use a rendering of a scene, e.g. a graph, within one thread, and a concurrent preparation of data in another thread for the scene in the subsequent frame.

**Process level parallelism (PLP):** Under this term, we understand an instance of a programme running in a device which has its own sources assigned by a system such as memory, registers, processor time, etc. At this level, it is a classic multitasking enabling to run processes independently on input and output. For instance, a parallel running analysis of data in different instances of MATLAB; or from everyday life playing music from a media player and writing a document in \LaTeX{}.

### 2.1.5 API for parallel programming

Together with a long list of all the possibilities in the field of parallel accesses, there are also many programming languages and interfaces that support parallel programming. The choice of Application Programming Interface (API) always depends on a number of factors: application domain, solved task, accessible hardware, and last but not least, on an operating system. The last mentioned factor, the operating system, is not, from the perspective of threading, so important because a majority of current systems support multithreading. Nevertheless, it can have an influence on the parallel computing in, e.g. clusters or grids, because it contains protocols for a necessary communication, work with a virtual address space, etc. In connection with operating systems we can mention IEEE standard Portable Operating System Interface (POSIX) \[12, 57\] which is a uniform API providing compatibility throughout operating systems based on Unix. As a part of this
2.2 Selected APIs

API there is, among others, a support of thread construction and control, synchronization or memory control.

In general, API can be divided into two basic groups according to if they facilitate the access to hardware tools and support the creation of parallel application, or they focus on communication and synchronization during distributed computing. A common feature of the first group of API is a support of the creation of highly optimized codes in terms of particular hardware workload. Although there is a number of them, we can mention the following: Open Multi-Processing (OpenMP) [14, 13], Open Hybrid Multi-core Parallel Programming (OpenHMPP) [92], Open Accelerators (OpenACC) [29], Open Computing Language (OpenCL) [82, 103, 131, 140] or Compute Unified Device Architecture (CUDA) [15, 87, 130, 139, 153]. The attribute “Open” can indicate a certain openness in the development and support of many programming languages. In the second group of API there are for example Open Message Passing Interface (OpenMPI) [110, 121] implementing Message Passing Interface (MPI) that is primarily designed for communication in clusters and grids within distributed computing applications.

A specific group, which support parallelism at the lowest level, is represented by instruction sets of processors that are today focused especially on SIMD architectures, and their accessibility depends on the type of a processor. Here we can find Streaming SIMD Extensions (SSE), Advanced Vector Extensions (AVX) or Fused Multiply–Add (FMA) [91].

If the lowest level from the point of view of support of parallel application development has been mentioned, it is also very important to mention the opposite end, i.e. extensive frameworks enabling parallelism. Hadoop [135, 152] is the largest current framework that enables distributed data processing throughout a cloud with the use of an easy programming model. It uses a concept of Map Reduce [99] which enables data parallelism (DLP). Alternative to this Hadoop can be for example Cluster Map Reduce which partially refers to Hadoop and the concept of Map Reduce, but it is focused especially on a minimization of data transfer. The last mentioned framework is Hydra which is a distributed system for requirement processing focused on real-time analyses of extensive data, which is what distinguishes it from Hadoop.

2.2 Selected APIs

As was mentioned above, a large part of the research objective was focused on the utilization of NVIDIA graphics processors. In terms of the development, we could analyse graphics cards of all generations, from Tesla, Fermi, Kepler, Maxwell, to the current series of Pascal. We were allowed to observe the development in terms of architecture and computing possibilities of individual generations of this hardware. Nowadays, NVIDIA is preparing a new family of GPU cards with the title of Volta, whose main advantage will be the so-called stacked DRAM which will allow a bandwidth of almost 1TB/sec. and it will increase the efficiency of cards by a level higher. This card can appear on the market around 2018.
In terms of used application interfaces the choice was not so clear. The primary role was played by software equipment from NVIDIA and their API labelled as NVIDIA CUDA [15, 87, 130, 139, 153], but in some tasks some other approaches were used such as OpenCL [82, 103, 131, 140], OpenMP [14, 13], OpenACC [29], GLSL [129, 154], and others. The three most important are described below in the order as they have been used:

**NVIDIA CUDA** is an acronym for Compute Unified Device Architecture and NVIDIA corporation uses it to label its proprietary sets of tools that enable programmers to write a parallel code for graphics processors only with several superstructures to a programming language C. As a matter of course, there is a link between libraries and other languages such as Fortran, Python, C# and others. In connection with the utilization of a particular hardware it is necessary to observe the so-called compute capability represented by a number which indicates a set of features and instructions which are supported by a particular hardware.

The main advantages are:

- Libraries are written and optimized for specific hardware, which is demonstrated especially in speed.
- Free access is connected with a big community, a number of supporting development tools.
- Support of parallel computing on more cards: shared address space, direct communication between graphics processors.

The main disadvantages are:

- Suitable only for NVIDIA graphics and computing cards.
- More work for a programmer in comparison with the use of OpenMP on CPU.
- More complicated scalability with regard to different series of cards and different compute capabilities.
- More complicated synchronization in comparison with OpenCL.

**OpenCL** means Open Computing Language. It is a standard for the parallel programming of heterogeneous systems, whose development is primarily in hands of the syndicate of companies under the title of Khronos Group. The last versions of OpenCL already use the latest standards of C++14. It is necessary to mention that not all the hardware producers support the latest version.

The main advantages are:

- It is a multiplatform in terms of the use of GPU from different producers, which enables easier transfer of a code.
- It uses the so-called intermediate language Khronos SPIR-V which is unitary even for languages purely focused on graphics such as OpenGL or Vulkan [86, 134].
In comparison with NVIDIA CUDA, OpenCL has a support of real-time compilation and the execution of a new code within the application, which enables to write a dynamic code.

The main disadvantages are:

- With regard to wide support of hardware producers there can be ambiguity in terms of supported functions of OpenCL library.
- Smaller efficiency in comparison with NVIDIA CUDA.

OpenMP is an abbreviation of Open Multi-Processing and it is a collection of compilation directives, methods and variable environments which together define the OpenMP API for parallel programming in C, C++ and Fortran. Currently it is in version 4.5, and it is, same as OpenCL, supported by a number of hardware producers. It is focused on classic processors.

The main advantages are:

- Support of majority of compilers.
- Easy application by writing construction into an existing code, e.g., in C++ language. Most of the code is not changed.
- If it is not possible, the same programme can be executed without a support of parallelization, it means it will run serially.

The main disadvantages are:

- It needs a compiler which supports OpenML, not all the compilers support new versions of OpenMP. Some compilers, e.g., CL compiler from Microsoft, support OpenML only up to version 2.0.
- Predominantly used only for the cycle parallelization.
- Limited to the number of processors of one machine, it does not support the model of distributed shared memory.

2.3 The assessment of parallel approach efficiency

A correct selection of the method of data analysis and its suitable implementation can, up to a certain extent, influence the results of this analysis. For instance, this feature can be easily demonstrated in classification methods when using basic statistics such as sensitivity, specificity, model accuracy, F1 score, etc. In regression analysis, it is a matter of proposed metrics and calculation of a total regression error such as mean square error, $R^2$ metric, etc. In terms of a parallel approach, the most important part is a construction of the method at the lower level, i.e., at the level of its implementation. The assessment of benefits of parallelism does not have to be easy, and in most cases it is necessary to deal with a theoretical part of a possible parallelism as well as a practical analysis using a particular hardware architecture.
2.3.1 Speedup factor

The basic tool to assess the benefits of the parallel approach is the so-called Speedup Factor (S). This sets the ratio between a time necessary for the execution of a particular task on one processor and a time necessary for the execution of the same task with the use of N processors.

\[ S(N) = \frac{T_p(1)}{T_p(N)} \]  

(2.1)

, where \(T_p(1)\) is a computing time of one processor and \(T_p(N)\) is a computing time of \(N\) processors running in parallel. In an ideal case and for the so-called total parallelism, it is valid that \(S(N) = N\). It is only a theoretical linear speedup, because there are limits ignored for an access to a memory, necessary communication between processors, e.g. during reduction, etc.

In case of an access to data, even to internal shared memory or to the network in a distributed model, we have to take into account a longer time period which finally makes the parallel approach more costly in terms of computing time. The following points must be taken into account when designing a parallel algorithm:

**Network delay:** In the case of distributed computing in a grid or a cluster it is necessary to transfer data between separate machines via OpenMPI. This fact is de facto limited by the architecture of the network and accesses into different types of memories and storages.

**Memory collisions:** They occur in cases when more processors ask for an access into shared memory (distributed or local). It must be guaranteed that only one processor is allowed to access a memory in a given time. This is valid for the read-out access as well as for recording. Although, more limiting aspect in real situations is the recording into a memory.

**Memory bandwidth:** It is a degree with which it is possible to record data into a memory, or how to read them from a memory within one cycle. This degree does not relate to a total memory capacity.

**Memory hierarchy:** It relates to a hardware architecture and a memory level which are used by a processor. The execution of instructions is much faster than access to a memory, that is why it is better to do computing again than to store data temporarily. When storing data it is necessary to consider the level of a memory with regard to its speed, and this should be done in this order: register, cache, RAM/VRAM, electronic disc, magnetic disc, optic disk.

The ideal case is then a situation when it is necessary to execute \(N\) independent tasks theoretically while using \(N\) processors. With regard to an access to a memory additional variables are introduces – see Table 2.1:

Then the following equations are valid:

\[ T_p(1) = Nt_p \]  

(2.2)
2.3. The assessment of parallel approach efficiency

Table 2.1: Time variables in parallel \( N \) processor system with shared memory.

<table>
<thead>
<tr>
<th>Activity</th>
<th>Notation</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Read from memory</td>
<td>( T_r(N) )</td>
<td>The time necessary for the data read-out from the memory shared by ( N ) processors</td>
</tr>
<tr>
<td>Write to memory</td>
<td>( T_w(N) )</td>
<td>The time necessary for the data recording into the memory shared by ( N ) processors</td>
</tr>
<tr>
<td>Communication latency</td>
<td>( T_c(N) )</td>
<td>The latency during communication between two processors in ( N ) processor system</td>
</tr>
<tr>
<td>Data processing</td>
<td>( T_p(N) )</td>
<td>The time necessary for the data processing with the use of ( N ) processors</td>
</tr>
</tbody>
</table>

\[
T_p(N) = t_p \tag{2.3}
\]

where \( t_p \) is a time necessary for the processing of one independent task. In case of accesses into a memory, the time necessary for reading out accesses by one processor is given by the following formula:

\[
T_r(1) = Nt_m \tag{2.4}
\]

where \( t_m \) is an access time to a memory necessary for the reading of one block of data. In an ideal case, when the execution of one task needs only one access into a memory, we can use the following formula for \( N \) independent tasks:

\[
T_r(N) = \alpha T_r(1) = \alpha Nt_m \tag{2.5}
\]

where \( \alpha \) is a factor representing the limitation into shared memory dependent on a particular architecture. For the value \( \alpha \) three extremes are introduced:

\[
\alpha \begin{cases} 
= 1/N & \text{, if every from } N \text{ processors can process its own copy of data independently.} \\
= 1 & \text{, if data is distributed for every processor from a central memory.} \\
> N & \text{, if accesses of processors into a memory collide.}
\end{cases} \tag{2.6}
\]

The Equation 2.5 can be modified as follows:

\[
T_r(N) \begin{cases} 
= t_m & \text{for fully distributed memory.} \\
= Nt_m & \text{for shared memory without collisions.} \\
> Nt_m & \text{for shared memory with collisions.}
\end{cases} \tag{2.7}
\]

The same rules are valid for the memory record and it can be expressed as:

\[
T_w(1) = Nt_m \tag{2.8}
\]

\[
T_w(N) = \alpha T_w(1) = \alpha Nt_m \tag{2.9}
\]

Total time for the processing of input data by one processor is:

\[
T(1) = T_r(1) + T_p(1) + T_w(1) = N(2t_m + t_p) \tag{2.10}
\]
If the same task of data processing is performed by \( N \) processors, the total time will be given by the following formula:

\[
T(N) = T_r(N) + T_p(N) + T_w(N) = 2N\alpha t_m + t_p \tag{2.11}
\]

Speedup factor is then expressed by the formula:

\[
S(N) = \frac{T_p(1)}{T_p(N)} = \frac{N(2t_m + t_p)}{2N\alpha t_m + t_p} \tag{2.12}
\]

### 2.3.2 Theoretical prerequisites of speedup

From the above mentioned facts two laws are derived which define purely theoretical limits and benefits of parallelism in otherwise serial computing. To give a detailed explanation, it is necessary to mention the terms such as latency, workload and throughput.

**Latency (L)** sets the degree of delay between input and required output during the task processing and is given by \( L = T/W \), where \( T \) is a total time necessary for the task execution and \( W \) is a workload. Very often it is expressed in Cycles Per Instruction (CPI).

**Workload (W)** indicates a number of instructions being executed by a processor during a given time period or generally, the average amount of work handled by a processor at a particular instant of time.

**Throughput (Q)** represents the performance of tasks by a processor or device generally over a specific period and is given by \( Q = \rho N/L \), where \( \rho \) is a number of instructions in a computing pipeline (or density), \( N \) is a number of processors (or capacity) and \( L \) is latency. The most frequently it is expressed in Instructions Per Cycle (IPC).

**Amdahl’s law** [65, 68, 89], It expresses the theoretical speedup in latency of task execution with fixed workload. It is used for the formulation of the maximum possible speedup during the programme execution with the use of more processors. If \( p \) is a ratio of a parallel part of computing and \( 1\cdot p \) is a ratio of a serial part of computing, then the law can be expressed by the following equation:

\[
S(N) = \frac{T_p(1)}{T_p(N)} = \frac{N(1-p)t_p + Npt_p}{N(1-p)t_p + pt_p} = \frac{N}{N(1-p) + p} = \frac{1}{(1-p) + p/N} \tag{2.13}
\]

From this follows that to reach the maximum speedup \( (1-p) + p/N \) should be as small as possible, then \( p \to 1 \) for a multitude of \( N \) processors. In extreme cases, if \( p = 1 \), it will be a completely serial run and the speedup will be expressed by a number of parallel processors \( S(N) = N \). On the other hand, if \( p = O \), everything will run serially so that \( S(N) = 1 \).
Gustafson–Barsis’s law: [65, 68] The drawback of Amdahl’s law lies in its concept of fixed workload, in other words, fixed size of a problem and constant \( p \) which sets the degree of parallel and serial part of a programme. Gustafson and his colleague Barsis pointed out that with the growing problem grows also the degree of parallelism, and his concept of computation of theoretical speedup therefore assumes rather a constant time of computation:

\[
S(N) = \frac{T_p(1)}{T_p(N)} = \frac{(1 - p)t_p + Npt_p}{(1 - p)t_p + pt_p} = \frac{t_p((1 - p) + Np)}{t_p} = 1 - p + Np = 1 + (N - 1)p
\]

From the above mentioned formula follows that Speedup Factor can acquire values \( \gg 1 \) even if there is a small \( p \), but provided there is big \( N \) as well. This fact is more likely connected with practical assumptions while constructing parallel applications, because parallelism occurs primarily in critical and less frequent parts of an algorithm.
Regardless of the fact which theoretical approach is considered as the most suitable for a measurement, it is evident that the most important role is played by a degree of parallel parts expressed by $p$ with respect to the number of processors. As was mentioned in section 1.2, there are three main research directions focused on the use of graphics processors in the field of data analysis and processing.

In contrast to existing publications we would like to mention aspects of parallel programming that affect especially critical points during the implementation of selected methods of data analysis. By overcoming the critical points we could use graphics processors for a number of tasks of real data processing in information retrieval, from basic analyses, classifications, regressive tasks to pattern matching, indexing, or real-time visualization of extensive graph structures.

3.1 Research direction I: Parallel approach at the level of algorithm design

The first research direction is a general approach to parallel programming which crosses all the mentioned research activities in this thesis. The lowest level of abstraction means the level when parallel algorithms are constructed on graphics processors. The aim is to show that with the change of an approach during algorithm implementation we can reach a higher degree of parallelism and point out the fact that graphics processing units (GPU) are optimized for vector, or more precisely matrix operations. A case study devoted to this issue is focused on the parallelism of the selected method called Self-Operating Map (SOM). The same approach was used during the implementation of other bio-inspired methods such as genetic algorithms, symbolic regression, random forest, artificial bee colony and others. Their application and utilization in data analysis are linked with publications mentioned in chapter 5.

3.1.1 Loops elimination and higher vectorization

Majority of evolutionary algorithms or methods are based on iterative computing, testing of input data and gradual improvement of trained model or the calculation of errors which this model has in comparison with the well-known results of test data [16, 27, 60, 136]. There are always the same questions: “How many times are input data used? How many iterations are used? What metrics is used to calculate errors? What is the dimension of
a problem (input vectors)?” etc. Our aim was to use the computing potential of graphics cards maximally during the implementation of bio-inspired methods in terms of data sharing on different levels of memory, scalability and total workload of Arithmetic Logic Units (ALU) [113, 130].

From the point of view of mathematics the most direct way is the transfer of a problem into vector or matrix operations. These are maximally optimized for a particular hardware at the level of the so-called “device libraries”. What is more, when using NVIDIA CUDA, CUBLAS library can be directly used [7, 108, 157]. It is the implementation of methods in Basic Linear Algebra Subprograms (BLAS) [58, 104]. For instance, the actualization of neural network in Self-Organizing Map requires the calculation of weighted distance between every neuron and input vector.

Algorithm 3.1: Updating the distances in a given SOM network during a single epoch.

<table>
<thead>
<tr>
<th>inputs</th>
<th>m ... the number of input vectors</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>n ... the number of SOM neurons</td>
</tr>
<tr>
<td></td>
<td>dim ... the problem dimension</td>
</tr>
<tr>
<td></td>
<td>M ... the input data matrix of dimension $[m \times \text{dim}]$</td>
</tr>
<tr>
<td></td>
<td>N ... the weights matrix of the SOM network of dimension $[n \times \text{dim}]$</td>
</tr>
<tr>
<td></td>
<td>D ... the distance matrix D of dimension $[m \times n]$ from the previous epoch; zero matrix in case of first epoch.</td>
</tr>
<tr>
<td></td>
<td>$w$ ... a weighting vector of dimension $\text{dim}$ for distance calculation</td>
</tr>
</tbody>
</table>

output: Updated distance matrix $D$

1 for $i \leftarrow 1$ to $m$
2 for $j \leftarrow 1$ to $n$
3 for $k \leftarrow 1$ to $\text{dim}$
4 $D[i, j] \leftarrow D[i, j] + w[k] \cdot (M[i, k] - N[j, k])^2$
5 end for
6 end for
7 end for

If we take into account the fact that the Euclidean distance between two vectors $u$ and $v$ is expressed by Equation 3.1.

$$d(u, v) = \sqrt{\sum_{i=1}^{\text{dim}}(u_i - v_i)^2} = \sqrt{\sum_{i=1}^{\text{dim}}(u_i^2 + v_i^2 - 2u_i \cdot v_i)} \quad (3.1)$$

And if we consider that the root can be eliminated due to the redundancy within the whole selected algorithm, the calculation of distances within single epoch can be done according to Algorithm 3.2.
3.1. Research direction I:
Parallel approach at the level of algorithm design

Algorithm 3.2: Updating the distances in a given SOM network during a single epoch using matrix operations.

| inputs: | m | the number of input vectors |
|         | n | the number of SOM neurons |
| dim     |   | the problem dimension |
| M       |   | the input data matrix of dimension \([m \times \text{dim}]\) |
| N       |   | the weights matrix of the SOM network of dimension \([n \times \text{dim}]\) |
| D       |   | the distance matrix D of dimension \([m \times n]\) from the previous epoch; zero matrix in case of first epoch. |
| w       |   | a weighting column vector of dimension \(\text{dim}\) for distance calculation |
| o_m     |   | a row vector of dimension \(m\) filled by ones |
| o_n     |   | a column vector of dimension \(n\) filled by ones |

output: Updated distance matrix \(D\)

\[
D \leftarrow (N)^2 \cdot w \cdot o_m + o_n \cdot w^t \cdot (M)^2 - 2N \cdot \text{diag}(w) \cdot M^T
\]

/* ,where \((X)^2\) is the element-wise square matrix operation over the given matrix \(X\), and \(\text{diag}(x)\) is a function returning a diagonal matrix from the given vector \(x\). */

The mentioned Algorithm 3.2 can then demonstrate the purpose of the transfer to matrix operations. By the multiplication of the weighted vector \(w\) and the vector of \(\text{ones}\) we can calculate and store data in the constant or shared memory of the processor. In case of spacious data we can use a texture memory on GPU. All the scalar values are also constant and they are stored in the constant memory on GPU. As the access of threads is uniform, the access will be just as fast as in the case of using registers. Element-wise matrix operation can be parallelized by an easy function on GPU, whereas the method can be scaled based on the size of input matrix. What remains is to solve matrix operations with usage of CUBLAS library. In this case it is a gradual multiplication of two matrices because column vectors, or row vectors can be considered as matrices with one column, or one row respectively. Used method will be \(\text{cublasSgemm}\) for the multiplication of two column matrices with float data type. In case of the example with SOM method, the calculation and modification of SOM within one epoch would be done according to Algorithm 3.3. With regard to the fact that GPU matrix operations prefer a column definition all the necessary matrices were transposed. Furthermore, it is better to have a diagonal matrix \(W\) and if necessary to read a diagonal, than to have a vector of a diagonal \(w\) and to construct a diagonal matrix in the process.
Algorithm 3.3: Updating the distances during training phase using matrix operations.

**inputs**: noEpochs ... the number of SOM epochs
m ... the number of input vectors
n ... the number of SOM neurons
dim ... the problem dimension
M ... the input column matrix of dimension $[\text{dim} \times m]$
N ... the column weights matrix of the SOM network of dimension $[\text{dim} \times n]$
D ... the distance matrix $D$ of dimension $[m \times n]$ from the previous epoch; zero matrix in case of first epoch.
W ... a weighting diagonal matrix of dimension $[\text{dim} \times \text{dim}]$
$\text{o}_m$ ... a row vector of dimension $m$ filled by ones or a matrix of dimension $[1 \times m]$
$\text{o}_n$ ... a column vector of dimension $n$ filled by ones or a matrix of dimension $[n \times 1]$

**output**: Updated distance matrix $D$

*Init phase:*

1. $A \leftarrow o_n \cdot \text{diagonalOf}(W)$ /* $[n \times \text{dim}] \leftarrow [n \times 1] \cdot [\text{dim} \times \text{dim}]$ */
2. $B \leftarrow \text{diagonalOf}(W) \cdot o_m$ /* $[\text{dim} \times m] \leftarrow [\text{dim} \times 1] \cdot [1 \times m]$ */
3. $M_2 \leftarrow (M.)^2$ /* $[\text{dim} \times \text{m}] \leftarrow [\text{dim} \times \text{m}] \cdot [\text{dim} \times \text{m}]$ */

*Create model:*

4. for $e \leftarrow 1$ to noEpochs do
5.   if SOM weights changed then
6.     $N_T \leftarrow -2 \cdot N^T \cdot W$ /* $[n \times \text{dim}] \leftarrow [n \times \text{dim}] \cdot [\text{dim} \times \text{dim}]$ */
7.     $N_2 \leftarrow (N.)^2$ /* $[\text{dim} \times n] \leftarrow [\text{dim} \times n]$ */
6.  end if
9.  $D \leftarrow N_T \cdot M$ /* $[n \times m] \leftarrow [n \times \text{dim}] \cdot [\text{dim} \times m]$ */
10. $D \leftarrow D + A \cdot M_2$ /* $[n \times m] \leftarrow [n \times m] + [n \times \text{dim}] \cdot [\text{dim} \times m]$ */
11. $D \leftarrow D + N_2^T \cdot B$ /* $[n \times m] \leftarrow [n \times m] + [n \times \text{dim}] \cdot [\text{dim} \times m]$ */
12. Find the Best Matching Units (BMUs)
13. Update SOM
14. end for

/*, where $(X.)^2$ is the element-wise square matrix operation over the given matrix $X$, and \text{diagonalOf}($X$) is a function returning a diagonal of the given matrix $X$. */
3.1. Research direction I: Parallel approach at the level of algorithm design

Listing 3.1: CUBLAS implementation of the row 11 of Algorithm 3.3. All variables correspond to definitions in Algorithm 3.3; \( \alpha \) and \( \beta \) are constants set to 1.

```c
__host__ inline void updateD()
{
  cublasStatus_t stat = cublasSgemm(handle, CUBLAS_OP_N, CUBLAS_OP_N,
    n, m, dim, &alpha, A, n, M2, dim, &beta, D, n);
  if ( stat != CUBLAS_STATUS_SUCCESS )
  {
    cout << "CUBLAS error" << endl;
  }
}
```

As was already mentioned, the multiplication of matrices is done using methods of implemented linear algebra in CUBLAS library. A simple example of the code (see Listing 3.1) shows the concrete implementation of line 11 of Algorithm 3.3 only when all the data is stored in video memory of the graphics card.

The decomposition of the algorithm to vector operations is a good assumption to speed up applications in SIMT architectures. In most cases, the decomposition means a higher requirement on the memory which must be allocated in case of bulk data processing. Iterative approach which would be more friendly to memory utilization would mean, in case of using GPU, more frequent data transfer to graphics card, or more frequent access to structured global memory. In both cases this would mean a significant slowdown of the application. In contrast, reading matrices from a memory where data is well aligned, is not a burden for the graphics card. Using current computer architectures, the problem with a memory size can be solved with the so-called unified memory which uses Unified Virtual Addressing (UVA) [107]. It is possible to use RAM and VRAM uniformly without distinguishing which memory we can access at the level of a code. The price of extension of graphics card memory is at the expense of the data transfer via PCI-Express. Next possibility is to utilize more graphics cards and connect them together using NVLink [107].

The above mentioned approach brings a number of advantages in the form of application speedup or the possibility of using highly optimized BLAS libraries. From the general point of view of algorithm construction, it is also necessary to mention some drawbacks of such parallelism. Vector or matrix operation scheduling leads sooner or later to the need of solving another problem, thus to reduce partial results back to one scalar value. An example can be the calculation of a fitness function and a search for the best individual in one population with the best fitness function. General process and theoretical schedule of reduction is described in graphics processing units in detail. Problems arise with the variability of hardware architectures and their possibility of scaling the reduction to more streaming multiprocessors. It is necessary to solve the question of synchronization, size of shared memory, parallel threading in warps and/or shuffle operations on the level of separate registers. This will be described in the following chapters.
3.1.2 Application domains and achieved results

Over the years we succeeded in implementing a number of bio-inspired methods, during the construction of which we used the above mentioned approach. We have to point out that our aim was not only to implement selected methods, but we also wanted, as a part of the second level of our research and publishing activity, to find out the possibilities for the utilization of these methods in real data analysis without any need for the specification of what method is selected. Effective implementation played very important role, though on the other hand, it was not a key role when comparing it with the context of really carried out analyses. The meaning of this is that one part of our research would not make sense without another one.

The first results of our research include the implementation of Self-Organizing Map which was described above as an example showing that with the change of an iterative calculation to a batch calculation using GPU in vector and matrix operations we can achieve a high degree of parallelism and final speedup. This topic was described in papers [52, 119] which closely mapped obtained results and experimental comparison with the implementation on CPU. The result was an optimized method for GPU which was subsequently used for the real data analysis.

The examples are:

- **Tension Prediction - a Real-time simulation and prediction of physical values:**
  The computation of the relative deformation $\varepsilon$ and the tension $\sigma$ within the selected construction points can be done by using simplified, but precise enough approach based on the analytical simulations. Proposed simulations try to express dependences between input parameters ($\varepsilon$, $\sigma$) and measured parameters temperature $T$, overpressure $p$, and flow rate $Q$ of a medium running inside the conduits, chambers, and heat transfer tubes of the energetic facilities. Next, the computation of the relative deformation $\varepsilon$ and tensions $\sigma$ is varying in time and represents a dynamic system. This is more important especially in case of more complex constructional shapes of the mentioned facilities. Then the computation can be performed e.g. by using the Finite Element Method (FEM) [71, 11]. Unfortunately, FEM is very time-consuming and cannot be effectively used in the continuous computations of $\varepsilon$ and $\sigma$. All predictions were substituted by SOM (Figure 3.1). More information on the application can be found in [43].
3.1. Research direction I: Parallel approach at the level of algorithm design

Fig. 3.1: The practical comparison of simulation on GPU: An example of the achieved results [43]. A) FEM visualization of tension simulation; B) The comparison of the normalized results of FEM and SOM in the process of computation of the tension $\epsilon$.

- Mining spatial and temporal anomalies, and monitoring and analysis of the performance of the emergency call-taking system: The research dealt with an emergency call-taking information system in the Czech Republic which forms a network of cooperating emergency call centers processing emergency calls to the European 112 emergency number. Large amounts of various incident records were stored in the databases, and the data was used for mining spatial and temporal anomalies. Monitoring and analysis of the performance of the emergency call-taking system was described in more detail in [138]. A method for knowledge discovery and visualization targeted at the performance analysis of the system with respect to the organization of the emergency call-taking information system and its data characteristics was introduced. To handle the massive data, the growing grid algorithm is implemented in a parallel environment using compute unified device architecture. Experimental results illustrated that the proposed method is very efficient (Figure 3.2).

Fig. 3.2: Speedup of the final implementation of the selected method: An example of the achieved results [138]. GPU based implementation led to higher performance of the whole system.
With the deployment of real applications, we came across a problem with the size of input data of the graphics card memory. As was mentioned above, this problem can be solved in a several ways. In addition to the use of graphics processing units we focused on data parallelism and the modification of algorithms for gradual batch data processing. This approach did not fully work for all the methods, but for the above mentioned SOM method we managed to design a new modified algorithm GM-SOM which solved the calculation of neuron network by a subsequent linking of results of partial calculations [36, 37, 38, 39].

As was said above, the principle of algorithm decomposition and transfer to matrix operations was used in other methods and analyses of real data. As an example we can use the implementation of Extreme Learning Machine (ELM) [80] with the use of graphics processing units. We compared several versions of implementation according to the changing input data. The same happened in the next method which was Artificial Bee Colony (ABC) [75].

If we do not strictly follow the bio-inspired methods, we can also introduce implementations of purely matrix tasks such as Singular Value Decomposition (SVD) or Orthogonal Matching Pursuit (OMP) and their subsequent utilization in data classification. With regard to a nature of methods and input data the most optimal were tasks from the field of signal processing. We had the possibility to use the classification on real collections from Electrocardiography (ECG) or Electroencephalography (EEG) [118, 20].

Included papers are:


3.2 Research direction II: Improving scalability and occupancy

The previous chapter was focused on the issue of algorithm implementation and set up links to several publications which described the utilization of a selected approach in a particular real data analysis. This chapter deals with the issue of scalability of parallel algorithms [62, 66] and total workload of available computing hardware. Scalability is the ability of a system, process, or at the lower level, a particular implementation of an algorithm to react to the growing amount of inputs and adapt to this fluctuation. At the
3.2. Research direction II: Improving scalability and occupancy

level of methods, it is one of the key features which is currently connected with another aspect of total energy efficiency of computing. In the last decade, various supercomputer centres popped out very quickly, and the accessibility of cloud services for parallel computing increased as well. A higher degree of parallelism has brought a higher need for task scalability. Total workload of available hardware tools decreases when there is a small possibility of scaling a particular task. From the standpoint of a programmer, it is the so-called “occupancy” [107, 130] whose aim is to reach the highest utilization rate of available hardware.

Scalability can be viewed from many angles. When implementing all methods which could be used on available hardware, we encountered two main scalabilities:

- **Scalability in the context of fixed task size:** It is a task solution where fixed amount of inputs is assumed, but we are interested in the solution which finally uses more processors, cores, etc. We analyse a situation how many processors we need to solve the task effectively, and how to meet the objective when the solution is uniformly distributed to separate processors. In this group of solved tasks there are, for instance, implementation of methods for data analysis which use non-linear data structures and where the aim is to correctly solve non-linear inputs to memory and eliminate synchronizing points in particular blocks of a code. This work closely looks at this issue when explaining one of the selected methods, more precisely the method of symbolic regression [10, 84] which is based on general tree structures [94, 69, 151] and evolutionary algorithms. Its parallel implementation brings many drawbacks, especially when traversing data structures, during the growth of a tree structure and when obtaining constitutive outputs from processors (gathering).

- **Scalability from the perspective of variable task size:** In this case there is a hidden assumption that a hardware platform which will be used for the problem solution is fixed (e.g. we want to utilize 50% of available processors), whereas the solved problem is variable. As an example can be used the implementation of methods which solve some combinatorial or permutation problems such as flow-shop [63, 64], job-shop [31, 67], etc., or even general methods used for the processing of streaming data. The second case involves the implementation of methods for real-time analysis of signals [3, 17].

3.2.1 Parallel implementation with regard to scalability:

**Fixed size of a task**

Within this research direction, bio-inspired methods based on tree structures were implemented and used, such as symbolic regression, random forest, or fuzzy rules [10, 93, 150]. All of them were basically used for the solution of classification [2] or regression tasks [59]. From the perspective of parallelism, we would like to mention only one method – symbolic regression, because the mechanism of operating internally used non-linear structures is for all methods similar and the principle of distribution of computations among processors is for all mentioned methods basically the same.
Example description:
Let’s say we have a set of $N$ input vectors of dimension $k$. Then $v[0]$ represents the first element of input vectors. The formula
$$r_i = (0.5 \times 0.123) + (0.5 \times ((0.7 \times v[0]) \times (0.3 \times 0.456)))$$
returns a result $r_i$ for every $i$-th input vector, where $i \in N$.

Fig. 3.3: An illustrative result of a model of symbolic regression.

Symbolic regression

Symbolic regression (SR) via genetic programming [10, 85] is a branch of empirical modelling that evolves summary expressions for available data. Although intrinsically difficult (the search space is infinite), recent algorithmic advances coupled with faster computers have enabled application of symbolic regression to a wide variety of industrial data sets. Unique benefits of the symbolic regression include human insight and interpretability of model results, identification of key variables and variable combinations, and the generation of computationally simple models for deployment into operational models.

The challenging task of the symbolic regression is to identify and express a real or simulated system or a process, based on a limited number of observations of the system’s behaviour. The system under study is characterized by some important parameters which need to be available for an observer, but are usually difficult to monitor, e.g. they need to be measured in a laboratory, simulated or observed in real time only, or at high time and computational expenses. Empirical modelling attempts to express these critical variables via other controllable variables that are easier to monitor, can be measured more accurately or timely, are cheaper to simulate, etc. Symbolic regression provides such expressions of crucial process characteristics, or, response variables, defined (symbolically) as mathematical functions of some of the easy-to-measure input variables, and calls these expressions empirical input-output models (or input-response models).

A particular solution of the symbolic regression in the form of mathematical formula can be illustrated by a tree structure as well, see Figure 3.3. Some nodes have different colours, which will be discussed in more detail later. The edge or edges can have weights to make the whole model more fuzzy.

The set of symbols (functions) depends on the application area and represents one of the advantages of symbolic regression. In statistics, attribute dependences can be simulated and tested with a set of symbols of logical functions $S = \{\land, \lor, \supset, \equiv, \neg\}$, in the area of signal processing with a set of goniometric functions $S = \{\sin, \cos\}$, etc.

To discover acceptable models with realistic time and computational effort, symbolic regression exploits a stochastic iterative search technique, based on the artificial evolution of model expressions. This method called genetic programming looks for appropriate expressions of the response variable in the space of all valid formulas containing a minimal set of input variables and a proposed set of basic operators and constants.

Evolutionary algorithms [16, 136] are stochastic search methods that mimic the meta-
phor of natural biological evolution, which applies the principles of evolution found in nature to the problem of finding an optimal solution to a solved problem. An evolutionary algorithm is a generic term used to indicate any population-based optimization algorithm that uses mechanisms inspired by biological evolution, such as reproduction, mutation and recombination. Candidate solutions to the optimization problem play the role of individuals in a population, and the cost function determines the environment within the solutions “live”. Evolution of the population then takes place after the repeated application of the above operators. Genetic algorithm (GA) is the most popular type of evolutionary algorithms. It was described by John Holland in 1960s and further developed by Holland and his students and colleagues at the University of Michigan in the 1960s and 1970s. GA used Darwinian Evolution to extract nature optimization strategies that use them successfully and transform them for application in mathematical optimization theory to find the global optimum in a defined phase space [61, 97, 98]. We refer to [28, 70, 72, 81, 111, 112, 133, 142] for more detailed information on the usage of GA.

Process of implementation with the use of real-time compilation on GPU

Parallel implementation of a selected data analysis method which, in general, internally uses dynamically changing tree structures was in early GPU architectures basically ignored, because it induced high overhead costs in the form of inefficient utilization of streaming multiprocessors and their access to shared memory. A possible utilization of textures for fast random access to memory was stymied by the problem of dynamic data, which meant that a new texture on GPU had to be constructed whenever a new change occurred. All the overhead costs degraded the importance of GPU utilization in these tasks, because optimized implementation on common processors brought better results in terms of computing time. New GPU architectures brought the so-called dynamic parallelism [107, 130] that allowed to dynamically branch calls of GPU functions and kernels, which led to the formulation of a prediction for the construction of algorithms using recursion and embedded calls of kernel functions. This partially solved the problem of traversing non-linear structures, nevertheless the issue of dynamic data has remained, which means that it is still not possible to uniformly plan the execution of kernel functions on all processors. The example can be the traversal of ten tree structures, when every tree has a different depth and different branching. These possibilities can occur when:

1. One thread processes one tree: Then it is significantly inefficient, because threads execute a heterogeneous code, and in case of a streaming processor they become serial.

2. One block of threads processes one tree: If the walk of the tree requires sequencing, the threads will be dependent on each other. This would lead to the synchronization or suspension of threads.

3. One streaming multiprocessor would process one tree: this would lead to the same problem as in the above mentioned case. What is more, this would lead to a minimum occupancy of the processor.
4. The view on the task will be changed: Data parallelism will be utilized in situations when we do not want to find parallelism in individual walks of the tree. At the same time, it must be ensured that all processors will be walking the same tree, but assessing different input data.

Only point 4 evidently seems to be a reasonable solution from the perspective of task scalability. On the basis of fixed inputs, we can exactly schedule hardware tools and workload of cores of separate processors. At the same time, all cores execute the same instructions because they walk the same tree structure and they have a homogenous access to memory. The problem with dynamic data has de facto remained, but it does not seem to be an efficiency threat. Data is updated in global memory and, for homogenous read-out operations, they are stored in cache to provide faster access to subsequent read-out.

Regarding the implementation of symbolic regression method we went further. Our aim was to find a method for classification or regression which would process a large amount of input data. In this case, scalability has certain limitations regarding hardware possibilities of graphics processor chips, and the effort to speed up the whole operation then leads to the elimination of access to memory and substitution of these accesses by arithmetical operations with lower latency, which will be explained later. Listed Algorithm 3.4 shows basic steps of the construction of symbolic regression model provided that all the data is stored in video memory of the card.

All steps of Algorithm 3.4, apart from line 8 and 13, do not require any specific operation with input data. It is only a modification and set-up of vector data of tree structures in memory. As soon as the number of trees does not change and all the next inputs and variables are fixed, subtasks can be easily scaled, and their computations can be scheduled to separate processors. With the reference to the above mentioned transfer of I/O inputs to memory using the arithmetic operation and increasing the efficiency of algorithm for large amount of input data, the key step was Algorithm 3.4, lines 13 and 8 in the subsequent computing iteration. Graphics processing units are capable of the so-called online compilation [82, 103, 131], which means that we can prepare a code in the form of a structured text string and compile this string into any executable assembly directly for a particular graphics card while the application is running. It is a form of Just in Time (JIT) compilation. The advantage of this compilation is a code portability, but in our case, the most important fact is that we can transfer the walk of the tree into a particular method and particular instructions. One drawback of JIT compilation is a time necessary to compile an assembly, which is, in case of large amount of inputs, eliminated by a faster computing time. The disadvantage of JIT compilation is the fact that not all APIs offer this possibility, and thus OpenCL has substituted previously mentioned NVIDIA CUDA technology during the implementation of symbolic regression method. Shortened example of the real-time compilation of the code in OpenCL is shown in Listing 3.2.
3.2. Research direction II: Improving scalability and occupancy

Listing 3.2: The example of real-time compilation C/C++ with API OpenCL. Assumption is that all the variables listed for the method are already set and mSource includes a source code ready for the compilation.

```c
char* mSource = (char *) malloc(sizeof(char) * (1<<26));
cl_context mGPUContext = nullptr;
cl_command_queue mCommandQueue = nullptr;
cl_platform_id mPlatform = nullptr;
cl_device_id mDevices = nullptr;
cl_program mProgram = nullptr;
cl_kernel mKernel = nullptr;

void createKernel()
{
    if(mKernel) { clReleaseKernel(mKernel); mKernel= nullptr; }
    if(mProgram) { clReleaseProgram(mProgram); mProgram = nullptr; }

    size_t sourceLength = 0;
    cl_int clError = 0;

    sourceLength += sprintf(& mSource[sourceLength], "... C/C++ code ...");

    mProgram = clCreateProgramWithSource(mGPUContext,
        1,
        (const char**) & mSource,
        & sourceLength,
        & clError);

    checkError("clCreateProgramWithSource", __LINE__);

    #ifdef MAC
    char* flags = "-cl-denorms-are-zero -cl-fast-relaxed-math -DMAC";
    #else
    char* flags = "-cl-denorms-are-zero -cl-fast-relaxed-math";
    #endif

    clUnloadCompiler();
    clError = clBuildProgram(mProgram, 1, mDevices, flags, NULL, NULL);
    checkError("clBuildProgram", __LINE__);

    hlKernel = clCreateKernel(mProgram, "mKernel", &clError);
    checkError("clCreateKernel (mKernel)", __LINE__);
}
```
Algorithm 3.4: Construction of symbolic regression model with the use of genetic algorithm (GA).

**inputs**: m ... the number of input vectors
n ... the number of trees
dim ... the problem dimension
nEpochs ... the number of GA epochs
nBests ... the number of best trees to be stored for next epoch
pDel ... the percentage of deleted trees after each GA iteration
pDup ... the percentage of deleted trees that will be substituted by clones of best trees

**output**: The tree with the best fitness value at the first place in the array of sorted trees.

**Init phase:**
1. \( ndt \leftarrow pDel \cdot n \)
2. \( nct \leftarrow ndt \cdot pDup \)
3. \( nnt \leftarrow ndt - nct \) /* the number of new generated trees */
4. \( nmt \leftarrow n - nnt \) /* the number of mutated trees */
5. \( nct \leftarrow n - nnt \) /* the number of combined trees */
6. \( \text{initTrees} (n) \)

**Create model:**
7. for \( e \leftarrow 1 \) to \( nEpochs \) do
8. \( \text{evaluateTrees} () \) /* get fitness value for every tree */
9. \( \text{sortTrees} () \) /* based on values of fitness functions */
10. \( \text{storeBests} () \) /* temporary store of current best trees */
11. \( \text{deleteBadTrees} (ndt) \)
12. \( \text{cloneBestTrees} (ndt, nct); \) /* a part of deleted trees will be substituted by clones of best trees */
13. \( \text{generateNewTrees} (nnt) \)
14. \( \text{mutateTrees} (nmt) \)
15. \( \text{combineTrees} (nct) \)
16. end for

### 3.2.2 Parallel implementation with respect to scalability:
**Variable size of a task**

The previous case emphasized the importance of fixed task size. In other words, we can say that the size of a solved problem offers a number of ways of its implementation, and it only depends on the design of an algorithm how it is divided on separate processors, cards, nodes, etc. Processors do not have to execute the same instructions or operate the same data structure. This is completely different in situations when we consider these available processors as devices that will have to be utilized, and the question is how to approach the issue of inputs. In majority of these tasks, inputs are large amounts of data, or they are dynamic data, e.g. streams and their continuous processing.
There is a number of problems that can be solved by using task vectorization and subsequently distributing the solution among processors/cores/threads with the use of standard principle of data parallelism. The basic example can be the addition of two vectors \( \vec{w} = \vec{u} + \vec{v} \) of \( m \) dimension, where every element \( w_i \) is computed by one thread as \( w_i = u_i + v_i \), where \( i \in \{1, m\} \). In the analogue case of addition of two matrices \( W = U + V \) with size \( [m \times n] \), where every element can be again computed by one thread as \( W_{i,j} = U_{i,j} + V_{i,j} \), where \( i \in \{1, m\} \) and \( j \in \{1, n\} \). Although the task is still linear, it is clear that with the growing dimension of variables we will exceed the amount of possible allocated threads, and thus it is necessary to explore another algorithm construction for the addition of elements so that the thread would process more instructions and does more basic additions.

Within the research we implemented tasks which were solved by means of instruction level parallelism (ILP) and optimization at the lowest level of memory control. It enabled CUDA cores to be fully utilized when using the access to memory with low latency. To illustrate this, the optimization of a selected scheduling problem, where the use of ILP is demonstrated on the computation of one schedule table, will be mentioned. More information can be found in the attached publication \[9\] in which we focused on the implementation of Self-Organizing Migration Algorithm (SOMA) to solve the problem.

**Flow-shop scheduling problem**

It is one of the classic construction problems \[31, 67\] which can be briefly described as follows: Let a set of \( \mathcal{M} \) production machines and a set of \( \mathcal{J} \) production jobs be given. We are trying to search for the most optimal schedule of a set of \( \mathcal{J} \) tasks for a set of \( \mathcal{M} \) machines, which means that every job \( j \in \mathcal{J} \) must be executed on all machines \( m \in \mathcal{M} \), and a particular job \( j \) can be executed just only on one machine \( m \) at a particular moment. If \( M = |\mathcal{M}| \) and \( J = |\mathcal{J}| \) and the set of machines is firmly scheduled from \( m_1 \) do \( m_M \), the number of possible solutions is equal to the number of all permutations, or \( J! \), which in case of \( J = 10 \) means 3628800 possible solutions. We usually compute in thousands of jobs and hundreds of machines, and thus it is evident that the computing by “brute force” will not be possible. In this case, it is necessary to focus on some evolutionary algorithms again, which is described in the attached publication \[9\]. This task is mentioned because of another reason which involves the utilization of a processor for the computing of one plan, one permutation of jobs \( J \) on machines \( M \). Reaching a high level of parallelism can finally enable to scale the task on the basis of the necessity of computed permutations.

**Implementation using synchronous instructions at the level of WARP**

Architectures of NVIDIA Kepler \[106\] and higher introduced several new features that can significantly decrease kernel run-time. Shuffle instructions brought another way of sharing data among threads within the same WARP, in addition to shared memory utilization. However, using these instructions has its limits, e.g. at most 32 threads (= warp size) can be affected by the shuffle instruction call, the number of used registers increases, thread/lane indexing can make some code parts more complex, etc. On the other hand,
shuffle instructions can reduce the amount of utilized share memory, eliminate thread synchronization barriers, or reduce total number of instructions with respect to thread data processing and warp data transfers. This can keep CUDA cores busy with memory accesses that have low latency and high bandwidth. The shuffle instructions were primarily used in case of Block/Warp Computation.

Such implementation represents the next level of parallelism, where a set of threads cooperates and computes a single schedule. Next Figure 3.4 shows a successive evaluation of cells in the grid of a schedule marked by zero-based indices. The same indices mark the cells that are processed in parallel. Let the number of jobs be \( J \). Then there is an inner loop of \((J + B - 1)/B\) steps, where \( B \) is a block size; a block of 4 threads was used in this illustrative example. In every step, a strip of at most \( B \) columns is computed, so that all values in the last active column are called border values and are stored in the shared memory for the next step. If \( t \in \{0, 1, 2, 3\} \) is a thread index and \( s \) is the zero-based step index, then a thread \( t \) computes the whole column \( s \cdot B + t \) in every step. In case of \( B < 32 \), where 32 is the warp size \([106, 107, 130]\), threads can store intermediate data in registers and share them by shuffle instructions to achieve the best performance.

A strip of \( B \) columns is computed in a loop of \( M + B - 1 \) iterations. If a thread computes a single value (cell of a schedule), then it moves down and evaluates a new cell in the following iteration reusing previously computed value that was shuffled to subsequent thread simultaneously. The cells marked by number 3 were evaluated in the 4-th iteration, which is the first one, where all threads run the same instructions in parallel. Until then, only \( i + 1 \) threads were active, where \( i \in \langle 0, 1, \ldots, M + B - 1 \rangle \) is the zero-based iteration index. Next, all threads run parallel while \((B - 1) <= i < M\). Finally, the threads subsequently finish their computations during the last \( B \) iterations. After that, the thread block shift right and process the next column strip. The whole method can be seen in Listing 3.3, with respect to the kernel call, such that a single schedule is processed by a single WARP. It means, that a single CUDA block should have 32 threads or a multiple of 32.

This strategy is suitable for faster processing of schedules with respects to GPU limits. As it was mentioned above, the usage of shuffle instructions needs CUDA blocks of at most 32 threads. Moreover, this number of threads limits the total number of active blocks. Although such implementation is more complicated, it finally brings significant performance improvement and it is more suitable for DSOMA in general. More information can be found in \([9]\).

3.2.3 Application domains and achieved results

Both above mentioned approaches, the real-time compilation of a code and processing of non-linear structures, or the optimization of memory accesses at the level of Warp, require advanced knowledge of architectures, and from the perspective of task parallelism they are considered as one of the key domains which are still developing nowadays. In our research, we solved a number of tasks using these two approaches, or at least one of them. It is evident that both approaches can be applied at the level of data structures, and thus it does not depend on a particular application domain.
3.2. Research direction II: Improving scalability and occupancy

Listing 3.3: CUDA kernel function: Computation of one plan in warp using shuffle instructions according to Figure 3.4.

```c
__global__ void getSchedulesWPSKernelB(
    const unsigned int* __restrict__ mIds, // Machines indices
    const unsigned int* __restrict__ jIds, // Jobs indices.
    const cudaTextureObject_t tex, // machines/jobs times
    float* fvs) // the final fitness values = Time
{
    const unsigned int tid = threadIdx.x;
    const unsigned int laneID = tid & WARP_SIZE_MINUS_ONE;
    const unsigned int warpID = tid >> WARP_SIZE_SHIFT;
    const unsigned int sid = blockIdx.x*WPS_WARPS_PER_BLOCK + warpID;
    if (sid >= noS) return;

    unsigned int nat = WARP_SIZE; // number of active threads
    unsigned int i = 0; // warp loop variable
    unsigned int mid = 0; // loop variable - machine ID
    unsigned int jid = laneID; // loop variable - job ID
    float l = 0.0f; // left neighbor value in the matrix
    float x = 0.0f; // currently computed value
    unsigned int run = 1; // stopping criteria

    extern __shared__ float sBorderValues[];
    float* ptrSM = &sBorderValues[warpID * noM];

    for (i = laneID; i < noM; i += WARP_SIZE) { ptrSM[i] = 0.0f; }
    if (BLOCK_SIZE > 32) __syncthreads(); // because of more WARPs in the BLOCK

    for (jid = laneID; jid < nextMultipleOfWarp(noJ); jid += WARP_SIZE)
    {
        int nat = ((jid >> WARP_SIZE_SHIFT) == (noJ >> WARP_SIZE_SHIFT)) ? noJ & WARP_SIZE_MINUS_ONE : WARP_SIZE;
        int i = 0;
        while (run && (laneID < nat))
        {
            if (laneID == 0) l = ptrSM[mid];
            if ((laneID < i) && (mid < noM))
            {
                if (useCMF)
                    x = MAXIMUM(l, x) + tex2D<float>(tex, getJIDinCMF(jIds, sid, jid),
                                               getMIDinCMF(mIds, sid, mid));
                else
                    x = MAXIMUM(l, x) + tex2D<float>(tex, getJIDinRMF(jIds, sid, jid),
                                               getMIDinRMF(mIds, sid, mid));
            }
            if (laneID == (nat - 1))
            {
                if (mid == noM-1) run = 0;
                ptrSM[mid] = x;
                mid++;}
            else
                l = __shfl_up(x, 1);
            run = __shfl(run, nat - 1);
            i++;
        }
    }
    if (nat != WARP_SIZE) break; // last iteration was computed.
    run = 1;
    mid = 0;
    x = 0;
}
```
The examples are:

- **Dependence of measurement stations on the environment pollution:**
  Particular implementation of symbolic regression was utilized for, e.g., the analysis
  of the dependence of measurement stations on the environment pollution [45]. In-
  formation on air quality plays a significant role in everyday life of each of us. In
  our region, we can find a number of measurement stations whose data helps us to
  monitor not only the concentration of individual constituents of pollution, but also
  directions of pollution. Dependence of measured values among individual stations
  is considered as a key analysis. The knowledge of basic functions of measurement
  stations could solve a problem of station substitutability in case of their outage.
  Figure 3.5 shows the results of the dependence model of a selected station TORZA
  and two stations TORVA and TTRKÅ.

- **Real-time analysis of EEG data:**
  We managed to publish many scientific results in this field [47]. We could use
  a few types of sensors to monitor EEG, from gaming devices produced by Emo-
tive company to professional medical devices produced by g.Tec company which offered a rank higher monitoring frequency, higher number of sensors and higher total accuracy of measurements. We tested a number of methods during whose implementation we utilized the above mentioned parallel approaches. Figure 3.6 shows the scheme of data processing with the use of GPU.

![g.Tec biosignal amplifier](image)

**Fig. 3.6:** The scheme of EEG signal analysis with the use of GPU.

- **Combinatory problems**
  As was mentioned above, combinatory problems seem to be a very interesting domain for the utilization of mass parallelism. To explain optimization at the level of Warp, we used scheduling problem, or precisely flow shop problem, which is described in detail in attached publication [9]. A part of our research in recent years is also focused on the domain of DNA data analysis with respect to a specific and individualized treatment of patients. In short, the aim is to search for the dependence of analytes in relation to the course of a disease of real patients. Combinations which could influence more accurate classification of patients and follow-up treatments or recommendations are analysed. In connection with this, we have already published several papers whose nature belongs to the field of medicine, and implementation of methods represents supportive part of experiments [132]. Figure 3.7 shows the examples of patients based on the best combination of three and two analytes. Isomaps show the probabilities of group classification.
Fig. 3.7: The example of classification of real patients on the basis of analyte combination [132].

The above described research direction was focused on the method implementations and their parallelism, this time purely from the perspective of technical solution and with the selected technology. Implemented methods found a number of applications in real data analysis.

Methods based on tree structures were predominantly used for the classification and regressive tasks. The field of application was the processing of signals which could be either of biological origin [141, 73, 115], or they were technical data [18, 48, 45, 76].

Some publications were based predominantly on general observation of efficiency of classifiers such as [114]. Methods based on optimization were focused rather on combinatorial tasks [9, 132] or tasks whose solution required more effective approach to multi-dimensional data [20, 47].

**Included papers are:**


3.3. Research direction III:
Parallel computing and visualization

The last chapter is devoted to a group of tasks which link parallel computing and visualization. If we separate the field of gaming industry where we rely basically on the performance and options of graphics cards, the link between visualization and computing is predominantly a domain of simulation computing. Example can be the application of Finite Element Method (FEM) for the computation of object deformation [11, 71], simulation of particle system in analyses of gas particle flow [1, 158] or fluid particle flow [30], or visualization of force fields [100]. To put it simply, it is the whole application field where it is reasonable to connect a particular result with a visual perception.

Current construction of GPU have enough computing cores and shader units for the construction of an image. The advantage of combined approach of graphics processing units is especially in Video Random Access Memory (VRAM) which is shared by the computing and visualization parts of an application. Unitary storages do not have to transfer data, which speeds up computing at the end. By integrating more cards we can overcome the biggest disadvantage which is small memory capacity in comparison with operation memory of control systems whose capacity is expressed in terabytes. This capacity is, in comparison with the memory of graphics cards, thousand times bigger. The important role is played by the design of the whole infrastructure which must allow fast transfer between RAM and VRAM.

Among possible tasks there are real-time visualization of graph structures and indexation of spatial data. These tasks belong to the domain of analysis of graphs, networks, or possibly to the field of basic analysis of data and algorithms of k-Nearest Neighbors (kNN) type, etc. Obtained results will be described in attached publications.

3.3.1 Interoperability between computing and visualization

Generally, the approach to the use of GPU dates back to 2000 when the so-called programmable shared units became a part of graphics cards. At that time, they were very limited in terms of the size of instruction set. In any case, they could easily execute vector and matrix operations, which was sufficient for many tasks. This issue involved General-purpose computing on graphics processing units (GPGPU)). Later, this concept was a basis for OpenGL and DirectX, and even though every API developed its own way with many distinguishing features, the principle of programmable pipelines survived and shader units became more important. Although current computer shaders allow to do a number of operations with data, we cannot consider neither OpenGL, or DirectX nor their modern successor Vulkan as computing API as NVIDIA CUDA [130, 139] or OpenCL [82, 131] and the others are.

The important role of computing APIs is played by the so-called interoperability which is represented by a set of methods and instructions that allow effective linking of computing API and visualization. Further in this paper, we will describe solved tasks using
NVIDIA CUDA and OpenGL. Selected domain will be this time visualization of graph structures where we managed to improve one well-known algorithm for graph layouting.

Let a graph \( G \leftarrow (V, E) \) be a set of vertices \( V \) and edges \( E \) that connects those vertices. One of the well-known algorithms for the graph layout strategies is the Fruchterman-Reingold [83, 105, 32], which belongs to the family of force-directed graph layout algorithms. Vertices connected by an edge attract each other. It also defines an ideal distance for each vertex. The vertices should be drawn near each other, but not too close. To lay out a graph, the vertices are replaced by steel ring and each edge with a spring mechanical system [32].

**Computing part**

Computing part of the whole issue lies in the actualization of a graph node on the basis of layout algorithm, in a search for neighbouring nodes and solutions to a number of related subtasks (order, clamp, application of forces according to a simplified physical model, etc.). In terms of parallelism, it is a linear task which can be easily described as (see Algorithm 3.5):

**Algorithm 3.5: Updating coordinates of nodes using Fruchterman-Reingold algorithm**

```plaintext
inputs: n ... the number of nodes
        coords ... coordinates of nodes in 2D/3D
output: Updated coordinates of nodes

1 while true do
2    Read current coordinates of nodes
3    Compute repulsive forces
4    Compute attractive forces
5    Update coordinates of nodes
6    Apply gravitation factor to prevent graph expansion
7    Apply displacement limits to prevent graph contraction
8    Compute \( \Delta \) on coordinates            /* previous vs current coordinates */
9    if \( \Delta > \text{limit} \) then
10       Map all required resources (textures, buffer objects)
11       Store updated coordinates of nodes for visualization
12       UnMap the resources
13    end if
14    Notify end to visualization thread
15 end while
```

**Visualization part**

The visualization part of algorithm runs in a separate thread, thus it is necessary to solve synchronization between the computing and visualization parts. Furthermore, visualiza-
3.3. Research direction III: Parallel computing and visualization

Parallel processing has a higher priority because it provides an output to the end-device (monitor), and this thread cannot be interrupted for a longer period because it would lead to the loss of an image at output or to the significant decrease of Frames Per Second (FPS). The aim of visualization is to grasp coordinates, apply shaders on them and create the image for the output device. Simplified procedure is described in Algorithm 3.6.

**Algorithm 3.6: Visualization of the graph structure**

| inputs: n | ... the number of nodes |
| coords | ... coordinates of nodes in 2D/3D |
| output: Rasterized image displayed on the output device. |

1. while true do
2. Bind all required resources (textures, buffer objects)
3. Read the nodes’ coordinates from resources
4. Apply a graphic pipeline (on-screen or off-screen rendering)
5. Unbind the resources Send result to the output device
6. Notify the computation thread to start a new iteration
7. end while

The above mentioned interoperability between both parts can be solved in a selected architecture and selected APIs in several ways:

1. Via the so-called buffer objects which are, in the case of OpenGL, Pixel Buffer Objects (PBO) for these purposes.

2. Via textures, which seems to be more complicated process nowadays because textures must be created in OpenGL, but they are registered as resources and subsequently mapped for a possible updating on the computing part.

It is necessary to point out that in both cases we speak about a process of writing into global memory of a graphics card, which brings a drawback in the form of high latency during writing. Within the whole process, the emphasis must be given to the synchronization of operations such as Map/Write/Unmap in the computing part, and Bind/Read/Unbind in the visualization part [130]. In terms of application of parallel approach, these are shared tools at the level of two application planes. In practice, there is a possibility that computing is provided by one or more cards. The visualization part is mostly provided by one card, but as for the so-called Deferred Rendering [139, 86] it is possible that a final image is created by more cards.

3.3.2 Application domains and achieved results

Over the recent years we have had many possibilities to test this approach in a number of tasks. Probably the first more significant achievement was the offscreen rendering of 3D scene with the follow-up visualization of floodplain within FLOREON+ project [54]. It was a unique design of the application enabling 3D visualization on website with regard to
hardware tools and capabilities of web browsers at that time. Applications that appeared later are:

- **Visualization of graph structures** We managed to improve the above mentioned Fruchterman-Reingold [49] algorithm and create the application that was capable of doing visualizations of graphs in tens of millions of nodes in real time (Figure 3.8). The innovation lied in the use of spatial curves to carry out the indexations of points whose coordinates were the basis for the computation of a hash code which defined the position of a point on a spatial curve. The computation of point neighbourhood is then approximated by the search of neighbours on the curve up to a certain distance. Possible inaccuracies which are brought by spatial curves are subsequently solved by using artificial noise, the so-called jittering, which randomly moves with points in a limited plane. This movement influences the position of a point in virtual grid, as well as the final computing of a hash on the spatial curve.

Fig. 3.8: The example of application output published in [49] which demonstrates the importance of jittering in the proposed method.
3.3. Research direction III:
Parallel computing and visualization

- **Searching subroutines in the point cloud:**
  Next publication is based on the similar approach. This publication focuses on task optimization by searching various points in the point cloud. This method was named as Multidimensional Discrete Differential Evolution (MDDE) and it combines the principles of Differential Evolution (DE) with indexation of spatial curves in multidimensional space. More information can be found in the attached publication [144]. Visualization played here only supervisory role, nevertheless the principle of interoperability between computing and visualization was maintained.

![Diagram](image-url)

Fig. 3.9: The random vertices (A) are ordered by the Hilbert curve (B). A mutant individual \( V \) is computed by the mutation operator from the three parent individuals (best, \( A \), \( B \)). The mutation is computed on the level of vertex indices assigned by the corresponding SFC. The vertex order defined by the SFC improves the spatial convergence of the evolution.

Apart from the above mentioned tasks, the issue of parallelism in connection with data visualization was solved even within other experiments such as [148, 145, 146, 147, 143]. With respect to a possible utilization of graph structure visualization in a number of applications, we will continue to focus on this issue in future.

**Included papers are:**


Chapter 3. Research directions in the context of parallelism on GPU
Summary

With respect to data analysis, this work summarized three important aspects of parallel programming which stand behind real implementations of algorithms, methods or experiments. The first described approach was the approach to the algorithm construction with respect to SIMD/SIMT architectures and their support of vector and matrix operations. Next part was focused especially on approaches which improve scalability of algorithms or methods, and at the same time they enable to fully utilize hardware tools, or more precisely graphics cards. The last aspect of parallelism utilization in data analysis was described in the chapter focusing on the link between computing and visualization.

The aim of this paper was not to offer an exhaustive overview of possibilities of parallel computing, which would not be even possible with respect to the extent of this topic, diversity of technologies or application domains. Basically, this paper introduced the research directions in which we managed to utilize particular technologies and find new solutions to theoretical and practical tasks. This paper also points out interesting application results which have been reached over the recent years. It is obvious, that before reaching satisfactory results at the application level, technologies have to be studied properly, programming skills have to be acquired and all theoretical backgrounds of solved tasks have to be mastered.

4.1 Future work

Utilization of parallel approaches to the problem solutions at the hardware and software levels has become a standard procedure nowadays. The development of technologies still opens new opportunities of searching for solutions of problems which are very difficult to grasp today. The example can be the development of quantum computers, memristor memories or chips which can change their form between a processor and a memory. This will mean a conceptual change of view on addressed issues in the development of algorithms and methods for data analysis, which opens the door to new ideas. And definitely, I want to be there on its doorstep.
ARTICLES IN JOURNALS WITH IMPACT FACTOR


**OTHER RECORDS INDEXED IN ISI**


**OTHER RECORDS INDEXED IN SCOPUS**


**OTHER RECORDS INDEXED IN DBLP**


Tension prediction by using ANN and SOM in heavy facilities

Petr Gajdos, Jan Platos
Department of Computer Science
VSB-Technical University of Ostrava
17. listopadu 15, 70833 Ostrava Poruba, Czech Republic
{petr.gajdos, jan.platos}@vsb.cz

Petr Fiala
VITKOVICE UAM a.s.,
Mezirka 775/1, 602 00 Brno, Czech Republic
petr.fiala@vitkovice.cz

Abstract—Diagnostic systems based on mathematical models of material damaging process can be used to collect necessary information on trends and/or level of material and function damage. This paper is focused on the improvement of a particular part of the power plant diagnostic system. It describes some alternatives based on Artificial Neural Networks and Self-Organizing Maps. Finally, this can help to eliminate the damages of power plant facilities.

Keywords—Neural Networks, FEM, SOM, soft-computing, tension prediction.

I. INTRODUCTION

Energetics, chemical and food industry, rolling/mills and many other industries uses large amount of facilities under very hard conditions. All facilities undergo many changes during their lifetimes. Their operators need information on gradual damage of materials of such facilities. Gathered information are used in process of control to reduce possible damage and to achieve optimal lifetime of individual facilities. For example, a large enough time period after which the facility’s owners are willing to pay necessary repairs and preventive services can play an important role in the lifetime of particular facility.

Diagnostic systems based on mathematical models of material damaging process can be used to collect necessary information on trends and/or level of material and function damage. One of the parameters consumed by this mathematical model is a time flow of relative deformation \( \epsilon \). Another one is a tension \( \sigma \) within selected construction points with maximal material damage caused by facility wear. The losses of facility integrity, fatigue of materials, corrosion, and creep fall into the set of most common and most dangerous faults.

Diagnostic systems can be used in on-line and off-line mode. Typically, an off-line diagnosis of measured data from stored files is performed once per day. This approach is used in such cases, when multiple working load cycles are performed during a day, e.g. rolling mill, derrick and giant machinery in open pit mines. The first mentioned on-line diagnostic approach can be used in such case when, e.g., a machine works with limited and rated power for many days, even more that one month. In this case, off-line analysis provides a good result only if the working load cycles are closed. Then the off-line analysis is performed repeatedly, e.g. every week from January, 1st of the current year till the evaluation day, and achieved results of defined periods usually update the results achieved during previous measurement.

The computation of the relative deformation \( \epsilon \) and tension \( \sigma \) within selected construction points can be done by using simplified, but precise enough approach based on analytical simulations. These simulations try to express a dependence among input parameters (\( \epsilon \), \( \sigma \)) and measured parameters of working load like a temperature \( T \), an overpressure \( p \) and a flow rate \( Q \) of running medium inside conduits, chambers and heat transfer tubes in energetics facilities.

Next, the computation of the relative deformation \( \epsilon \) and tension \( \sigma \) with respect to working load is varying in time and represents a dynamic system. This is more important especially in case of complex geometric construction shapes of a particular facility. Then the computation can be performed by using the Finite Element Method (FEM). Unfortunately, FEM is very time-consuming and cannot be effectively used in continuous computation of \( \epsilon \) and \( \sigma \) as a part of diagnostic system.

Because of very high time complexity of FEM, the more progressive approach may be used. The proposed method is based on two-step algorithm. In the first step, a time flow of relative deformation \( \epsilon \) and tension \( \sigma \) is computed by using FEM. These results represent an input set for the second step in the so called learning phase. Many soft-computing methods suitable for this problem exists, e.g. Artificial Neural Networks, Self Organized Maps, Flexible Neural Trees, Adaptive Neuro-Fuzzy Inference system, Fuzzy rules based system. The paper is primary focused on the utilization of Artificial Neural Networks and Self Organized Maps in the project FR-TI1/86 New design approach of energetic components and steel structures with high utility parameters. Prepared soft-computing models can be used as a part of on-line and off-line diagnostic systems, respectively. The main contribution of such utilization consists in an evaluation of remaining lifetimes of various facilities.

Examples of the time flow calculation for relative deformation \( \epsilon \) and tension \( \sigma \) in relation with typical working load changes \( T \), \( p \), \( Q \) using FEM is shown in the Figures 1 and 2. Both images show a connection of membrane wall tube...
and chamber body.

**Figure 1.** Temperature distribution $T$ in connection of membrane wall tube and chamber body

**Figure 2.** Tension intensity distribution $i$ in connection of membrane wall tube and chamber body

In this paper, neural networks and self-organized maps were used in solving defined problem. Neural networks were successfully used in prediction of interfacial tension at the crystal/liquid interface [1], cable tension cable-stayed bridge using BP neural network [2], computation of concrete breakout strength of single anchors in tension [3], determination of surface tension of pure compounds [4]. Self organized maps were used in prediction of oil temperature [5] and for prediction of ground water level [6]. This is the reason, why we select these two methods for solving defined problem. Rest of the paper is organized as follows. Section 2 contains description of the artificial neural networks and Section 3 contains description of the self organized maps. Section 4 describes experimental data, Section 5 reviews achieved results and last section an conclusion and description of future work.

## II. Artificial Neural Networks

Artificial Neural Networks (ANN) are an attempt at modeling the information processing capabilities of nervous systems [7]. Since their investigation, ANNs were used in many areas with very good results. They are very often used in pattern recognition, time-series prediction, data classification, data regression, etc. Very powerful attribute of the ANN is its ability to generalize their knowledge, e.g. ANN is able to compute function value in a point which was not known in a training phase.

One of the most common ANN is the Feed Forward Artificial Neural Network (FFANN). The FFANN contains several layers. The first layer contains only the input neurons - neurons which represents an input values. The last layer contains one or more output neurons which represents output value of the network. One or more hidden layers is placed between input and output layer. The hidden layers make all computations of the FFANN. The function of the neurons depends on the layers in which are placed. The example of the ANN is shown in the Figure 3.

**Figure 3.** Illustrative example of the ANN.

At the input of neuron in all but first layer are outputs of neurons in layers below the actual. These output are weighted. The function of the neuron is to calculate the weighted sum of all inputs. The weighted sum is then passed to the activation function which generates an output value. The most popular activation function is sigmoid function defined as $\frac{1}{1+e^{-\lambda x}}$, where $x$ is the weighted sum.

The principle of FFANN with one hidden layer may be defined as follows:

- Assign values of the input vectors to the input neurons.
- Use output values from the input neurons for computation of the output of neurons in first hidden layer.
Use the output of the neurons in hidden layer for computation of the output of neurons in the output layer.

The previous algorithm has one important disadvantage; the setting of the proper weights to the inter-neuron connections. The algorithm for setting of these weights is usually called a teaching or learning of the neural network. Many teaching algorithms exist, e.g. genetic algorithm, particle swarm optimization, etc. The most popular algorithm for FFANN is well known Back Propagation, which is a gradient descent based algorithm. This technique requires differentiable activation function. The basic principle consists in minimizing of measured error according to gradient descent algorithm and propagating such error from the output layer into hidden layer. More information on FFANN, Back Propagation and ANN may be found in [7].

III. SELF-ORGANIZED MAPS

Presented in the early 1980s by Kohonen [8], [9], the self-organizing map is one of the common approaches on how to represent and visualize data and how to map the original dimensionality and architecture of the input space onto another, usually lower dimensional, architecture in the output space. Further, the SOM can be used as a classification or clustering tool that can find clusters of input data which are more close to each other. All experiments and examples in this paper respect following specification of the SOM (see also the Figure 4):

- The SOM is initialized as a network of fixed topology. The variables $\text{dim}X$ and $\text{dim}Y$ are dimensions of such 2-dimensional topology.
- $V^m$ represents an m-dimensional input vector.
- $W^m$ represents an m-dimensional weight vector.
- The number of neurons is defined as $N = \text{dim}X \times \text{dim}Y$ and every neuron $n \in \mathbb{N}, N - 1 >$ has its weight vector $W^m_n$.
- The neighborhood radius $r$ is initialized to the value $\min(\text{dim}X, \text{dim}Y)/2$ and will be systematically reduced to a unit distance.
- All weights vectors are updated after particular input vector is processed.
- The number of epochs $e$ is know at the beginning.

The Kohonen algorithm is defined as follows:

1) Network initialization
   All weights are preset to a random or pre-calculated value. The learning factor $\eta, 0 < \eta < 1$, which determines the speed of weight adaptation is set to a value slightly less than 1 and monotonically decreases to zero during learning process. So the weight adaptation is fastest in the beginning, being quite slow in the end.

2) Learning of input vector
   Introduce $k$ training input vectors $V_1, V_2, \ldots, V_k$, which are introduced in random order.

3) Distance computation
   An neighborhood is defined around each neuron whose weights are going to change, if the neuron is selected in competition. Size, shape and the degree of influence of the neighborhood are parameters of the network and the last two decrease during the learning algorithm.

4) Choice of closest neuron
   We select the closest neuron for introduced input.

5) Weight adjustment
   The weights of closest neuron and its neighborhood will be adapted as follows:
   $$W_{ij}(t + 1) = W_{ij}(t) + \eta(t) dh(v, t)(V_i - W_{ij}(t)),$$
   where $i = 1, 2, \ldots, \text{dim}X$ a $j = 1, 2, \ldots, \text{dim}Y$ and the radius $r$ of neuron’s local neighborhood is determined by adaptation function $h(v)$.

6) Go back to point 2 until the number of epochs $e$ is reached.

IV. TESTING DATA

The data for all mentioned experiments were extracted from the real measured data. Following input parameters were used - temperature, overpressure and flow rate. The data set is stored in real-time when a large enough data change occurred. Therefore, the parameters are not store in the same time period. As a consequence, the first step consists in data preprocessing which scales data into same time intervals. This was made by linear interpolation of all values into defined time grid. The density of the grid defines the number of input values. Some non-linear grid can be used to better reflect the nature of data.

The main goal is to approximate the tension during several states of the power plant generator. The first state is called start state, when the generator started after it was off for long time. At the end of the start state, the generator runs on int maximal power; this is called full state. When generator is
off for less than 1 hour, the state is called very hot state. A hot state means, that the generator is off for less than 3 hours and the warm state means that the generator was off for less than 12 hours. The cold state is a final state. In that state, the generator was off for more than 12 hours and must be started again. Normal sequence of the states is given by following states:

1) start-state
2) several very hot
3) hot
4) warm states interleaved by full-state on long time duration
5) cold state

Between the start state and following phases, the generator runs on the full power without any changes for a long time and, therefore, no data is measured and stored into data files. The data interpolation during this time period will lead into long sequences of constant values, which are not needed during model learning. This is the reason, why a non-linear time grid should be used.

In our experiments, a long sequence of data was used. It contains start state, very hot state, hot state, warm state and cold state. Between the start and the cold state and other states was always a full-state. The testing data collection consists of 808 records. Every record is represented by a vector of temperature, overpressure and flow rate. The testing data can be seen in the Figures 5 (temperature), 6 (overpressure), and 7 (flow rate).

As can be seen, the full-states are not visible, because the non-linear time grid was chosen to eliminate unnecessary values for learning phase. Presented data was used in the learning phase of the ANN and SOM algorithm. Because we need to compare our model with a more sophisticated approach, the tension was computed by using FEM as well. Such calculation of satisfactory precision takes usually several hours using.

V. EXPERIMENTAL RESULTS

A. Artificial Neural Networks

In the first experiment, the Artificial Neural Networks was used to for calculate tensions from input data set. A standard Feed-forward artificial neural network in combination with Back-Propagation algorithm was implemented. The learning rate was set to 0.3, momentum was set to 0.2 and a single hidden layer with five and 30 neurons was used. In case of five neurons in the hidden layer, the learning algorithm run in 500 epochs. The experiment with 30 neurons in the hidden layer applies 2000 epochs in learning phase which results in more accurate values for every single neuron. The results can be seen in the Figures 8 and 9.

![Figure 8](image_url1)

Figure 8. Tension computed by using ANN in comparison with FEM for 5 neurons in hidden layer and 500 epochs.

![Figure 9](image_url2)

Figure 9. Tension computed by using ANN in comparison with FEM for 30 neurons in hidden layer and 2000 epochs.

As can be seen in the figures, the computed values of both experiments with different algorithm setting lead to interesting result. The results for the first variant of settings of ANN algorithm are worse than for the second variant, but still the major trends in the final graph correspond each other. The minor changes of values are not followed so precisely. Moreover, the tension in very hot state is much lower than the tension computed by FEM. Better results are depicted in the figure 9. Again, some minimal fluctuation is ignored like that in the start state. Some other fluctuations leads to the different fluctuation in tension computed by the ANN.
As a next step, a feedback signal was used for computation of ANN model. The results are depicted in Figure 10. This signal correspond to the previous value of the tension. In the first step, this signal is undefined, therefore, the algorithm skips the first record in the data set. As may be seen, the achieved results are much better than in case of using previous variants, event when there were just 5 neurons in the hidden layer and 500 iterations. All changes are followed without problems with very good precision.

**B. Self-Organising Maps**

As well as in the previous experiments, the Self-Organizing Maps (SOM) should bring the faster response in the process of computation of tensions from input data set. This time, the experiments differs in the settings of SOM itself. The layer of neurons can be preset as well as the number of epochs. At the beginning, a grid of the SOM was set to $8 \times 8$, which means, that there was 64 neurons that must be learned to achieve their best distribution over solution space. It was done in 500, 1000, and 1500 epochs. The corresponding result are illustrated in the Figures 11, 12, and 13.

The setting $8 \times 8$ of SOM grid leads to some unexpected results which can be seen as individual bounce in resulting values. This is due to small grid and/or large variance of input data set. On the other side, the computation take approximately two seconds in comparison of FEM method. A large grid was used to eliminate this drawbacks. The second set of experiments used the grid of $16 \times 16$ neurons. The Figures 14, 15, and 16 illustrate the results which approximate the FEM data with minimal offsets.

The large grid and higher number of epochs lead to better results. Mentioned experiments have also their own limits, e.g. the size of grid should not be greater than the number of input vectors in the data sets. This could never happen in the real system operation, because the process of data retrieval runs in real-time and the input set can contain tens or hundreds of thousands records.
VI. CONCLUSION AND FUTURE WORKS

The paper describes an alternative approaches to Finite Element Method in the process of power plant diagnostic system primary focused on tension prediction. Proposed methods were based on Artificial Neural Networks and Self-Organising Maps. Although, there were some small errors in computed data in comparison with measured data set, the final approximation bring interesting results. Moreover, the proposed methods are much less time-consuming, therefore they can be used in real-time. In the future, we want to evaluate more experiments and develop a new complex tool for diagnostic system based on soft-computing methods.

ACKNOWLEDGMENT

This work was supported by the Ministry of Industry and Trade of the Czech Republic, under the grant no. FR-T11/086.

REFERENCES


Self Organising Maps on Compute Unified Device Architecture for the Performance Monitoring of Emergency Call-Taking Centre

Václav Snášel¹, Petr Klement², Petr Gajdos¹, and Ajith Abraham¹,³

¹ IT4Innovations, Department of Computer Science, VŠB-Technical University of Ostrava, 708 33 Ostrava-Poruba, Czech Republic
{vaclav.snasel,petr.gajdos}@vsb.cz
² Medium Soft a.s., Cihelní 14, 702 00 Ostrava, Czech Republic
petr.klement@mediumsoft.cz
³ Machine Intelligence Research Labs (MIR Labs), Scientific Network for Innovation and Research Excellence, WA, USA
ajith.abraham@ieee.org

Abstract. The collaborative emergency call-taking information system in the Czech Republic forms a network of cooperating emergency call centers processing emergency calls to the European 112 emergency number. Large amounts of various incident records are stored in the databases. The data can be used for mining spatial and temporal anomalies, as well as for the monitoring and analysis of the performance of the emergency call-taking system. In this paper, we describe a method for knowledge discovery and visualization targeted at the performance analysis of the system with respect to the organization of the emergency call-taking information system and its data characteristics. The method is based on the Kohonen Self-Organising Map (SOM) algorithm and its extension, the Growing Grid algorithm. To handle the massive data, the growing grid algorithm is implemented in a parallel environment using compute unified device architecture. Experimental results illustrate that the proposed method is very efficient.

Keywords: Emergency Call, Self-Organising Map, Growing Grid, Knowledge Discovery in Databases.

1 Introduction

Emergency call taking in the Czech Republic, is supported by a distributed collaborative information system operated at fourteen regional emergency call centres, or PSAPs (Public Safety Answering Points). Each of these PSAPs serves emergency calls from its home region primarily. If the home PSAP is occupied or out of order, the system automatically reroutes emergency calls to another PSAP, where the call is processed in the same way as it would be processed by the PSAP of the home region. Every single operator knows the actual operational status and language skills of all the other operators logged into the
system. Thanks to the cooperative functionality based on an instant messaging subsystem, which is transparent to the user, operators can ask for help or offer their free capacity and skills in conference mode to the other operators. As all the descriptive and operational data are shared or replicated between system nodes in the background, any operator can receive an emergency call from any region, regardless of his/her position with respect to the location of the incident.

Experience from the operation of the emergency call-taking system suggests that in a normal situation calls are smoothly processed by operators in the region where the incident originated, without any special demands on the system settings. In highly critical situations, when many incidents happen in a short period of time (e.g. during storms or floods), or many people are announcing the same incident (e.g. a plane crash, gas explosion, or large fire), the system could be locally overloaded. In this case some intelligent reconfiguring scheme would help to balance the system load with respect to the resources available. By means of routing schemes via which calls are distributed, the quality of the service affecting the network throughput and the prioritisation of critical services, as well as postponing the replication of less important data to lower network traffic in overloaded regions, can be managed dynamically, with the goal of improving system responses in critical situations.

In order to be able to apply proper and timely management actions, the system must first recognise the critical or anomaly situation. There is a central database, containing records of emergencies, or incidents, from the whole territory of the Czech Republic. This database can be used for the monitoring and analysis of the current situation with respect to the system and operators’ performance, as well as for learning from historical incidents and mining spatial and temporal anomalies.

This paper presents an enhancement of the previous work described in [14,15]. Building on their experience with the detection of anomalies, the authors focused on the cooperative characteristics of the emergency call-taking system in order to show a non-traditional approach to the monitoring and analysis of the system’s performance. The authors describe the detection of an anomaly situation from emergency data by unsupervised machine learning, namely the application of the Kohonen Self-Organising Map (SOM) algorithm. Measures taken after the critical situation is detected are supposed to be applied outside this analytic process and are not discussed here.

This article is organized as follows. In Section 2, we compare our approach with other related works. Section 3 describes the principles of the SOM algorithm used for clustering and Section 4 deals with features of the software tools. Section 5 illustrates the basic experimental results with data transformations applied to the training set to form a suitable search space and achieve satisfactory outputs. This experiment is focused on the nature of the incident in order to search for similarities in the type and place of the incident, bearing in mind the proof of concept in general. Section 6 depicts a set of experiments with the classical and Growing Grid SOM applied to the incident report attributes, describing rather the architecture and technology in the system background. In this way we can
reveal global or temporal weaknesses in the emergency call-taking system as a whole to improve its performance. In Section 8 we discuss the results followed by conclusions in Section 9.

2 Related Works

Anomaly detection methods for various professional domains have been designed and well described. SOM has been used for network intrusion detection [18,22], fraud detection [3], mechanical fault detection [26], and anomaly detection in the generic time series data [9] etc. A common approach in using SOM for anomaly detection is to build a classifier distinguishing between anomalous and non-anomalous classes of data. The non-anomalous data are used to create a model of the correct situation. After that, a single input vector is presented to the trained SOM and the winner neuron closest to the input vector is found. If the distance from the winner’s representative is below a certain limit, the input is classified as belonging to the winner’s cluster of non-anomalous data. Otherwise, the input is considered anomalous. These classification methods assume that either a non-anomalous subspace is known before the learning starts [22] or the resulting clusters are compared with an expert classification [3]. Generating artificial anomalous cases with a Negative Selection Algorithm inspired by the human immune system [2,5] in combination with a back-propagation neural network [9] falls into this category. Thus the detection of anomalies in this traditional view is based on supervised learning.

Our approach is based on two facts. First, distinguishing between anomalous and non-anomalous cases within emergency calls is disputable. Second, if an emergency situation exceeded the “normal” scale, it would probably be reported by a set of single emergency cases having certain attributes in common. We are therefore interested in revealing certain patterns in the emergency call data, rather than deciding whether a fresh new case is somehow strange compared to the previous experience. After the patterns are recognised automatically, the resulting map is always presented to a person for them to analyse the situation.

To enhance this concept, providing that the algorithm is being run periodically, the new map is shown to the supervisor or to the network management module if the composition of the patterns found in the recent run is different from the composition formed in the previous run. SOM quality measures include an index of the map’s disorder [20] or the goodness of the map based on the distances between the winner and the second-best match node [13]. While the SOM algorithm has been widely used for classification tasks, using it for clustering data analysis has been relatively outside the focus of the research community [25]. This paper describes another application of SOM in cluster analysis. Some authors analysed [21] the performance of a business-oriented call centre with respect to its operator’s activity, efficiency, and ability to meet client and business needs. In this paper, we focus on the performance of the Emergency Call Centre (ECC) mainly from the technological point of view, although we also attempted to address the operator’s activity analysis.
3 Clustering and Self-organizing Maps

Cluster analysis groups objects (data records) into classes (clusters) in such a way that objects in the same cluster are very similar, while objects in different classes are quite distinct. One of the possible clustering methods is using competitive learning [8]. Given the training set of objects, competitive learning finds an artificial object (representative) most similar to the objects of a certain cluster.

A commonly used application of competitive learning is the Kohonen Self-Organising Map [16], or SOM, described by Teuvo Kohonen in 1982. SOM is inspired by the cortex of the human brain, where information is represented in structures of 2D or 3D grids. Formally, SOM is a type of artificial neural network [11] with two fully interconnected layers of neurons, the input layer and the output or Kohonen layer.

The first step of Kohonen learning is competition. Given the training vector on the network’s input and weight vector for each neuron of the Kohonen layer, the neuron with the minimal (usually Euclidean) distance between the weight and input vectors is excited or selected as the winner of the competition [11].

The second step is adaptation. The neurons of the Kohonen layer are organised in a one-, two-, or three- dimensional lattice, reflecting its biological inspiration. A topological neighbour-affecting function is defined on the Kohonen layer, assigning a degree of participation in the learning process to the neurons neighbouring the winning neuron. In every learning step the weight vectors of the winning neuron and its neighbours are adjusted to move closer to the input training vector.

In the batch version of the SOM algorithm, equivalent to Lloyd’s vector quantisation [19], the winning neuron weights are not adapted immediately after the competition step. When all the training set is consumed, the weight vector of the output neuron $N_i, i = 1, \ldots, n$, where $n$ is the number of neurons in the network, is replaced by the weighted mean value of the training cases assigned to the clusters represented by the neuron $N_i$ and its neighbours, using the neighbour-affecting function as the weight function for the mean calculation.

The trained network finally sets its weights in such a way that the topologically near neurons represent similar training cases, while distant ones reflect different cases. This is analogous with the cortex of the human brain, where similar knowledge is represented by adjacent parts of the cortex. The topology of a trained SOM forms an inherently useful base for clustering.

SOM realises the transformation of the relations of the objects from the $m$-dimensional input space in a two-dimensional map of nodes (neurons) of the resulting Kohonen network. The complexity of the input space is reduced significantly and, in conjunction with colouring the nodes of the resulting network, data clusters can be effectively visualised.

4 Characteristics of the Software Tools

The authors developed a SOM learning module and a corresponding SOM explorer. The tools were also used in the framework of another project [1]. In the
course of this work the SOM explorer has been enhanced by the capability of revealing nodes according to the values of the selected attribute and a Growing Grid has been implemented in the SOM learning algorithm. The learning module exploits a classical SOM learning algorithm, with the neighbourhood radius initially set to 50% of the largest SOM dimension and the exponential radius shrinking.

Some of the commercially available products have the capability to build clusters automatically from the trained SOM, using, for example, the agglomerative clustering over the SOM-Ward distance [24]. Our learning module, however, generates a SOM density map, which shows for every node the mean distance to its neighbouring nodes. The lines formed by nodes having relatively distant neighbours can be taken as cluster borders.

The Growing Grid algorithm [7], compared to the classical fixed-grid SOM, allows the input data to influence the size and shape of the resulting SOM grid. When the entropy of the input data set is low, the growing SOM needs fewer nodes to approximate the input space, the algorithm is much faster, and the resulting SOM is more consistent in terms of having fewer empty nodes. Another advantage of the growing SOM lies in its capability to partially adapt its shape. When the input data have dominant variance in a certain direction, one of the dimensions of the resulting maps extends along this direction. This feature can be amplified by setting the weights of the initial SOM nodes to values evenly distributed between the maximal and minimal values of the attributes with the $k$-largest variances (for $k$-dimensional SOM), identifying them after the $k$-largest eigenvalues of the input data set covariance matrix [12]. Even if this adaptability is not as powerful as in the growing neural gas [6], for example, we consider it useful with respect to emergency data containing spatial information. In our experiments incidents belonging to adjacent geographic areas proved to occupy nodes in neighbouring clusters, thus contributing to the logical composition of the map.

The Growing Grid algorithm starts with an initial size of $2 \times 2$ nodes. After every learning phase, the node $N$ with the highest number of input records assigned [7] or with the largest quantisation error [4] is found. Within the closest neighbours of the node $N$ the node $D$ is selected in such a way that the distance $d(N,D)$ is maximal with respect to the distance metric used in the algorithm. In our experiments the commonly used Euclidean distance [8] was exploited. Between nodes $N$ and $D$, depending on the mutual position of the nodes, either a row or column of new nodes is added. The new nodes are initialised with the mean values of the weight vectors of the adjacent original nodes and the new learning phase starts on the enlarged SOM. The original nodes keep their weights from the previous iteration, and thus the weights of the new nodes fit in between them smoothly. In this way the training vectors are distributed roughly between a few nodes in the first learning phases and the distribution is further refined in the following phases on the maps, which grow in the direction where the probability of refinement is high, and the relationship between the records distributed in the previous iterations is preserved.
The iterative growth-and-learning process can be finished either after a fixed number of iterations or a predefined number of nodes is reached, or when a condition characterising the precision of the map learning is met. We calculated the vector \( \bar{x} \) of the training set’s attributes means and a training set variance \( \delta \) as the sum of the distances of the training vectors \( x_k \) from the vector \( \bar{x} \):

\[
\delta = \sum_{k=1}^{n} ||x_k - \bar{x}||
\]

(1)

where \( n \) is the number of input vectors in the training set. We evaluated the absolute quantisation error \( q \) of the whole map after every learning phase:

\[
\delta = \sum_{i=1}^{m} \frac{1}{c_i} \left( \sum_{j=1}^{c_i} ||x_j - w_i|| \right)
\]

(2)

where \( m \) is the number of nodes in the current iteration map, \( c_i \) is the number of vectors assigned to the node \( i \) in the previous training phase, \( x_j, j = 1, \ldots, c_i \) are the vectors assigned to the node \( i \) in the previous training phase, and \( w_i \) is the weight vector of the node \( i \). The process stops when the expression (3) is encountered.

\[
q < \frac{\delta}{\varphi}
\]

(3)

where \( \varphi \) was the input parameter defining how many times lower than the initial training set variance the learning precision had to be to stop the algorithm. In expression (3) we compare the factor of inaccuracy in the trained map defined by the quantisation error \( q \) of the expression (2) with the initial variance of data defined by the expression (3). The algorithm stops when the learning inaccuracy factor is satisfactorily lower than a certain fraction of the initial variance of data, defined by the parameter \( \varphi \). For \( \varphi \) we used values between 10 and 100, the lower ones resulting in a shorter run time but a final map of poorer quality, consisting of fewer nodes and vice versa with the higher \( \varphi \) values. In comparison with the fixed number of iterations [4] or the maximum number of nodes [7] the stopping criterion (3) used in our algorithm adapts to the original data distribution, and allows better tuning of the algorithm with respect to the run time and accuracy of the final map.

To be able to experiment with the algorithm speed, we implemented a heuristic consisting of the count of variable epochs per learning phase. We increased the epoch count in each new learning phase, according to:

\[
\epsilon_t = \epsilon_{max} \frac{1 - q_{t-1}}{q_1}
\]

(4)

where \( \epsilon_t \) is the epoch count in a learning phase \( t \), \( \epsilon_{max} \) is the input parameter defining the maximal epoch count, \( q_{t-1} \) is the map quantisation error after the learning phase \( t - 1 \), and \( q_1 \) is the quantisation error after the first learning phase. In this way the initial iterations, when the data were roughly separated,
had a lower epoch count, while the final iterations, when the SOM was to be fine-tuned, reached the maximal epoch count.

Expression (4) is a natural mathematical formulation of the intention of having a lower epoch count at the beginning of the learning of the map and reaching the maximal epoch count towards the end of the learning process to fine-tune the map.

As the weight vectors of the nodes of the map are quite distant from the training vectors and the number of nodes is small at the beginning of the training, fewer learning epochs are needed to roughly organize the input space before the map grows and the next iteration starts. At the end of the training process, when the quality of the map learning becomes high (quantisation error gets low values) and the map growing is no more needed, the epoch count should reach its maximal value to find the best data distribution with respect to the actual topology of the map.

Although the epoch count, thanks to the stochastic nature of the SOM learning and corresponding quantisation error calculation, does not change monotonically, the function converges in both increasing and decreasing modes, speeding up the learning.

We used the absolute quantisation error in evaluating the learning quality as it implied better results with the variable epoch count heuristic, compared to the mean quantisation error proposed in the literature [4].

5 General Test of SOM Emergency Data

We aim to show that the SOM can be successfully used to reveal anomalies in the emergency incidents database in general. Incident records from the period February 1st March 31st 2008 were used for the experiments. On March 1st 2008 the territory of the Czech Republic was affected by Hurricane Emma.

The input data set consisted of about 25,000 records. The SOM-based procedure was supposed to find records related to Hurricane Emma (Emma records), depicting the Emma cluster formed by nodes representing the Emma records characterised by:

- specific types in the incident classification (storm, danger status removal, obstacle removal);
- a higher frequency of incidents, namely of the above-stated types, in the time period in question;
- a higher frequency of incidents in certain regions (districts).

The training vectors presented to the SOM algorithm consisted of the attributes “time of the incident beginning”, “incident classification”, “region of origin of the incident”, and artificial attributes derived from these primary ones.

As the incident classification and region are attributes of a categorical type, they are not suitable for the conventional SOM learning algorithm [14,15].

Therefore we were looking for a transformation of the categorical attributes into the real numbers domain, which would have preserved the distribution
characteristics of the original attribute values. We introduced frequency characteristics in a time quantum, the hour of the incident origin, as the basis for transformation.

The categorical attributes Classification and District were used as the source for artificial numeric attributes produced by a function of relevance, combining the classification or district frequency within the time quantum of origin of the current incident and the occurrence of the same classification or district in the remaining quanta (adopted from the text classification TF-IDF weighing model [23]).

\[
IMP_{CL} = \text{FREQ}_{CL_H} \log \frac{N_h}{CL_{HRS}}
\]

\[
IMP_{DS} = \text{FREQ}_{DS_H} \log \frac{N_h}{DS_{HRS}}
\]

(5) (6)

where \(N_h\) means the number of hours in the period analysed, \(FREQ_{CL_H}\) is the frequency of the current incident classification within the subset of incidents related to the hour of origin of the current incident, \(FREQ_{DS_H}\) denotes the frequency of the current incident district within the subset of incidents related to the hour of origin of the current incident, and \(CL_{HRS}\) or \(DS_{HRS}\) denote the number of hours in which the classification or district emerged.

Another attribute that was transformed was the district relevance.

\[
IMP_{CL_DS} = \text{FREQ}_{CL_DS} \log \frac{N_d}{CL_{DS}}
\]

(7)

where \(N_d\) means the number of districts in the Czech Republic, \(FREQ_{CL_DS}\) is the frequency of the current incident classification in the current incident district during the whole period analysed, and \(CL_{DS}\) denotes the number of districts where the current incident classification occurred.

The training set with transformed artificial attributes was presented to a static SOM network of 40 × 40 nodes. Figure 1 shows the final output after 100 training cycles.

Using the node analysis functionality of the SOM explorer, we were able to derive some knowledge from the trained map, as shown in Figure 2.

The results of this basic experiment illustrate that the SOM technique succeeds with anomaly detection in the emergency call-taking system database. In the following section we will further elaborate on this conclusion with respect to the monitoring of the performance of the emergency call-taking system.

6 Performance Monitoring of the Emergency Call-Taking System

The performance monitoring relies on attributes describing the architecture and technology in the system background, rather than the nature of the incident as seen in the previous experiment. A brief overview of the architecture of the emergency call-taking system is provided in Figure 3.
Fig. 1. Shadowed nodes contain Emma records; the Emma cluster is well bordered by the wavy line in the bottom left-hand corner of the SOM density map (bottom middle). The rainbow-like stripes within the Emma cluster, visible in the IMP_DS map (top right), point to the movement of Emma across the territory of particular districts. The anomalies not related to Emma are concentrated in the upper right-hand corner of the IMP_CL_DS map (bottom left), collecting specific incidents assisted by the Prague Municipal Police, pointing to a higher rate of car thefts in Prague and system tests being performed regularly at the particular PSAP.
Fig. 2. The output of the analysis of the nodes in one of the rainbow-like stripes from Figure 1 states that on March 1\textsuperscript{st}, from 11 to 12 a.m., 90 Emma-related incidents of types 3331 (windstorm), 3501 (removing dangerous objects) and 3526 (removing obstacles) were reported from two districts in the central part of the Czech Republic.
Fig. 3. Example of the architecture of the emergency call-taking system reduced to two communicating Emergency Call Centres (ECC) with related emergency response agencies’ proxy services and the central database service.
In a normal situation emergency calls are processed by operators in the region, where the related incident originated, e.g. in Region A. Once the call is processed by an operator-agent of the Emergency Call Centre (ECC), the incident report is stored in the database, which is a remote service to most of the centres, accessible through a Wide Area Network (WAN). A delay in the operation of the database may suggest a network overload or a DB service capacity problem. In parallel with the incident report being inserted into the DB, it is passed to the Emergency Response Agencies (ERA) via the respective proxy service. Fire and Medical Rescue services are accessible in the local network with respect to the receiving ECC, while the police provide one central proxy accessible via WAN for all but one ECC. The proxy services, after passing the incident report to the ERA, expect a confirmation message. Delays in receiving this confirmation may give evidence of the ERA systems being overloaded. In the telephone subsystem we can measure the ringing time, i.e. the time interval between the moment when the call arrives at the agent and the moment when the operator picked up their phone. As picking up the phone is automatic (auto-answer mode is preset in the whole system), stretching of the ringing interval may suggest congestion in the telephone system. Moreover, two values characterising the operator’s working style are measured, the length of the call and the time interval between the start of the call and the insertion of the incident report into the DB.

In critical situations, when many incidents happen in a short period of time, or many people are announcing the same incident and the operators of ECC A are busy, calls are routed to and processed at the ECC of another region (say B). In the course of processing the calls the DB and telephone call measures have the same meaning as in the previous case. But after the call is processed by ECC B, the reports to the ERA are communicated via the WAN to the ECC of the original region, A. The further incident report flow is identical to the previous case. The time it takes to transfer the incident report between ECC B and the proxy services of ECC A is measured and evaluated.

For the performance monitoring experiments we used data from a special testing of the emergency call-taking system under peak load. In an hour the system in its full capacity (14 ECC, 70 operators) received 3400 emergency calls and produced around 10,000 incident reports. As there was only one incident type reported, the “system test”, it made no sense to take the incident classification into account. Table 1 shows the attributes used in our experiments.

### 6.1 Comparison of SOM Alternatives

At the beginning we show the results of the traditional fixed SOM algorithm compared to its Growing Grid modification with the heuristic of a variable epoch count. We used a fixed epoch count, as well as increasing the epoch count according to the formula (4). The epoch count, node count, and the grid size produced by the growing SOM algorithm were used as input parameters for the fixed SOM which was to be compared. The results are shown in Table 2.

Empirical results illustrate that the Growing Grid SOM is faster than the respective fixed SOM algorithm, achieving a slightly worse learning quality.
Table 1. Description of attributes used in the performance monitoring. The categorical attributes were replaced by natural numbers to suit the SOM algorithm.

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ID_ECC_REC</td>
<td>Categorical</td>
<td>ID of the receiving ECC (natural number)</td>
</tr>
<tr>
<td>X_ECC_REC, Y_ECC_REC</td>
<td>Numeric</td>
<td>Geo coordinates of centroid of the region which the reported incident belongs to</td>
</tr>
<tr>
<td>OFFHOOK_DLY</td>
<td>Numeric</td>
<td>Off-hook interval, time delay between the ringing and call start moments</td>
</tr>
<tr>
<td>CALL_LENGTH</td>
<td>Numeric</td>
<td>Call length</td>
</tr>
<tr>
<td>ID Era</td>
<td>Categorical</td>
<td>ID of the Emergency Response Agency (ERA) which is to be alerted (natural number)</td>
</tr>
<tr>
<td>ID_WS_REC</td>
<td>Categorical</td>
<td>ID of the agent responding to the call (natural number)</td>
</tr>
<tr>
<td>ID_SRV_CENTRE</td>
<td>Categorical</td>
<td>ID of the proxy service responsible for communication with the particular ERA (natural number)</td>
</tr>
<tr>
<td>MINING_TIME</td>
<td>Numeric</td>
<td>Time from the start of the call to the moment of saving the incident record</td>
</tr>
<tr>
<td>INSERT_DLY</td>
<td>Numeric</td>
<td>Database service response time</td>
</tr>
<tr>
<td>DELAY_ECC</td>
<td>Numeric</td>
<td>Time delay of the transfer of the incident in the ECC technology</td>
</tr>
<tr>
<td>DELAY_CAD</td>
<td>Numeric</td>
<td>ERA technology response time</td>
</tr>
<tr>
<td>ID_ECC_RESP</td>
<td>Categorical</td>
<td>ID of the target ECC which the incident is transferred to (natural number)</td>
</tr>
</tbody>
</table>

Table 2. Comparison of the fixed and growing SOM alternatives. The “Epochs mode” denotes an increasing (Inc) or constant (Const) epoch count, and the “Candidate selection” denotes the way the central node for growing is selected, using either the quantisation error (Qerr) or the number of mapped input records (Match) of the node. The Euclidean distance was used in all the cases presented here. The tests were run on a PC with a 3-GHz Pentium 4 CPU and 2 GB of RAM.

<table>
<thead>
<tr>
<th>ID SOM type</th>
<th>Epochs</th>
<th>ϕ</th>
<th>ϵmax</th>
<th>Candidate selection</th>
<th>Epoch Node</th>
<th>Grid size</th>
<th>Run time</th>
<th>Final Map MQE</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Growing</td>
<td>10</td>
<td>50</td>
<td>Qerr</td>
<td>571</td>
<td>595</td>
<td>17 × 35</td>
<td>0:08:51 0.0482</td>
</tr>
<tr>
<td>2</td>
<td>Fixed</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>571</td>
<td>595</td>
<td>17 × 35</td>
<td>0:13:28 0.0333</td>
</tr>
<tr>
<td>3</td>
<td>Growing Const</td>
<td>10</td>
<td>50</td>
<td>Qerr</td>
<td>1850</td>
<td>384</td>
<td>16 × 24</td>
<td>0:09:33 0.0548</td>
</tr>
<tr>
<td>4</td>
<td>Fixed</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>1850</td>
<td>384</td>
<td>16 × 24</td>
<td>0:31:30 0.0446</td>
</tr>
<tr>
<td>5</td>
<td>Growing Inc</td>
<td>10</td>
<td>50</td>
<td>Match</td>
<td>320</td>
<td>468</td>
<td>26 × 18</td>
<td>0:05:02 0.0522</td>
</tr>
<tr>
<td>6</td>
<td>Fixed</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>320</td>
<td>468</td>
<td>26 × 18</td>
<td>0:08:20 0.0397</td>
</tr>
<tr>
<td>7</td>
<td>Growing Const</td>
<td>10</td>
<td>50</td>
<td>Match</td>
<td>1750</td>
<td>352</td>
<td>16 × 22</td>
<td>0:08:33 0.057</td>
</tr>
<tr>
<td>8</td>
<td>Fixed</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>1750</td>
<td>352</td>
<td>16 × 22</td>
<td>0:23:03 0.0482</td>
</tr>
<tr>
<td>9</td>
<td>Growing Inc</td>
<td>25</td>
<td>20</td>
<td>Qerr</td>
<td>797</td>
<td>1517</td>
<td>37 × 41</td>
<td>0:31:21 0.0256</td>
</tr>
<tr>
<td>10</td>
<td>Fixed</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>797</td>
<td>1517</td>
<td>37 × 41</td>
<td>1:13:46 0.0145</td>
</tr>
<tr>
<td>11</td>
<td>Growing Const</td>
<td>25</td>
<td>20</td>
<td>Qerr</td>
<td>1640</td>
<td>1764</td>
<td>36 × 49</td>
<td>0:50:04 0.0254</td>
</tr>
<tr>
<td>12</td>
<td>Fixed</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>1640</td>
<td>1764</td>
<td>36 × 49</td>
<td>2:21:29 0.0112</td>
</tr>
</tbody>
</table>

Self Organizing Maps on Compute Unified Device Architecture 351
The differences between the mean quantisation errors of the respective growing and fixed SOM represent a max. 3% of the initial variance of the training vectors. The selection of the central node for map growth according to the maximal number of mapped input records produces smaller maps, resulting again in faster and slightly less precise learning. The higher the $\varphi$ parameter in (3), the larger and more precise, but also more time-consuming, the maps produced are. Figure 4 displays comparable maps produced by Tests 9 and 10 from Table 2.

**Fig. 4.** Maps produced by the fixed SOM (Test 10 in Table 2) in the upper row, and growing SOM (Test 9 of Table 2) in the lower row. The growing SOM maps appear to be more coherent: emergency centres separated by the $ID\_ECC\_REC$ attribute are close to each other according to the geographical adjacency of their regions; workplaces inside emergency centres are pointed out in the $ID\_WS\_REC$ growing SOM map; the $ID\_ERA$ growing SOM map clearly separates incidents attended by the medical rescue service in the western regions from those in the eastern regions, supporting the geographical dependencies in the maps.

The visual quality of the Growing Grid SOM output in this example is even better than that of the more accurate fixed SOM, which takes double the time. The following experiments exploit the growing SOM algorithm.
6.2 Technology Performance Analysis

In the test of the emergency call taking system the aim was to show the system with a heavy load in unrealistic conditions in order to reveal its hidden flaws. Figure 5 (below) shows the anomalies shown in the system technology.

Fig. 5. Maps showing delays in the telephone subsystem (OFFHOOK_DLY), database operations (INSERT_DLY), incident transfer in the emergency centres’ technology (DELAY_ECC) and in the responding agencies’ technology (DELAY_CAD)

Interesting values reported by the analytical module for the maps in Figure 5 are shown in Table 3.

Table 3. An example of the records related to the highlighted nodes from Figure 5. Time intervals characterising delays are in seconds.

<table>
<thead>
<tr>
<th>ID</th>
<th>ID_ECC</th>
<th>REC</th>
<th>RINGING_TIME</th>
<th>INSERT_DLY</th>
<th>OFFHOOK_DLY</th>
<th>DELAY_ECC</th>
<th>DELAY_CAD</th>
<th>ID_ECC_RESP</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>30</td>
<td>20:03</td>
<td>3</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>40</td>
</tr>
<tr>
<td>2</td>
<td>110</td>
<td>20:05</td>
<td>199</td>
<td>15</td>
<td>8</td>
<td>1</td>
<td>4</td>
<td>40</td>
</tr>
<tr>
<td>3</td>
<td>110</td>
<td>20:05</td>
<td>199</td>
<td>15</td>
<td>8</td>
<td>4</td>
<td>4</td>
<td>40</td>
</tr>
<tr>
<td>4</td>
<td>80</td>
<td>20:11</td>
<td>26</td>
<td>1</td>
<td>128</td>
<td>219</td>
<td>20</td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>20</td>
<td>20:12</td>
<td>146</td>
<td>5</td>
<td>500</td>
<td>500</td>
<td>20</td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>20</td>
<td>20:12</td>
<td>157</td>
<td>2</td>
<td>500</td>
<td>500</td>
<td>20</td>
<td></td>
</tr>
<tr>
<td>7</td>
<td>20</td>
<td>20:13</td>
<td>214</td>
<td>19</td>
<td>500</td>
<td>500</td>
<td>20</td>
<td></td>
</tr>
<tr>
<td>8</td>
<td>90</td>
<td>20:13</td>
<td>1</td>
<td>1</td>
<td>228</td>
<td>500</td>
<td>20</td>
<td></td>
</tr>
</tbody>
</table>

Row 1 stands for an example of the normal situation, when emergency centre 30’ received an incident for centre 40’. The database response is 3 seconds, which is within acceptable limits. The delay in the telephone subsystem is 1 second, as is the incident transfer time in the emergency centres’ network and in the emergency response agencies’ technology.

Rows 2 to 8 represent records mined out from the red and blue spots of the four maps in Figure 5. Rows 2 and 3 point to problems with the database and telephone subsystem for centre 110’. The communication with the target centre 40’ and its related emergency agencies is alive, even if the slightly higher values may suggest incoming problems. On the contrary, row 4 shows that centre 80’, when receiving for centre 20’, encounters long database responses, as well as
significant delays in communicating the incident report to the target centre. A minute later, records 5-7 show deadlock at centre 20’. With unacceptable database responses the centre cannot even communicate with the other centres and with its own emergency agencies (here the value 500 stands for unknown information). The last row, row 8, shows that another centre, centre 90’, handles incidents properly, but cannot propagate incident reports to target centre 20’ and its emergency response agencies, definitely because of the problems of centre 20’ reported above.

6.3 Observation of Operators’ Behaviour

This experiment concentrated on a few attributes which we hoped to use in exploring differences in operators’ behaviour under a heavy load. The respective maps are shown in Figure 6.

If there was an anomaly in the time the operator needed to collect basic information from the person calling, it correlated with the length of the call and was caused by the technical problems discussed above. See the correspondence of the red points in the CALL\_LENGTH and MINING\_TIME maps. The ones closest to the centre of the map and its left-hand margin have their counterparts in the OFFHOOK\_DLY and INSERT\_DLY maps from the previous paragraph. These anomalous calls took from 70 to 240 seconds.

The operators of the emergency centres, which did not have technical problems took the emergency calls one after another without any pause and tended to a shorter speaking time, typically 15 to 40 seconds. This is clearly shown by the green areas in the top third of the CALL\_LENGTH and MINING\_TIME maps, while the corresponding continuous area at the top of the ID\_WS\_REC map indicates that these operators were occupying workplaces in Prague.

Fig. 6. Maps characterising operators and their activity: ID\_WS\_REC defines the operator’s workplace at the emergency centre, CALL\_LENGTH shows the time the operator is speaking to the caller, and MINING\_TIME measures the time interval from the start of the call to the moment the incident record was saved or the time which the operator needs to get the basic information from the caller.
7 System Performance and Its Improvement

As mentioned earlier, the system response represents the crucial element of the Emergency Call-Taking System and the same holds for all utilized software components. The Table 2 (see the column named Run time) shows that the computation of the SOM algorithm itself can take a lot of time with respect to the individual settings. This times can differ with respect to hardware platform as well. An alternative hardware platform and software modifications, which bring significant improvements, are described in the following text.

7.1 nVIDIA CUDA

Modern graphics hardware play an important role in the area of parallel computing. Graphics cards have been used to accelerate gaming and 3D graphics applications, and now, they are also used to accelerate computations with relatively distant topics, e.g. remote sensing, environmental monitoring, business forecasting, medical applications or physical simulations etc. Architecture of GPUs (Graphics Processing Unit) is suitable for vector and matrix algebra operations. That leads to the wide usage of GPUs in the area of information retrieval, data mining, image processing, data compression, etc. [27]. Nowadays, one does not need to be an expert on graphics hardware because of existence of various APIs (Application Programming Interface), which help programmers to implement their software faster. Nevertheless, it will be always necessary to keep basic rules of GPU programming to write required code more effective.

Andrecut [29] described computing based on CUDA on two variants of Principal Component Analysis (PCA). The usage of parallel computing on GPU improved efficiency of the algorithm more than 12 times in comparison with CPU. Preis et al. [30] applied GPU on methods of fluctuation analysis, which includes determination of scaling behavior of a particular stochastic process and equilibrium autocorrelation function in financial markets. The speed up was more than 80 times than the previous version running on CPU. Patnaik et al. [31] used GPU in the area of temporal data mining in neuroscience. They analyzed spike train data with the aid of a novel frequent episode discovery algorithm. Achievement of more than 430 speed up is described in the mentioned paper.

7.2 GPU Parallelism

Several different implementations of parallel SOM have already been presented in [33], [34], [35] and [36] and there are also studies on how the computer architecture could be modified in order to support highly parallel calculations especially for SOM training [37]. These approaches focus mainly on time efficiency issues and how the mathematical operations can be distributed on different machines to speed up the self-organization process. Different approach was presented by I. Valova [38], [39], where the amount of parallelism is not determined by the number of available hardware resources but rather by the size of the input pattern.
More convenient approach for a completely parallel SOM has been presented by Weigang [40].

All cited approaches have to solve some kind of vector or matrix operations which are more or less time consuming. The power of GPU is shown on a basic concept of self organizing maps, although proposed improvements can be applied on any kind of SOM.

7.3 Basic Notation

The following notation is used to better describe the process of parallel computation of SOM. All experiments and examples in this subsection follow the specification of SOM presented below:

- The SOM is initialized as a network of fixed topology. The variables $dimX$ and $dimY$ are dimensions of such 2-dimensional topology.
- $V^m$ represents an m-dimensional input vector.
- $W^m$ represents an m-dimensional weight vector.
- The number of neurons is defined as $N = dimX \times dimY$ and every neuron $n \in <0, N-1>$ has its weight vector $W^m_n$.
- The neighborhood radius $r$ is initialized to the value $\min(dimX, dimY)/2$ and will be systematically reduced to a unit distance.
- All weights vectors are updated after particular input vector is processed.
- The number of epochs $e$ is know at the beginning.

7.4 Kernel Functions

Well designed CUDA kernel functions enable us to get better performance during runtime. Kernels functions, when called, are executed N times in parallel by N different CUDA threads, as opposed to only once like regular functions. Thus it is necessary to design a kernel function so that every thread access different memory block. We refer to [45] for more information on CUDA memory management.

A kernel function is defined using the `global` declaration specifier and the number of CUDA threads for each call is specified using a new `<<< ... >>>` syntax. The brackets `<<<` and `>>>` hold grid size, block size, amount of shared memory and stream index. The setting of these parameters affect the distribution of threads over GPU. We refer to [27] for more information.

The principle of PR (Parallel Reduction) brings a significant time savings. A modified version of PR was used in a phase of computation of vector differences and Euclidean distances. The Figure 7 shows an illustrative example of settings of grid, blocks and threads which were applied on SOM, where the number of neurons $N = 9$ and vector dimension $m = 8$. The GPU grid contains 3 blocks. Every block has 32 threads, therefore it can cover at most four weights vectors. The setting of kernel function itself play an important role in CUDA. It depends on GPU hardware parameters as well. The parallel reduction is shown in the middle part of the Figure 7 and has following steps:
A: The differences between all elements of a given input vector and an appropriate weight vector are computed and stored in the shared memory on GPU. The elements can be powered by 2 in case of Euclidean distance.

B: A half of previously used threads computes partial sums of elements of difference vector.

C: Last thread computes square root of the final sum and stores the result output into distance vector, which is stored in global GPU memory.

Fig. 7. Processing of a single input vector by parallel GPU threads

when all the blocks finish, the process of parallel reduction is used to find a minimum value in the distance vector. An index of the minimal element represents BMU (Best Matching Unit).

7.5 Parallel SOM Update

Some fix data is precomputed in our application. Such data is stored in the global memory and then partially copied into shared memory by all thread blocks to reduce computational time [45] during SOM update. First, it is a distance matrix which holds the distances among neurons in a SOM. In the figure 8 on the left, there is an example of the SOM with rectangular topology of dimension 7×7. The maximum distance in this topology is \( \text{maxDist} = (\text{dimX} - 1) + (\text{dimY} - 1) = (7 - 1) + (7 - 1) = 12 \). Just for illustration, the black node in the middle represents BMU and all nodes with the same gray scale have the same distance to the BMU with respect to the rectangular topology.

A vector of learning factors \( F \) makes the second fix data. In the illustrative figure 8 on the right, there are curves of learning functions for all epochs (\( e = 4 \)). The vector \( F \) has a dimension equal to \( (\text{maxDist} + 1) \times e = (12 + 1) \times 4 = 48 \).
in our example and represents a set of values \( f_i(d) \), where \( f_i(d) \) is the evaluated learning function (learning factors) for a given epoch \( i \), and a distance \( d \), where \( d \in <0, maxDist> \) and \( i \in <0, e> \).

**Fig. 8.** Neuron distances in the SOM for a given BMU in the center of the SOM (on the left) and curves of learning functions for all epochs.

The process of updating of SOM runs in a few steps:

1. The GPU kernel function, that is responsible for updating, is set so that one neuron is processed by \( m \) threads, where \( m \) is a dimension of weight vector.
2. Fixed distance vector and learning factors for a given epoch are copied into GPU shared memory to ensure better performance.
3. Every thread updates a part of its weight vector with respect to the learning factor and the distance to BMU.

### 7.6 Kernel Example

Although there is no place to show the whole code of GPU implementation of SOM because it contains hundreds of lines, following part of the code illustrates the philosophy of parallel programming with CUDA. The kernel function `update-CompleteSOM` is called from the main program when BMU is already known. We refer to [27] for more information on GPU programming and to [28] for a detailed description of implementation of SOM on GPU.

### 7.7 Experiments on GPU

All experiments include comparison between GPU and CPU implementation of SOM. The Table 4 includes detailed specifications named GPU, CPU-A and CPU-B with respect different hardware configuration used for experiments.
void updateCompleteSOM(float* weightVectors, const unsigned int* wSize, const unsigned int* distanceMatrix, const unsigned int* BMUindex, const float* distances, const float* learningFactors)
{
    unsigned int tid = threadIdx.x;
    shared unsigned int sBMUindex;

    if (tid == 0) sBMUindex = *bmuIndex;
    syncthreads();

    unsigned int wid = blockDim.x * blockIdx.x + tid;
    if (wid >= wSize) return;

    unsigned int nid = (wid / vectorDim) * dimX * dimY + sBMUindex;
    weightVectors[wid] += learningFactors[distanceMatrix[nid]]
        * distances[wid];
}

Table 4. Hardware specification

<table>
<thead>
<tr>
<th></th>
<th>GPU</th>
<th>CPU-A / CPU-B</th>
</tr>
</thead>
<tbody>
<tr>
<td>CPU</td>
<td>Intel Core 2 Duo 3,0Ghz</td>
<td>4 x AMD Opteron 1,8 GHz</td>
</tr>
<tr>
<td>RAM</td>
<td>4 GB</td>
<td>32 GB</td>
</tr>
<tr>
<td>GPU</td>
<td>GeForce 280 GTX, 1 GB</td>
<td>-</td>
</tr>
<tr>
<td>Threads</td>
<td>depends of GPU</td>
<td>2 (CPU-A), 8(CPU-B)</td>
</tr>
</tbody>
</table>

A SOM with a fix network size is computed in the first experiment. Dimensions of input vectors are fix as well and they are set to 4 in this case. This experiment reveals hidden disadvantages of GPU utilization, as can be seen in the Figure 9 or in the corresponding table. The computation time increases near linear, however, GPU time is approximately 2 times greater in comparison with CPU-A multi-threaded implementation. This is due to additional time costs associated with transactions between RAM and GPU memory. Next, calling kernels (GPU functions) and inner thread indexing take some time. These costs make GPU implementation not suitable for SOM in case of small data.

In the Figure 10, there are results of the second experiment which deals with increasing dimension of input and weight vectors. Again, GPU implementation seems to be not effective until dimension 32 because of the same reasons as in the first experiment. However, the trends of graphs predict future computation times and it is clear, that GPU time increases very slowly in comparison with CPU-A.

The third experiment and its results in the Figure 11 show the power of parallelism on neurons (we refer to the section 7.5), whereas the previous two experiments illustrate parallelism on particular vectors (we refer to the section 7.4). The contribution of GPU implementation is perceptible in higher dimension of SOM network.
Previous demonstrations confirm, that the power of GPU utilization increases with the size of neural network and with the dimensions of input and weights vectors, respectively. The last two experiments (see the Figure 12 and 13) show the computation times in such cases.

Although the massive parallelism is not the main goal in case of Monitoring of the Emergency Call-Taking System, it brings additional benefits in the area of data analysis. In case of GPU utilization, the shorter computation time enables us to retrieve more accurate results because more input data can be processed and the size of neural network can be larger in comparison with CPU version.
8 Discussions

The emergency call system could be locally overloaded in major crisis situations, natural disasters, or big accidents. As the system is distributed over the whole territory of the state, operators in the regions not affected by a crisis can receive calls from the affected regions. An intelligent system reconfiguration could help in setting up routing policies, optimising network traffic, and balancing the system load with respect to the available system resources.

In the first part of our work, we proposed transformations of categorical attributes, the discrete values of which are not suitable for the SOM algorithm, into the real numbers domain and showed that after this transformation the SOM is generally able to detect anomalies in the emergency data. The transformations are built on the frequencies of incidents in time and place, expressing the relevance of the incident with respect to the time and place of origin of the incident. The learning proceeds on the values of the transformed attributes, which are nevertheless bound to the original records. In this context, the values of the original understandable attributes could be used for explaining the result, as well as for further processing.

In the second part of the paper, we presented the dynamic SOM alternative, the Growing Grid [7], enhanced with a variable learning epoch counter heuristic. It has been shown that the growing SOM with the above-mentioned heuristic produced satisfactory results in a significantly shorter time. We programmed a corresponding software tool and used it in experiments with data collected in the
period of a specific emergency system test to simulate performance monitoring of the emergency call centres network and analyse performance issues related to the technology, as well as the human factor the operators of the centres. The proposed SOM modification is well suited to the emergency data domain, which is characterised by relationships of a geographical, organisational, and technological nature. As some of the relationships in the emergency data are inherently hierarchical, we consider the application of the hierarchical SOM [4] in the SW tool developed in the course of this work.

While the first part of the paper was to show the justification of the SOM usage in detecting and visualizing anomalies in the emergency data, in the second part we exploited this partial result in the simulation and performance monitoring of the emergency call centres network under a heavy load. The load was simulated by the system tests which were lasting for one hour. In the tests 80 emergency call operators over the whole country took part together with about 200 of callers who were simulating emergency calls. The data which were describing the behaviour of the system as well as of the operators were collected and processed by the modified growing SOM algorithm. In this way the anomalies in the whole system or in behaviour of all the operators could be seen in the resulting SOM picture.

As one could see and comprehend the status of the whole system at a glance, this holistic visualization of the system performance tends to be much more transparent and intuitive comparing to the traditional monitoring techniques based on trapping and displaying particular events. Moreover, the proposed method inherently includes monitoring of non-technical components of the system, particularly the behaviour of the emergency call operators, which is not covered by traditional trap-based monitoring. From this point of view the method described in this paper could be passed as a contribution to the socio-technical system performance monitoring field.

Traditional anomaly detection methods [9,22,26] use SOM for modelling the anomaly-free space from a set of data approved as correct (bearing signs of supervised learning) and then classify a new case as anomalous if it falls outside the modelled non-anomalous space. As hardly any emergency situation can be considered anomalous, or all of them are anomalous on the other hand, we could not use the two-class classifier approach with supervised learning. Therefore we searched for certain patterns in the data. After the patterns were recognised by unsupervised learning, the composition was always presented to a human for them to analyse the situation.

This concept can be enhanced in such a way that, provided that the algorithm is run periodically, the new composition would be further processed only if the composition in the current run is different from the composition in the previous run. SOM quality measures [20,13] could help in achieving good results here. We expect that the monitoring of the performance of the emergency call-taking system and analysis technique presented in this paper can be effectively combined with a model of resilient incident report transfer between call centres inspired by [10,17]. Such a model could be created and the Growing Grid SOM tool used for its output monitoring and performance analysis, provided that detailed
information regarding the underlying technology of the ECC network (the WAN topology, technology and parameters) were available.

9 Conclusion

This paper presented the monitoring and analysis of the performance of the emergency call system, using a novel approach of exploring anomalies in the emergency call system databases. We reused our experience with the detection of anomalies, as described in [14], concentrating on the technology and cooperative characteristics of the emergency call-taking system to show the system’s performance issues. Both the technological and personal points of view were included as our method discovered database problems and spotted operators’ behaviour in the course of testing a specific system under a heavy load.

We devised a novel method for unsupervised knowledge discovery in the emergency call data, based on the Self-Organising Map (SOM) algorithm and programmed respective software tools: the fixed and Growing Grid SOM learning module and the SOM explorer module. We compared various configurations of SOM learning and showed that the Growing Grid, combined with the variable learning epoch count heuristic, produces the best results in terms of the speed and quality of the output.

The algorithm consumed the training set of records describing the creation and transfer of incident reports, identified a subset of records containing certain common information, and built and visualised clusters over these records. The method devised here proved its ability to discover anomalies hidden in emergency data, visualising them in an effective user-friendly manner and indicating a way towards the further development of the intelligent management of the emergency call information system.

The experiments presented here focused on revealing clusters of incident records which pointed to abnormal situations in time and place and formed a sound basis for the performance monitoring and analysis of the emergency call system.

References


Utilization of Parallel Computing for Discrete Self-Organizing Migration Algorithm

Marek Běhálek, Petr Gajdoš, and Donald Davendra

Abstract Evolutionary algorithms can take advantage of parallel computing, because it decreases the computational time and increases the size of processable instances. In this chapter, various options for a parallelization of DISCRETE SELF-ORGANISING MIGRATING ALGORITHM are described, with three implemented parallel variants described in greater detail. They covers the most frequently used hardware and software technologies, namely: parallel computing with threads and shared memory; general purpose programming on GPUs with CUDA; and distributed computing with MPI. The first two implementations speed up the computation, the last one moreover changes the original algorithm. It adds a new layer that simplifies its usage in the distributed environment.

1 Introduction

DISCRETE SELF-ORGANISING MIGRATING ALGORITHM (DSOMA) [8, 7] is one of the newer meta-heuristic algorithms. It has been developed to solve (NP-hard) combinatorial optimization problems. For such problems (it is believed), that there is no exact method to get a solution in a reasonable period of time. Still, computer

Marek Běhálek
Department of Computer Science, FEECS, VŠB – Technical University of Ostrava and IT4Innovations National Supercomputing Center, VŠB – Technical University of Ostrava, Ostrava, Czech Republic e-mail: marek.behalek@vsb.cz

Petr Gajdoš
Department of Computer Science, FEECS, VŠB – Technical University of Ostrava and IT4Innovations National Supercomputing Center, VŠB – Technical University of Ostrava, Ostrava, Czech Republic e-mail: petr.gajdos@vsb.cz

Donald Davendra
Department of Computer Science, FEECS, VŠB – Technical University of Ostrava, Ostrava, Czech Republic e-mail: donald.davendra@vsb.cz
scientists and programmers frequently encounter such problems and they are often addressed by heuristic methods or approximation algorithms. But even if we use these algorithms, the solution remains still very computationally demanding. Parallel/distributed computing in this aspect can be a great asset. It can decrease the computational time or increase the size of processable instances.

In this chapter, different options how to take advantage of the parallel computing for DSOMA are explored. Firstly, approaches that are frequently used in this area are described. Then various options how to parallelize DSOMA are analyzed. Finally, three selected variants that were implemented are introduced. The last section briefly summarizes results of various experiments that were performed.

2 Levels of Parallelization

Various meta-heuristics have been introduced in the last years. There are articles (for example [1, 21]) that try to categorize the most frequently used approaches, how to extend these algorithms to take advantage from the parallel/distributed computing architecture. DSOMA is a population based meta-heuristic, that iteratively searches for better solutions. Iterations in DSOMA are called migrations. The DSOMA population composes of individuals. In each migration, selected pairs of individuals are used to produce new trial individuals. If a better solution is found among the trial individuals, then the successful trial individual replaces the original individual in the population.

From the algorithm point of view, we can define different levels, where we can apply parallel computing.

- **Fitness function** - usually the most time consuming function is the computation of the fitness function. It is frequently used in every step of DSOMA. First option how to speed up the computation is to speed up this function. On the other hand, used fitness function depends on the solved problem and some fitness functions may be easier to parallelize then others.

- **Constructing trial individuals** - Trial individuals in DSOMA are computed based on two individuals from the population. After the trial individuals are generated, they are repaired and their fitness values are computed. These last two operations are more time consuming then generating the trial individuals, moreover for a given set of trial individuals they are independent and thus suitable to be performed in parallel.

- **Computing a migration** - a common strategy is to select pairs of individuals and construct trial individuals by combining the current best individual in the population with the remaining individuals. Computing the trial individuals for a given pair does not depend on other pairs and it can be performed in parallel.

To improve the obtained results, DSOMA (similarly to other evolutionary algorithms) utilizes some local search algorithm. In [8], the 2-Opt local search is utilised. Because an application of such local search algorithm on a single individual can be
even more time consuming than the whole DSOMA computations, its parallelization can be crucial, if we want to speed up the computation.

Moreover, similarly to other population based evolution algorithms, we can add some new layers to improve the overall parallel behaviour. For example, if we want to use a computer (cluster) with the distributed memory architecture (they represent a majority between current supercomputers) then none of the previously described levels may be appropriate. The reason is the communication bottleneck, which can outweigh the speed up achieved by the parallel execution.

The most popular and simplest model in the distributed environment is the so called *island model* [1, 2]. In this model, the population is partitioned in a small set of sub-populations (islands, colonies). These islands execute the original algorithm and then some (sparse) individuals exchange algorithm is applied to exchange information between such islands. It was shown, that sparse information exchange between such islands brings diversity into the population and thus prevents convergence in local optima. Moreover, the overall parallel behaviour is affected by the used communication topology or exchange data rate [22, 14].

In practical applications, some hybrid approaches that combine more than one level of parallelization are used (see Section 4.2). In general, the higher level of parallelization is coarse-grain implementation (for example the mentioned set of islands) and then each of these islands integrates other parallel model (or models).

### 3 Hardware and Software Options for Parallelization

In Section 2, different levels of parallelization for DSOMA were mentioned. But the appropriate level (or levels) of parallelization depends also on the target hardware and used software technologies.

The following paragraphs summarize the most distinguishing options that are state-of-the art in the world of high performance computing (HPC). The first such choice, that greatly affects the solution especially from a programmer’s perspective, is a target device that will be used for computing. There are two main options:

- **general purpose processors** - current (super)computers used for HPC contains up to hundreds of thousands of computational cores. Even current mainstream desktop processors contain between 4-8 cores. Considering the usage of general purpose processors, the parallelization usually implements a model - Multiple Program - Multiple Data (MPMD). In this model, each core runs independently of others and can perform a different program with its own data.

- **many-core (or massively multi-core) coprocessors** - the most common devices from this category are graphics processing units (GPUs) that allows general purpose programming. However, recently Intel has also introduced their Xeon PHI 1, a coprocessor for HPC. These coprocessors usually contain many (tens or

---

hundreds) lightweight cores and frequently implement Single Instruction - Multiple Data (SIMD) model. It describes devices with multiple processing elements that perform the same operation on multiple data simultaneously.

Considering the many-core coprocessors, usually the computation is divided between general purpose processor and many-core coprocessor. Such SIMD coprocessors successfully exploit data level parallelism, but not concurrency. A common approach is to use the coprocessor when appropriate and for remaining computations use the general purpose processors. Moreover, most applications do not use the coprocessor and CPU at the same time [16, 13]. Some effort has been made to exploit the full computation power of CPUs and coprocessors at the same time for evolutionary algorithms [26], but this idea is not explored in this chapter any further.

Current computers used in HPC frequently combine both type of devices. For example a computer named TITAN (from the Top 500 list\(^2\)) is a cluster composed from 18 688 nodes where each node contains a traditional general purpose processor with 16 cores and the NVIDIA TESLA K20 GPU accelerator.

Considering general purpose processors, there are two main memory architectures.

- **Shared memory architecture** - in this architecture, all processors share the same memory and this memory is used to exchange data and for synchronization. This architecture is frequently used for smaller HPC computers with tenths or hundreds computing cores.

- **Distributed memory architecture** - it is the main model in todays supercomputers. In this architecture, a computer (usually a cluster) composes from interconnected nodes. Each of these nodes usually contains multiple computational cores with the shared memory. Nodes can exchange some kind of messages, but they cannot directly access a memory in a different node. Thus the memory is in fact distributed among the nodes.

Considering HPC, the most widely used programming languages are C, C++ or FORTRAN. In this chapter, we will focus on C++, as our original sequential implementation of DSOMA is also in C++.

Concurrent programming with threads is the simplest way as to how to create a parallel application in the shared memory environment. Nearly every programming language supports this programming paradigm. Also C++ contains libraries for concurrent programming with threads\(^3\). Moreover, there are technologies that simplify the development of such applications. Like an example we can list OPEN MULTI-PROCESSING (OPENMP)\(^4\), CILK and CILK++\(^5\). For the distributed memory model, the *de facto* standard is MESSAGE PASSING INTERFACE (MPI)\(^6\).

---

\(^2\) [http://www.top500.org/system/177975](http://www.top500.org/system/177975)


\(^4\) [http://openmp.org/](http://openmp.org/)

\(^5\) [https://www.cilkplus.org/](https://www.cilkplus.org/)

The technologies for programming on GPUs have quickly evolved in last ten years. Currently, there are various approaches and technologies. Earlier, general purpose computing on graphics processing units was primarily based on modification of graphics pipeline, e.g. in OPEN GRAPHICS LIBRARY (OpenGL) where programmable shaders are used for time consuming BASIC LINEAR ALGEBRA SUB-PROGRAMS (BLAS). Nowadays, OPEN COMPUTING LANGUAGE (OpenCL) is the currently dominant open general-purpose GPU computing language. It is an open standard defined by the Khronos Group 7. The dominant proprietary framework is NVIDIA’S COMPUTE UNIFIED DEVICE ARCHITECTURE (CUDA) that was launched in 2006 as an SDK and API that allows using the C programming language to code algorithms executable on GPUs. MICROSOFT 8 also introduced its own API called DIRECTCOMPUTE that supports general-purpose computing on GPUs, this time with full support of DIRECTX.

In the following subsections, technologies that are later used for parallelization of DSOMA are introduced in more detail.

3.1 OpenMP

OPENMP is maybe the most widely used technology considering shared memory architecture. Moreover, it is also frequently used in combination with MPI for distributed memory systems. OPENMP is an API that supports multi-platform shared memory programming in C, C++, and FORTRAN. Its support is implemented on most processor architectures and operating systems. It consists of a set of compiler directives, library routines, and environment variables that influence run-time behavior.

The simplest example of the OPENMP usage is the automatic parallelization of for cycles. Listing 1 demonstrates a simple for cycle. The pragma directive defines, that this cycle should be parallelized by OPENMP. OPENMP is supported by a wide range of compilers. For example, its support is built in widely used GNU COMPILER COLLECTION (GCC) 9. This directive is ignored by a compiler, if there is no OPENMP support or if it is switched off.

```
const int N = 100000;
int a[N];

#pragma omp parallel for
for (int i = 0; i < N; i++) {
    a[i] = 2 * i;
}
```

---

7 http://www.khronos.org
8 http://www.microsoft.com
9 https://gcc.gnu.org/onlinedocs/libgomp/Enabling-OpenMP.html
Listing 2 shows how to enable OPENMP with GCC g++ compiler (parameter \texttt{-fopenmp}). Using the environment variable \texttt{OMP_NUM_THREADS}, it is possible to define the number of threads that are used during the execution.

**Listing 2** Compilation and execution of a source code with OPENMP pragmas in g++

```
g++ -fopenmp source.cpp -o result
OMP_NUM_THREADS=4 ./result
```

Even if OPENMP automatically parallelize annotated \texttt{for} cycles, other issues like a concurrent modification of shared data, must be solved by a programmer. OPENMP provides additional constructs controlling data sharing or synchronization.

### 3.2 Message Passing Interface

MPI is a language-independent communications protocol used to program parallel computers or clusters with distributed memory architecture. It defines a message-passing interface, together with protocol and semantic specifications for how its features must behave in any implementation. The MPI applications compose from independent processes where the MPI messages and constructs are the only communication mechanisms. It is used to exchange data and also to solve synchronization issues.

The primary functionality is a point-to-point and collective communication. Listing 3 shows a simple example of a point-to-point communication in MPI. Similarly to OPENMP, a user specifies the number of processes when the application starts. Each process is uniquely identified by its rank (a number from 0 to the number of processes). In the following example, two processes are expected. Process ranked as 0 sends some data and the process 1 is waiting until the data arrives.

**Listing 3** A simple MPI example

```c
MPI_Init (&argc, &argv);
int myrank;
MPI_Comm_rank(MPI_COMM_WORLD, &myrank);
if (myrank == 0){
    int data[10];
    MPI_Send(data, 10, MPI_BYTE, 1, 1, MPI_COMM_WORLD);
}
if (myrank == 1){
    int data[10];
    MPI_Recv(data, MPI_BYTE, 10, MPI_ANY_SOURCE, MPI_ANY_TAG, MPI_COMM_WORLD);
}
MPI_Finalize();
```

Even though MPI is relatively simple to use, it represents a quite low-level interface. Hence development of applications directly in C++ with MPI can be laborious.
and time-consuming. Furthermore, the complexity of creating parallel applications lies also in other supportive activities like debugging and profiling. Even an experienced programmer of sequential applications can spend a lot of time learning a new set of complex tools. Therefore, for many non-experts in the area of parallel computing, it can be difficult to create, debug, and analyze their distributed applications.

To overcome the described issues, the tool KAIRA \cite{4, 3} was used for the development of a distributed variant of DSOMA. KAIRA is a complete development environment for MPI C++ applications. It provide an environment in which a user can implement and experiment with his or her ideas in a short time; create a real running program; and verify its performance, scalability, and correctness.

KAIRA is an open source application and it is freely available at: http://verif.cs.vsb.cz/kaira/.

3.2.1 Brief Introduction into KAIRA

The key aspect of KAIRA is its usage of a visual program. A user specifies communication and parallel aspects in a visual way. However, the application is not completely programmed visually. Sequential parts of the developed application are written in C++ in a textual form and they are inserted into the visual program. So for real applications, the visual program is usually relatively small. The visual representation serves also as a natural unifying element for supportive activities like debugging and performance analysis. The used visual programming language is inspired by COLORED PETRI NETS (CPNs) \cite{10}.

From the combination of the visual program (that captures parallel aspects) and inserted sequential codes, KAIRA is able to generate a stand-alone MPI application in a fully automatic way. Such a generated program can be run directly on a cluster computer. For debugging purposes, multi-threaded and sequential versions of the application can be generated. It is important to mention that KAIRA is not an automatic parallelization tool. It does not discover parallelisms in applications. The user has to explicitly define them, however they are defined in a high-level way and the tool derives the implementation details.

In Section 4.2, a KAIRA’s visual program is used to present a parallel behaviour of an implemented distributed variant of DSOMA. The basic notation is the same as that of CPNs, hence circles (places) represent memory spaces and boxes (transitions) represent actions. Places have its types defined in lower right corner. In the upper right corner, there is an initial marking. Moreover, transitions can have priorities (a number in a upper right corner). Transitions with higher priorities are executed (fired) before transitions with lower priorities.
Figure 1 shows a simple example - *Ping-Pong* in KAIRA\textsuperscript{10}. This net is executed on each (MPI) process. An enriched C++ is used as an inscription language on arcs. When fired, every transition takes values (tokens) from its input places and produces tokens into its output places. The expression in the form \texttt{expr@target} means that created tokens (by evaluating \texttt{expr}) are sent to another process determined by its evaluating \texttt{target}. It allows communication between (MPI) processes.

In this example, first two processes exchange an integer token. Transition \texttt{ping} takes a value (named \texttt{x} in a scope of this transition), increments this value (by the expression \texttt{(x+1)}) and sends it to the process 1 (\texttt{@1} on its output arc). Similarly, transition \texttt{pong} takes a value, it increments this value and sends it to the process 0. While the initial marking is used only for process 0, there is just one token 0 at process 0 at the beginning of the computation and the only enabled transition is the transition \texttt{ping} on process 0 (it is detonated by green elements in Figure 2). After it is fired, there is just one token with a value 1 on process 1 and only the transition \texttt{pong} is enabled (see Figure 3). After the transition \texttt{pong} is fired, the token returns to process 0 and the exchange can start again (Figure 4).

The double bordered transition contains a C++ code that is executed whenever the transition is fired. This code is in fact a simple function with predefined definition. This definition is automatically derived from the net.

Figure 5 contains an initial marking as an example, where a C++ code is inserted into the transition \texttt{Compute}. This code is present in Listing 4. The structure \texttt{Vars} and the function’s header are generated automatically. Arcs detonated with \texttt{[bulk]} construct take not only one token but all tokens during the transition’s firing. The

\textsuperscript{10} The example is a part of Kaira’s distribution.
transition Compute takes all numbers from place \( p_1 \) and it multiplies them by a value from \( p_2 \). The results are inserted into the vector \( \text{result} \). The first step in computation is captured by Figure 6.

![Diagram](image)

**Fig. 5** An example with inserted C++ code

**Listing 4** The function inserted into the transition Compute form Figure 5

```cpp
struct Vars {
  std::vector<int> &result;
  int &x;
  ca::TokenList<int> &y;
};

void transition_fn(ca::Context &ctx, Vars &var) {
  ca::Token<int> *t;
  for (t=var.y.begin(); t!=NULL; t=var.y.next(t)) {
    var.result.push_back(t->value * var.x);
  }
}
```

### 3.3 GPU Computing with CUDA

**Compute Unified Device Architecture (CUDA)** [5, 28] was introduced by NVIDIA as a general parallel programming and computing platform in 2006. Although utilization of CUDA was very limited at the beginning and the first GPU chips based on G80 TESLA ARCHITECTURE were too expensive, it became very popular after a few years. NVIDIA developed their own hardware architecture that enables solving known problems in shorter time due to massive parallelism. The GPUs were well-suited to address real problems that could be expressed as data-parallel computations. Except for the gaming community that still plays a very important role for hardware producers and vendors, NVIDIA decided to focus on re-
search and computational areas as well. Nowadays, CUDA is the most popular and supported architecture running on GPUs \[6, 12, 25\] and its environment allows a heterogeneous programming approach.

Simply stated, a part of program pipeline can be processed by GPU, whereas another part by CPU. Moreover, current design patterns for parallel programming \[16, 15, 13, 27\] strongly suggest to distinguish between CPU and GPU parts to achieve the best application performance. A good program structuring itself does not ensure expected results. Memory arrangement and data alignment belong to the most important tasks that the programmer has to solve. CUDA ready graphic cards have basically five types of memory. The biggest is a global memory, which serves as a communication point between GPU and CPU and a primary storage if the data can not be given somewhere else. Since global memory is too slow, a shared memory (shared by all threads in one thread-block) is often employed to achieve better performance. Registers represent the fastest memory. Constant and texture memory are used in such cases where the read-only data structures can be used. The most important functions performed on GPUs are called kernels. Every kernel needs its runtime configuration that consists at least of CUDA grid and block settings. Both grid and blocks can have up to three dimensions. The grid consists of blocks and every block encapsulates a set of threads. The computation is performed on several streaming multi-processors independently (MAXWELL MULTIPROCESSOR SMM on MAXWELL architecture). The maximum number of blocks, threads and SMMS depends on GPU specification. We refer to \[5, 23, 28, 20\] for more details on CUDA programming since it is out of scope of this chapter.

Internal architecture of GPUs is suitable for vector and matrix algebra operations. That leads to the wide usage of GPUs in the area of information retrieval, data mining, image processing, data compression, etc. Nowadays, programmers usually choose between OPENCL which is supported by all hardware producers, and CUDA which is supported by NVIDIA only. An important benefit of OPENCL is its platform independence; however, CUDA still sets the trends in GPU programming.

CUDA kernels are usually relatively complex and particular implementation can require suitable data arrangement and indexing. Listing 5 represents an illustrative kernel implementation of the same loop as can be seen in Listing 1.

**Listing 5** A simple kernel in CUDA

```c
__global__ void foo(const unsigned int N, int* a) {
    unsigned int tOffset = blockIdx.x * blockDim.x + threadIdx.x;
    while (tOffset < N) {
        a[tOffset] = 2 * tOffset;
        tOffset = blockIdx.x * blockDim.x;
    }
}
```
4 Parallelization of DSOMA

In this section, three implemented parallel versions of DSOMA are introduced. All these variants are implemented in C++. First two solutions use general purpose processors. First is an OpenMP solution that uses the shared memory. OpenMP was chosen because it is relatively easy and it provides a meaningful speedup even on a common desktop computer. Second solution uses the distributed memory architecture. From the programmer’s perspective, it combines the usage of MPI with OpenMP. This solution can meaningfully use a computational power of hundreds of processors. Finally, the last solution uses the CUDA technology for the general purpose programming on NVIDIA GPUs.

4.1 OpenMP Implementation of DSOMA

OpenMP is a relatively easy to use technology. It is usually easy to identify time consuming for cycles that are suitable for the parallelization with OpenMP. Still, to get a meaningful speed up and at least modes scalability, it requires additional work and a programmer’s insight.

Simplified main method from our original sequential DSOMA implementation is captured in Listing 6. The most important class is Soma. The instance of this class allocates a memory space for storing the DSOMA population, information as to which individual represents the current best solution and also it contains a memory space for computing the fitness function and constructing trial individuals.

What remains are in fact two for cycles. The outer one is not suitable for a parallelization, because the current best solution is used during the construction of trial individuals. So, each step depends on results of the previous steps. In contrary, the inner cycle computes trial individuals for selected pairs from the population. Each of these computations is in fact independent and the trial individuals for every pair can be computed in parallel.

Listing 6 Initial simplified DSOMA main function

```cpp
Problem problem = new Problem(); // a problem description
// an object storing the population and an allocated memory
// for computing the fitness function and jump solutions.
Soma soma(problem);
soma.InitializePopulation();
for(int i=0; i<soma.GetNumberOfMigrations(); i++)
{
    for(int j=0; j<problem.numberOfIndividuals; j++)
    {
        // compose trial individuals between the best solution in
        // the population and i-th solution in the population
        soma->ComputeTrialIndividuals(i);
    }
}
```
soma.UpdateBestSolution();
}

So, the basic idea is to add the OpenMP pragma to parallelize the inner cycle. Still, there are some issues. First of all, there are some memory issues that can lead to incorrect results. To compute the fitness function or the trial individuals in parallel, each of the involved threads needs its own memory to store data for ongoing computations. Moreover, while the population changes during the computation of a migration (new better individuals are included into the population), it is easier to prepare the whole migration before its computation starts. These modifications along with OpenMP clauses are captured in Listing 7.

Listing 7 The modified DSOMA main function with OpenMP pragmas

```cpp
Problem problem = new Problem(); // a problem description
Soma soma(problem); // the population
soma.InitializePopulation();

for(int i=0; i<soma.GetNumberOfMigrations(); i++)
{
    //prepared migration
    std::vector<Migration*> migration;
    soma.PrepareMigration(migration);

    #pragma omp parallel default(shared) shared(soma, migration)
    {
        //contains a memory for computing the fitness function
        Fitness fitness(problem);

        //a memory buffer for generated trial individuals
        MemoryForTrials trials(problem.numberOfJobs);

        #pragma omp for schedule(dynamic,20)
        for(int j=0; j<migrations.size(); j++)
        {
            Solution solution;
            migration.at(j)->Compute(solution, fitness, trials);
            soma.IncludeSolution(solution);
            delete migration.at(j);
        }
    }
    migration.clear();
    soma.UpdateBestSolution();
}
```

After the mentioned memory issues were removed, the parallel implementation computed correct results. Still, the resulting application worked even slower than the original sequential implementation. The reason was due to some functions from the C++ API. For an example the function `rand`\(^{11}\) can be named. It is frequently called while repairing trial individuals. An access to this function is restricted to a

\(^{11}\) http://www.cplusplus.com/reference/cstdlib/rand/
single thread, because it modifies internal state objects and thus its usage slows the whole computation. As a solution, the function \texttt{rand} was replaced by the functions \texttt{rand_r}.

After such issues were solved, we got the desired OPENMP parallel implementation. Its performance (scalability and speedup) is summarized in Section 5.

4.2 Distributed Island Model implementation of DSOMA

As was mentioned in Section 3.2.1, a distributed MPI implementation of DSOMA was implemented in KAIRA. The visual program capturing the communication aspects is shown in Figure 7.

![Fig. 7 A visual program in Kaira defining communication for the distributed implementation.](image)

By default, initial marking is used only for process 0. The blue area is used for initialization in KAIRA, defining processes where the initial marking is applied. In our case, it is applied to all processes (the range is defined by the expression: \texttt{ca::range(0, ctx.process\_count())}). So, every process has its migration counter and an instance of a class \texttt{Soma}. It is the same class as in Listing 7. It stores the DSOMA population and functions necessary to compute new migrations. It represents an island in the mentioned island model.

The most important transition is \texttt{Compute migration}. It computes one migration. The same code as in Listing 7 is used. Also for this variant, the same OPENMP constructs are used to speed up the computation of a single migration. Moreover, it optionally sends the current best result to a neighbouring process. In our implementation, islands are connected in a ring configuration, where a process with rank \( x \) sends its results to a process ranked as \( x+1 \). The last process sends results to process 0. Results are exchanged every \( n \)-th migration (it is set as a command line parameter when starting). An index \(-1\) is used to define empty solutions, which are not exchanged between processes.
After the desired number of migration is computed (or a wall-time is reached), the best results from all processes are gathered and the overall best results is obtained.

4.3 GPU Implementation

In case of GPU utilization, the emphasis is put on the partial subtask that must be solved in advance, e.g. data transfers, memory allocations, the blocks and grids settings, etc. Finally, all implementation must cover individual hardware specification to achieve the best performance. This can be illustrated on data alignment and arrangement of thread blocks.

4.3.1 Data storage, transfers and alignment

Searching for the best parallel implementation usually starts with the design of data storage and analysis of all data transfers. There are several kinds of memory on CUDA devices, each with different scope, lifetime, and caching behaviour. The global memory, which resides in the device DRAM, is used for transfers between the host and device as well as for the data input to and output from kernels. Usually, dynamic and/or large data is stored in the global memory. However, the best approach is often based on pre-cached data model, where all static data is copied into fast memory to avoid higher latency of I/O operations. Texture memory meets these requirements. It is much more faster than the global memory due to texture cache and supports enough space for allocation. Moreover, Maxwell architecture brings a new model that combines unified L1/texture cache. The unified L1/texture cache acts as a coalescing buffer for memory accesses, gathering up the data requested by the threads of a warp prior to delivery of that data to the warp. This function was previously served by separated L1 cache in Fermi and Kepler. Texture memory was used to store a static timetable represented by a matrix \( A[M \times J] \), where \( M \) is a number of machines, \( J \) is a number of jobs and \( A[m,j] \) is a time demanded by a machine \( m \in \{0, 1, \ldots, M\} \) to process a job \( j \in \{0, 1, \ldots, J\} \). Other data such as vectors of machines or jobs permutations are stored in the global memory and can be transferred into shared memory within individual CUDA kernels. This data is modified very often when DSOMA is running and the only chance to increase I/O accesses is to use some inner CUDA optimization mechanism, such as `_restrict` pointers.

In case of DSOMA, all jobs permutations are stored in form of vectors in the global memory. Although accessing global memory is quite slow, the data alignment plays an important role as well and has a significant influence on final computation time.

Let \( x \in \mathbb{N}_0^N \) represents a single jobs permutation of dimension \( N \). An element of such permutation is represented by zero-based index of a job (jobID) in DSOMA. Then the permutation \( x_i \) is the \( i \)-th individual of the population \( P \), where \( |P| = \beta \).
and \( i \in (0, \ldots, \beta) \). Then the \( i \)-th individual can be written as \( N \) dimensional vector \( x_i = (x_{i0}, x_{i1}, x_{i2}, \ldots, x_{iN-1}) \), where \( x_{ij} \) is an individual’s element represented by jobID, where \( j \in (0, \ldots, N) \).

In case of Row-Major Format (RMF), all individuals are stored as a single vector \( v \) such that the first \( N \) elements of the vector \( v \) represent the individual \( x_0 \), the second \( N \)-tuple represents the second individual \( x_1 \) etc. Then

\[
v = (x_0^0, x_0^1, \ldots, x_0^{N-1}, x_1^0, x_1^1, \ldots, x_1^{N-1}, \ldots, x_{\beta-1}^0, x_{\beta-1}^1, \ldots, x_{\beta-1}^{N-1})
\]

In case of Column-Major Format (CMF), all individuals are stored as a single vector \( v \) such that the first \( \beta \) elements of the vector \( v \) represent all the first elements of all individuals \( x \in P \), the second \( \beta \)-tuple represents all the second elements of all individual \( x \in P \) etc. Then

\[
v = (x_0^0, x_1^0, \ldots, x_{\beta-1}^0, x_0^1, x_1^1, \ldots, x_{\beta-1}^1, \ldots, x_0^{N-1}, x_1^{N-1}, \ldots, x_{\beta-1}^{N-1})
\]

The RMF or CMF is selected according to inner implementation of CUDA kernels [6]. Figures 9, and 8 illustrate the successive computation of the same schedule in three different ways. In every table (index matrix \( M \)), the header row represents a job permutation, and the header column is a vector of indices of machines. Both vectors are indexed from zero in order to simplify this example. Next, all values \( M[i, j] \) represent iteration numbers in which the real schedule times \( [i, j] \) were computed, e.g. \( M[1, 0] \) and \( M[0, 1] \) in Figure 9 were computed in the second iteration indexed by 1 (zero-based indexing). Finally, Figure 9 illustrates parallel accesses into matrices by four CUDA threads (4 threads in a block), whereas Figure 8 shows a sequential computation processed by a single thread which will be described in more detail in the following section.

Having regard to the alignment of the memory, RMF was used in Figure 9 which ensures required coalesced access into global memory [6, 5, 16]. Contrariwise, CMF must be used in the case of Figure 8, because it is expected that every thread of the CUDA block will process a single schedule, thus the values \( M[i, j] \) of all schedules have to be aligned close to each other.

### 4.3.2 Data Level Parallelism

The above described memory alignment plays an important role for kernel design as well as arrangement of threads in blocks [15].

Obtaining schedules is the most time consuming part of DSOMA, especially in the case of local search, where a huge number of job permutations must be evaluated. Standard many-core architectures based on latest CPUs solve this problem by data division and distribution among several CPU cores. This is the first level of Data Parallelism (DLP) managed by SIMD architectures. In case of GPU utilization, this kind of data processing can be distributed to the level of individual
computation threads. The number of active threads that run in parallel will result in the number of processed schedules. Nevertheless, modern GPUs with support of OpenCL [24, 18, 11] or CUDA [5, 28] enable deeper and more complex data decomposition. Threads within a thread block can cooperate on a single schedule processing, they can share some intermediate data and reduce global memory accesses. Finally, better data parallelism enables higher occupancy of GPU chips that leads to the better computation performance.

Two CUDA kernels that compute individual schedules will be briefly introduced in the following text.

1. Single Thread Computation
2. Block/Warp Computation

NVIDIA Kepler architecture [19] introduced several new features that can significantly decrease kernel run-time. Shuffle instructions brought another way of sharing data among threads within the same warp, in addition to shared memory utilization. Using these instructions however has its limits, e.g. at most 32 threads (= warp size) can be affected by the shuffle instruction call, the number of used registers increases, thread/lane indexing can make some code parts more complex, etc. On the other hand, shuffle instructions can reduce the amount of utilized shared memory, eliminate thread synchronization barriers, or reduce total number of instructions with respect to thread data processing and warp data transfers. This can keep CUDA cores busy with memory accesses that have low latency and high bandwidth. The shuffle instructions were primarily used in case of Block/Warp Computation.

4.3.3 Single Thread Computation

In case of single thread computation (see Figure 8), it is supposed that a single thread will subsequently process one or more schedules. CUDA blocks can be designed to fit inner limits of a device, such as an optimal number of threads with respect to the number of used registers or amount of shared memory. This implementation is suitable especially during the 2OPT search because \( N^2 \) schedules must be computed in the worst case.

4.3.4 Block/Warp Computation

Such implementation represents the next level of parallelism, where a set of threads cooperates and computes a single schedule. Figure 9 illustrates a successive evaluation of cells in the grid of a schedule marked by zero-based indices. The same indices mark the cells that are processed in parallel. Let the number of jobs be \( J \). Then there is an inner loop of \( (J + B - 1)/B \) steps, where \( B \) is a block size; a block of 4 threads was used in this illustrative example. In every step, a strip of at most \( B \) columns is computed, such that all values in the last active column are called border values and are stored in the shared memory for the next step. If \( t \in \{0, 1, 2, 3\} \) is a
thread index and \(s\) is the zero-based step index, then a thread \(t\) computes the whole column \(s \times B + t\) in every step. In case of \(B < 32\), where 32 is the warp size [19], threads can store intermediate data in registers and share them by shuffle instructions to achieve the best performance.

A strip of \(B\) columns is computed in a loop of \(M + B - 1\) iterations. If a thread computes a single value (cell of a schedule), then it moves down and evaluates a new cell in the following iteration reusing previously computed value that was shuffled to subsequent thread simultaneously. The cells marked by number 3 were evaluated in the 4-th iteration, which is the first one, where all threads run the same instructions in parallel. Until then, only \(i + 1\) threads were active, where \(i \in \{0, 1, \ldots, M + B - 1\}\) is the zero-based iteration index. Next, all threads run parallel if \((B - 1) \leq i < M\). Finally, the threads subsequently finish their computations during the last \(B\) iterations. After that, the thread block shift right and process the next column strip. This strategy is suitable for faster processing of schedules with respects to GPU limits. As it was mentioned above, the usage of shuffle instructions needs CUDA blocks of at most 32 threads. Moreover, this number of threads limits the total number of active blocks. Although such implementation is more complicated, it finally brings significant performance improvement and it is more suitable for DSOMA in general.

![Fig. 8 Single thread computation](image1)

![Fig. 9 Block/Warp computation](image2)

## 5 Experiments

During the experiments, a fixed set of parameters was used. There are 1000 individuals in the DSOMA population, 300 migrations are computed, and the maximum number of trial individuals is 32. A 2-opt variant that stops after a better solution is found is used as a local search algorithm and it is applied to the current best solution, if it is not improved in 5 consecutive migrations. These setting may not be optimal for all problems. But in our experiments, we do not evaluate the DSOMA itself, but only its parallel behavior. Moreover, while randomness is a crucial part of the DSOMA execution, runs in our experiments are repeated 10 times and average numbers are presented as results. The experiments were performed on a flow shop instances from the extended TAIIARD SETS [17].
5.1 OPENMP Experiments

In the first experiment, the performance of the OPENMP solution that was described in Section 4.1 is evaluated. For this experiment, problems with varying size were used as an input. The varying size is important, because the size of the solved instance can affect the overall parallel performance. The measurements were performed on a computer with 84 cores (14 times 6 core Intel Xeon E5-4610 2.40GHz) and 1TB of shared memory.

Table 1 Execution times (measured in seconds) for the OpenMP implementation of DSOMA

<table>
<thead>
<tr>
<th>Problem</th>
<th>Machines</th>
<th>Jobs</th>
<th>1</th>
<th>2</th>
<th>4</th>
<th>8</th>
<th>16</th>
<th>32</th>
</tr>
</thead>
<tbody>
<tr>
<td>ta100</td>
<td>10</td>
<td>200</td>
<td>23.22</td>
<td>16.95</td>
<td>14.01</td>
<td>10.51</td>
<td>6.44</td>
<td>5.98</td>
</tr>
<tr>
<td>ta110</td>
<td>20</td>
<td>200</td>
<td>45.61</td>
<td>23.93</td>
<td>16.27</td>
<td>10.85</td>
<td>7.72</td>
<td>6.34</td>
</tr>
<tr>
<td>ta120</td>
<td>20</td>
<td>500</td>
<td>108.9</td>
<td>59.14</td>
<td>38.61</td>
<td>22.00</td>
<td>12.57</td>
<td>10.67</td>
</tr>
<tr>
<td>ta130</td>
<td>50</td>
<td>500</td>
<td>243.56</td>
<td>146.28</td>
<td>76.78</td>
<td>35.98</td>
<td>23.01</td>
<td>15.98</td>
</tr>
<tr>
<td>ta140</td>
<td>20</td>
<td>700</td>
<td>287.15</td>
<td>170.40</td>
<td>83.92</td>
<td>44.91</td>
<td>25.57</td>
<td>18.31</td>
</tr>
<tr>
<td>ta150</td>
<td>50</td>
<td>700</td>
<td>443.92</td>
<td>233.48</td>
<td>128.95</td>
<td>72.04</td>
<td>39.65</td>
<td>23.97</td>
</tr>
<tr>
<td>ta160</td>
<td>20</td>
<td>1000</td>
<td>529.37</td>
<td>182.93</td>
<td>116.36</td>
<td>70.80</td>
<td>34.25</td>
<td>21.24</td>
</tr>
<tr>
<td>ta170</td>
<td>50</td>
<td>1000</td>
<td>633.93</td>
<td>368.64</td>
<td>213.18</td>
<td>114.96</td>
<td>64.09</td>
<td>39.26</td>
</tr>
</tbody>
</table>

The average execution times for chosen problems (along with their sizes) are summarized in Table 1. The relative speedup and relative efficiency are captured in Figure 10. As defined in [9], the relative speedup on \( p \) processes (threads in our case) is a ratio between the execution time on one processor and the execution time on \( p \) processors. Similarly relative efficiency is the relative speedup divided by the number of processes. From the experiment results, it can be observed, that the parallel implementation is suitable for larger instances, where the amount of computation overcomes the overhead introduced by parallel execution. The main parallel bottleneck in our solution is the concurrent memory access. To further optimize the memory usage, some fundamental changes to the current implementation needs to be made. Considering the used hardware, it is very hard to achieve the efficiency close to 1 with growing number of threads. Still, for larger instances, the efficiency remained close to 50% even for 32 threads. In absolute numbers, the computational time for the largest instance was reduced by nearly 600s from 634s to 40s.

5.2 CUDA Experiments

The second set of experiments was focused on the CUDA implementation of DSOMA. The measurements were performed on a computer with the following specification: AMD FX(tm)-8150 Eight-Core Processor, 3.61 GHz, 32 GB RAM, Windows 64-bit and a single GPU NVIDIA GeForce GTX 970, 4 GB GDDR5 RAM, 13 SMx, Maxwell architecture with CUDA compute capability 5.2.
As it was aforementioned, the data alignment and Single or Warp/Block computation strategy plays an important role in the process of evaluation of time schedules. Hence the first GPU results (see Table 2) illustrate performances of different computation strategies only as a part of the whole DSOMA algorithm. 300 schedules were evaluated in parallel for every input data (row in the Table 2). According to expectations, the greater is the number of jobs, the Block/Warp computation gives better results with comparison to Single Thread Strategy. Then the Column Major Format (CMF) data alignment providing coalesced memory access enables the computation time reduction.

The next experiments were focused on the total computation time with the same settings as in the case of Table 1. The Block/Warp strategy was used to evaluate all individual schedules, whereas Single Thread strategy was used during 2Opt local search phase. Nevertheless, the computation time covers the whole DSOMA runtime, i.e. generating individuals and trials, repairing trials, 2Opt search, searing for leader, etc. All partial steps were implemented with the usage of CUDA in the form of individual kernels, and no additional data transfers between host and device were needed. All runtimes are given in Table 3.
Table 3  Execution times (measured in seconds) for the CUDA implementation DSOMA

<table>
<thead>
<tr>
<th>Problem</th>
<th>Machines</th>
<th>Jobs</th>
<th>Time (sec.)</th>
</tr>
</thead>
<tbody>
<tr>
<td>ta100</td>
<td>10</td>
<td>200</td>
<td>3.68</td>
</tr>
<tr>
<td>ta110</td>
<td>20</td>
<td>200</td>
<td>4.75</td>
</tr>
<tr>
<td>ta120</td>
<td>20</td>
<td>500</td>
<td>10.03</td>
</tr>
<tr>
<td>ta130</td>
<td>50</td>
<td>500</td>
<td>12.79</td>
</tr>
<tr>
<td>ta140</td>
<td>20</td>
<td>700</td>
<td>13.60</td>
</tr>
<tr>
<td>ta150</td>
<td>50</td>
<td>700</td>
<td>16.14</td>
</tr>
<tr>
<td>ta160</td>
<td>20</td>
<td>1000</td>
<td>24.64</td>
</tr>
<tr>
<td>ta170</td>
<td>50</td>
<td>1000</td>
<td>22.68</td>
</tr>
</tbody>
</table>

5.3 Distributed DSOMA Experiment

The final experiment was performed with the distributed variant of DSOMA (described in Section 4.2). ANSELM, a supercomputer from IT4INNOVATIONS$^{12}$ was used for this experiment. ANSLEM$^{13}$ is a cluster composed from 209 computing nodes, where each node contains two Intel Sandy Bridge E5-2665, 8-core, 2.4GHz processors and 64 GB of physical memory.

In this experiment, 16 islands were used. On every 5-th migration, these islands exchanged their current best solutions with their neighbours. Figure 11 captures the progress in computation and shows the fitness values of the current best solution for every island in time (time is measured in seconds).

6 Conclusion

In this chapter, different approaches as to how to take advantage of the parallel computing for DSOMA were explored. From all possible variants, three distinguished options were chosen and implemented. For the first variant, OPENMP which is a relatively easy to use technology at it is suitable even for common desktop computers with a multi-core processor was used. Based on performed experiments, we can conclude that it is possible to achieve meaningful speedup, but the efficiency drops down with growing number of processors. While it was relatively easy to get the first parallel solution with this technology, it still requires a lot of tuning to get a good parallel behaviour for a larger number of processors.

The second implemented solution uses the general purpose programming on GPUs supporting CUDA. This technology can be successfully applied to speed up the computation of tasks like evolutionary algorithms. Also for DSOMA, we were able to significantly reduce the overall computational time. Still, it requires a lot of additional work to use this technology and it requires a skilled CUDA programmer.

$^{12}$ http://www.it4i.cz/
$^{13}$ https://docs.it4i.cz/anselm-cluster-documentation/hardware-overview
In our case, the original sequential solution was in fact completely rewritten. Furthermore, it is closely tied to the used hardware and it can be hard to use it on a different GPU.

Previous solutions implement the original DSOMA, where the parallel behaviour is added to the architecture. The last solution adds a new layer (a set of distributed islands) that makes the algorithm suitable for distributed computing. This chapter focuses on parallelization of DSOMA. It does not argue about the quality of obtained results. This is especially true for the distributed variant. It adds new parameters like exchange rate or a number of islands and these parameters can affect the quality of obtained results. Such solution can be executed on hundreds of processors, but to use them meaningfully, it still requires a lot of testing and tuning.

Acknowledgements This work was supported by the IT4Innovations Centre of Excellence project (CZ.1.05/1.1.00/02.0070), funded by the European Regional Development Fund and the national budget of the Czech Republic via the Research and Development for Innovations Operational Programme, as well as Czech Ministry of Education, Youth and Sports via the project Large Research, Development and Innovations Infrastructures (LM2011033). The work was also partially supported by Grant of SGS No. SP2015/146, VŠB - Technical University of Ostrava, Czech Republic.

References

Abstract—In this paper, a massively parallel implementation of Common Tensor Discriminant Analysis is presented with applications to human brainwave pattern recognition. The implementation, accelerated by the NVIDIA Compute Unified Device Architecture technology, is shown to be 11.49x faster than the original MATLAB version. Before processing by the discriminant analysis, the data is segmented by a sliding window and converted into the time-frequency domain by the continuous wavelet transform.

Keywords—tensor; CTDA; BCI; pattern matching; parallelism

I. INTRODUCTION

Brain computer interfaces (BCI) are still considered to be the holy grail of human-computer interaction and as such are given a significant amount of attention and research efforts. With the variety of applications ranging from simple interaction with personal computers to aiding systems for disabled people to extensive military use, BCIs also have a tremendous potential in business. While most papers and researches focus on increasing the recognition accuracy of various brainwave classification techniques, it is just as necessary to be able to perform the classification as fast as possible with the minimum cost of the hardware used. In this paper, a massively parallel implementation of the Common Tensor Discriminant Analysis (CTDA) preprocessing algorithm is presented and compared to its original MATLAB implementation in terms of computational performance. To the best of the authors’ knowledge, the algorithm has not yet been implemented using a massively parallel paradigm.

II. SIGNAL PREPROCESSING

To improve recognition accuracy and sometimes computation speed, data is usually preprocessed prior to classification. The classic approach to preprocessing is signal decomposition, data factorization, feature extraction and selection etc. One of the most fundamental concepts in the computer science is the Fast Fourier Transform (FFT) [1], [2]. It is universal in signal processing but can also be used, among other things, to compress image and audio files, solve differential equations and price stock options [3]. Other methods use the principles of signal decomposition and approaches similar to FFT. Here, wavelet transforms, one of which is also used for data preprocessing in this paper, should be mentioned first with its wide use in different areas [4], [5]. The Hilbert-Huang Transform (HHT) can be mentioned as a member of the class of algorithms that decompose a signal into so-called intrinsic mode functions (IMF) and obtain instantaneous frequency data. It is designed to work well for non-stationary and non-linear data. In contrast to other common transforms like the FFT, the HHT is more of an algorithm (an empirical approach) that can be applied to a data set rather than a theoretical tool [6].

III. MASSIVE PARALLELISM

Throughout the last decade it became clear that modern graphics processing units (GPUs) can perform many tasks better than classic central processing units (CPUs) by splitting the task into many smaller, usually identical tasks executed in parallel. This led to the development of several software technologies that enable programmers to harness the parallel computing power of modern GPUs. The most notable among them are OpenCL by Khronos Group [7], [8], CUDA by NVIDIA [9], [10] and DirectCompute by Microsoft [11], [12]. This research uses the NVIDIA CUDA massively parallel paradigm.

A. Compute Unified Device Architecture

In order to utilize massive parallelism in various tasks, Compute Unified Device Architecture (CUDA) uses a structure of threads that are, as opposed to classic sequential programs, executed simultaneously. These threads are organized into three-dimensional blocks which are in turn organized into two-dimensional grids. While blocks can contain at most 512 threads, it is not defined which dimension should contain them. It is acceptable for the architecture to distribute the threads into the three-dimensions without limitation as long as the total number of threads in a single block does not exceed 512. While it is perfectly viable to create a block of 16 x 16 x 1 threads, one cannot specify a...
block whose dimensions are 32 x 32 x 1 as the total number of threads in this case is 1024. Grids do not constrain the number of blocks they can support as long as neither of the two dimensions exceed the value of 65,535. Overall, single multiplication makes it clear that any grid can contain up to 2,198,956,147,200 threads. That does not, however, mean that all these threads could be computed all at once. For details on how threads are distributed for computing and how they are executed, the reader is encouraged to further study the matter in, for example, [13].

IV. CTDA

Originally presented by Zhao et al. in [14], CTDA has been successfully applied in BCI systems achieving recognition accuracy of up to 95%. An overall description of the CTDA algorithm is provided in this section. It explains the tensor representation of an EEG signal, formally describes the CTDA feature extraction process and points out where exactly is the algorithm suitable for massive parallelism.

A. EEG Tensor Representation

Suppose that there is an electroencephalographic (EEG) epoch represented as a \( N_c \times N_t \) matrix \( \mathbf{X} \in \mathbb{R}^{N_c \times N_t} \) where \( N_c \) is the number of channels used for registration and \( N_t \) is the number of samples. Then, in order to capture the time-frequency structure of the signal the epoch \( \mathbf{X} \) can be transformed into tensor \( \mathbf{X} \in \mathbb{R}^{N_c \times N_f \times N_t} \) using Continuous Wavelet Transform (CWT) of its channels, i.e.,

\[
\mathbf{X}(c,:,:) = \text{CWT}(\mathbf{X}(c,:)), \ c = 1, \ldots, N_c
\]

when MATLAB notation is followed. Here \( N_f \) is the number of scales for which wavelet coefficients are computed.

Notion of a sample covariance matrix can be also generalized for tensor representation by defining a covariance tensor \( \mathcal{R} \) as \( \frac{1}{N_t} \mathbf{X} \odot \mathbf{X}^T \) with \( \mathbf{X}^T \) denoting a tensor from \( \mathbb{R}^{N_f \times N_t} \) for which \( \mathbf{X}^{\text{flat}} = \mathbf{X}^\text{flat} \) (note the permutation of \( \mathcal{R} \) dimensions). The elements of covariance tensor are

\[
\mathcal{R}_{i,m,n,t} = \frac{1}{N_t} \sum_{x} \mathbf{X}_{i,m,x}, \mathbf{X}_{j,n,x}. \quad (2)
\]

The notion of covariance tensor plays an important role in CTDA.

B. CTDA Problem Formulation and Feature Extraction

In general there is a set of several EEG records, \( \mathbf{X}^l, l = 1, \ldots, L \), attributed to different classes. In practice \( \mathbf{X}^l \) are obtained during a training session and are readily available. In order to extract features using CTDA records \( \mathbf{X}^l \) are split into epochs \( \mathbf{X}^l_i, i = 1, \ldots, N_l \), whose lengths are equal and determine how often the classifier output is generated. The epochs are then transformed into tensors \( \mathbf{X}^l_i \) as described in the previous subsection. For each of the tensors \( \mathbf{X}^l_i \) its covariance tensor \( \mathcal{R}^l_i \) is computed. These covariance tensors can be used to estimate the ”typical” covariance tensor for the \( l \)-th class as

\[
\mathcal{R}^l = \frac{1}{N_l} \sum_l \mathcal{R}^l_i. \quad (3)
\]

Given the estimates of covariance tensors of each class the CTDA problem is to find matrices \( \mathbf{W}^l_1 \) and \( \mathbf{W}^l_2, l = 1, \ldots, L \), (two matrices for each class) projecting the signal corresponding to performing the \( l \)-th class on the first and second modes respectively so that

\[
\mathcal{R}^l \times_1 (\mathbf{W}^l_1)^T \times_2 (\mathbf{W}^l_2)^T \times_3 (\mathbf{W}^l_2)^T \times_4 (\mathbf{W}^l_1)^T = \mathcal{D}^l, \quad (4)
\]

\[
(\sum_l \mathcal{R}^l) \times_1 (\mathbf{W}^l_1)^T \times_2 (\mathbf{W}^l_2)^T \times_3 (\mathbf{W}^l_2)^T \times_4 (\mathbf{W}^l_1)^T = \mathcal{J}^l \quad (5)
\]
and

\[ D_{i,j,k,l}^l = d_i d_j \delta_{i,j} \delta_{k,l}, \]
\[ J_{i,j,k,l}^l = \delta_{i,l} \delta_{j,k}. \]  

(6)

Once the matrices \( W_{l,1} \) and \( W_{l,2} \) are obtained for each \( l \), feature extraction can be performed as follows. Let \( Y \) be an EEG epoch represented as a tensor. First, \( Y \) is projected using matrices \( W \) for each class yielding \( L \) tensors

\[ Z^l = Y \times_1 \left( W^1 \right)^T \times_2 \left( W^2 \right)^T \]

(7)

After that, vectors \( v_i, i = 1, L \), are calculated as \( v_i^T = \text{diag} (\text{mat3} (Z^l) \text{mat3} (Z^l)^T) \). These vectors are concatenated so that \( v^T = v_1, \ldots, v_L \) and the resulting vector is log-transformed \( v = \log(v) \) component-wise. The vector \( v \) is then the feature vector corresponding to the epoch \( Y \). Complete formal specification of the CTDA process can be found in [14]. In the testing phase, equation 7 is computed for every test epoch, thus generating feature vectors to classify. The classifier itself was based on Euclidean distance.

C. Parallel implementation

Generally, all the equations mentioned above can be computed in parallel. The most time consuming parts lie in tensor to matrix multiplications which can be transformed into matrix to matrix multiplication with respect to appropriate tensor unfolding. cuBLAS routines (GEMM) [15] were used in our implementation to process such multiplications. Next, many hidden operations were done in parallel as well, e.g. sum of elements on a matrix diagonal, square roots of these vectors are concatenated so that \( v^T = v_1, \ldots, v_L \) and the resulting vector is log-transformed \( v = \log(v) \) component-wise. The vector \( v \) is then the feature vector corresponding to the epoch \( Y \). Complete formal specification of the CTDA process can be found in [14]. In the testing phase, equation 7 is computed for every test epoch, thus generating feature vectors to classify. The classifier itself was based on Euclidean distance.

Definition 1: (Unfolding): The Mode-\( n \) unfolding of tensor \( Y \in \mathbb{R}^{I_1 \times I_2 \times \cdots \times I_n} \) is denoted by \( Y_{(n)} \) and arranges the Mode-\( n \) fibers into columns of a matrix. More specifically, a tensor element \( (i_1, i_2, \ldots, i_N) \) maps onto a matrix element \( (i_n, j) \), where

\[ j = 1 + \sum_{p \neq n} (i_p - 1) J_p, \quad \text{with} \]
\[ J_p = \begin{cases} 1, & \text{if } p = 1 \text{ or } p = 2 \text{ and } n = 1, \\ \prod_{m \neq n} I_m, & \text{otherwise.} \end{cases} \]  

(8)

See Figure 2 that demonstrates Mode-1 unfolding.

Template function was used according to best practices in CUDA. This allows adjusting the compiled code with respect to target unfolding mode, e.g. Mode-1 \( \rightarrow \) 2 or Mode-1 \( \rightarrow \) 3 (see lines 23-30). The proposed method maximizes CUDA register utilization, avoids time-consuming modulo operations and minimizes access into global memory space. Moreover, every CUDA thread block processes several consequent data chunks (\text{noChunks}) which is constant and known in advance. Thus, the \text{for} loop can be completely unrolled (see line 13).

1 \template<typename T, unsigned int m>
2 \_\_global\_ _void unfoldFromModel_kernel(
3 \const T* data, \const dim3 dim,
4 \const unsigned int noChunks, T* m)
5 {
6 \const unsigned int dimXY = dim.x * dim.y;
7 \const unsigned int dimXZ = dim.x * dim.z;
8 \const unsigned int totalDim = dimXY * dim.x;
9
10 \unsigned int x, y, z, tmp;
11 \unsigned int di = blockIdx.x * \text{THREADS_PER BLOCK} + \text{threadIdx.x};
12
13 \#pragma unroll
14 \for (unsigned int i = 0; i < noChunks; i++)
15 {
16 \if (di < totalDim)
17 {
18 \x = z / \text{dimX};
19 \y = y / \text{dim.y};
20 \if (mode == 2)
21 \{ m[z * \text{dimX} + x * \text{dim.y} + y] = data[di]; \}
22 \} \else \break;
23 \}
24 \} \else \break;
25 \}
26 \} \else \break;
27 \}
28 \endloop can be completely unrolled.

Listing 1. CUDA kernel - Tensor Mode-1 unfolding

The overall workflow of the entire process of data collection, training and testing illustrated in Figure 1.

V. EXPERIMENTS

This section describes the dataset and the overall experimental setting, summarizes the experiments performed and presents the resulting computation times as well as accuracy. The experiments were performed on a personal computer composed of 32 GB RAM, 8-core AMD FX-8150 3.6Ghz and Tesla C2075 GPU. The used software equipment was 64b Windows 8 Professional operating system, CUDA 5.0 and the latest versions of cuBLAS, CULA and MATLAB.

A. Data collection

The experimental data was recorded in Prague in the Institute of Computer Science of the Academy of Sciences of the Czech Republic. The records were taken from a 26 year
old male right handed subject with no known neurological
diseases. The device used to capture the signal was g.Tec
g.USBamp with 48 channels placed on the following posi­
tions on the subject’s head defined by the 10-20 standard:
Fz, F2, F1, F4, F3, Fc1, Fc2, Fc3, Fc4, Cz, C2, C1, C4,
C3, Cpz, Cp1, Cp2, Cp3, Cp4, Pz, P2, P1, P4, P3, Poz, P03,
P04, Oz, O2, O1, P08, P07, P6, P5, P5, P6, P5, C6, C5, C6,
Fc5, F6, F5, A14, A13, P8, F7, IZ [21], [22]. It operated
on the frequency of 256 Hz and the 50Hz notch in the
obtained signal was filtered out with the Butterworth filter.
Before passing through CTDA, the data was preprocessed
with CWT as described in section IV. The used wavelet
was mexican hat and the 15 chosen scales correspond to the
frequencies of 4 through 40 Hz.

During the experiment the subject had to either relax,
imagine slow contraction of his left or right palm or imagine
slowly pressing his foot against a pedal, thus having 4 tasks
to perform. He was sitting in a comfortable chair in front
of a computer monitor with a circle drawn in the center of the
screen the purpose of which was to fixate the subject’s gaze.
The beginning of the task to perform was clued by changing
the color of one of the four gray arrows drawn around
the circle into green which is a common practice when
acquiring EEG data [23], [24], [25]. The session was split
into 8 periods, each period containing all tasks presented at
a random order. The subject had to keep performing each
task for 10 seconds so the experiment lasted 320 seconds.

B. Data arrangement

Overall, 144384 time samples were recorded from each of
the sensors, some of which were not part of the experiment
and could be omitted. The samples were windowed into 1
second long epochs (256 values) with the window shift of
0.25s. This created 2553 labelled epochs, 451 of which were
used to train CTDA, the rest for testing. After the CWT
processing, the training tensor’s dimensions were \(48 \times 15 \times 115456\), the testing tensor’s \(48 \times 15 \times 461312\).

C. Results

Given the tensor dimensions, computing CTDA and
generating feature vectors from the testing set was time­
suming enough for it to make sense to measure the
experiments in seconds. Table I shows the measured results.
The first row represents the number of iterations required
for the CTDA algorithm to converge. Although the same
number of iterations for both implementations is expected,
the difference can be explained through the different im­
plementations of SVD in CULA and MATLAB. Still, the
computation times are generally significantly lower. The
speed-up of computing each iteration in the training phase
reached the factor of 11.49 while the speed-up of generating
test vectors for classification was lower at 5.68x. The lower
speed-up is easily explained by the fact that generating test
vectors is a much simpler operation of simply extracting
the features from test signals, thus weakening the CUDA
parallelism advantage and, at the same time, increasing the
significance of unavoidable memory transfer delays. Still,
more than 5 times faster processing is satisfactory as this
increase in performance allows real-time classification.

VI. CONCLUSION

This paper reviewed the performance of a parallel im­
plementation of the CTDA preprocessing technique in ap­
lications to EEG wave recognition. It was shown that a
simple change of the programming paradigm can greatly
benefit the computation speed of both the CTDA algorithm
itself and extracting features for classification. The main
advantage of the paradigm is that it is widely available even
in low-cost personal computers and should be made available
for mobile devices in the near future as well. The main

<table>
<thead>
<tr>
<th>Phase</th>
<th>Measurement</th>
<th>Implementation</th>
<th>Speed-up</th>
</tr>
</thead>
<tbody>
<tr>
<td>Training</td>
<td>Feature vectors</td>
<td>CUDA: 451</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Iterations</td>
<td>MATLAB: 451</td>
<td>11.49x</td>
</tr>
<tr>
<td></td>
<td>Total Time (s)</td>
<td>CUDA: 350.41</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>MATLAB: 1207.76</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Iteration time</td>
<td>CUDA: 11.68</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>MATLAB: 134.20</td>
<td></td>
</tr>
<tr>
<td>Testing</td>
<td>Feature vectors</td>
<td>CUDA: 1802</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Total Time (s)</td>
<td>MATLAB: 1802</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>CUDA: 45.161</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>MATLAB: 250.39</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Iteration time</td>
<td>CUDA: 24.48</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>MATLAB: 138.95</td>
<td></td>
</tr>
</tbody>
</table>

Figure 2. An illustration of tensor Mode-1 unfolding [20].
disadvantage is more complicated practical programming. Possible directions for future work include decreasing the computational complexity of the algorithm to be computable on low-power and mobile devices, allowing for the algorithm to be implemented in easily portable and possibly even wearable EEG solutions.

ACKNOWLEDGMENT

This article has been elaborated in the framework of the IT4Innovations Centre of Excellence project, reg. no. CZ.1.05/1.1.00/02.0070 funded by Structural Funds of the European Union and state budget of the Czech Republic. The work is partially supported by Grant of SGS No. SP2013/70, VŠB - Technical University of Ostrava, Czech Republic. This work was also supported by the Bio-Inspired Methods: research, development and knowledge transfer project, reg. no. CZ.1.07/2.3.00/20.0073 funded by Operational Programme Education for Competitiveness, co-financed by ESF and state budget of the Czech Republic.

REFERENCES


Serum protein fingerprinting by PEA immunoassay coupled with a pattern-recognition algorithms distinguishes MGUS and multiple myeloma

Petra Schneiderova1,*, Tomas Pika2,*, Petr Gajdos3, Regina Fillerova1, Pavel Kromer3, Milos Kudelka3, Jiri Minarik2, Tomas Papajik2, Vlastimil Scudla2, Eva Kriegova1

1 Department of Immunology, Faculty of Medicine and Dentistry, Palacky University Olomouc, Olomouc, Czech Republic
2 Department of Hemato-Oncology, Faculty of Medicine and Dentistry, Palacky University and University Hospital, Olomouc, Czech Republic
3 Department of Computer Science, Faculty of Electrical Engineering and Computer Science, VSB-Technical University of Ostrava, Ostrava, Czech Republic
*These authors have contributed equally to this work

Correspondence to: Eva Kriegova, email: eva.kriegova@email.cz

Keywords: serum pattern, cytokines, growth factors, proximity extension immunoassay, post-transplant serum pattern

Received: May 04, 2016 Accepted: July 28, 2016 Published: August 12, 2016

ABSTRACT

Serum protein fingerprints associated with MGUS and MM and their changes in MM after autologous stem cell transplantation (MM-ASCT, day 100) remain unexplored. Using highly-sensitive Proximity Extension ImmunoAssay on 92 cancer biomarkers (Proseek Multiplex, Olink), enhanced serum levels of Adrenomedullin (ADM, \(P_{corr} = .0004\)), Growth differentiation factor 15 (GDF15, \(P_{corr} = .003\)), and soluble Major histocompatibility complex class I-related chain A (sMICA, \(P_{corr} = .023\)), all prosurvival and chemoprotective factors for myeloma cells, were detected in MM comparing to MGUS. Comparison of MGUS and healthy subjects revealed elevation of angiogenic and anti-apoptotic midkine (\(P_{corr} = .0007\)) and downregulation of Transforming growth factor beta 1 (TGFB1, \(P_{corr} = .005\)) in MGUS. Importantly, altered serum pattern was associated with MM-ASCT compared to paired MM at the diagnosis as well as to healthy controls, namely by upregulated B-Cell Activating Factor (sBAFF) (\(P_{corr} < .006\)) and sustained elevation of other pro-tumorigenic factors. In conclusion, the serum fingerprints of MM and MM-ASCT were characteristic by elevated levels of prosurvival and chemoprotective factors for myeloma cells.

INTRODUCTION

Monoclonal gammopathy of undetermined significance (MGUS) is a precursor lesion to overt multiple myeloma (MM), a clonal B-cell malignancy characterized by excessive multiplication of a plasma cell clone(s) in bone marrow, and accumulation of either a monoclonal immunoglobulin (Ig) (M-protein) or an Ig-free light chain in blood [1]. During the last decade, MM treatment and patient outcomes improved remarkably after the introduction of novel agents and autologous stem cell transplantation (ASCT) [2]. However, even with the best combination of currently available drugs, a cure is not achieved for most MM patients [2, 3]. Better characterization of neoplastic cells and microenvironment in particular myeloma stages is therefore needed as well as clarifying of reason(s) for treatment failure in most MM patients [2, 3, 4]. The neoplastic plasma cells in MGUS and MM share similar genetic abnormalities, probably occurring as early events [5, 6]. The key role in microenvironment play bone marrow stromal cells and other microenvironmental cells that secrete a plethora of cytokines and growth factors after paracrine stimulation and/or direct interaction with neoplastic cells [7]. Moreover, also myeloma cells secrete numerous cytokines and growth factors [8, 9]. The secreted molecules may, in turn, promote homing, migration, proliferation, survival of malignant plasma cells as well as contribute to the bone resorption and drug resistance [10].

Given the key role of cytokines and growth factors in MM pathogenesis, we investigated the complexity of serum microenvironment using novel multiplex highly-
sensitive PEA immunoassay on 92 cancer-related proteins followed by pattern-recognition analyses. Besides identification of serum fingerprints distinguishing MGUS and MM, we for the first time compared paired MM samples from the time of diagnosis and after autologous stem cell transplantation (MM-ASCT) as well as MGUS to healthy subjects.

RESULTS

Serum protein fingerprinting in MGUS and MM by PEA immunoassay

To assess the serum protein fingerprints associated with MGUS and MM, we compared serum protein pattern obtained by PEA immunoassay in MGUS and MM and healthy controls. Of ninety-two analyzed biomarkers (Supplementary Table S1), levels of six biomarkers (sBTC, CA242, sER, GM-CSF, IL2, IL4) were below the Proseek limit of detection (LOD) in all studied groups and were therefore excluded from further analysis. Comparing MGUS and MM, 26 analytes were deregulated between these groups, whereas 13 analytes reached the significance after the adjustment for multiple comparisons (Supplementary Table S2A). The distribution of serum levels of top-ranked proteins (ADM, TRAP, GDF15, suPAR, REG4, TGFBI1, sMICA, IL1RA, HE4, sHGFR, sVEGFA; see Table 1A), all found upregulated in MM, is shown in Figure 1. The protein serum fingerprints associated with MGUS and MM and the changes in protein levels between MGUS and MM for top-deregulated analytes are shown in Figure 2A.

Comparison of protein pattern obtained in MGUS and healthy controls revealed deregulation of 33 proteins (Figure 2B), of these 21 reached significance after multiple comparisons (Supplementary Table S2B). The protein levels of top-ranked proteins (midkine, THPO, sTNFRI SF4, sHER4, INFγ, TGFBI1, sPECAM1, sIL17RB, KLK6, suPAR) are presented in Table 1B and Supplementary Figure S1A.

When comparing MM and controls, we observed deregulation of 46 serum proteins (Figure 2C), of these 41 reached significance after adjustment for multiple comparisons (Supplementary Table S2C). The distribution of serum levels of top-ranked proteins between MM and controls (PGF, GDF15, HE4, sTNFR2, CSF1, midkine, sPECAM1, CCL19, sVEGFA, INFγ; see Table 1C) is shown in Supplementary Figure S1B. The subanalysis based on cytogenetic/FISH analysis was not performed due to the high heterogeneity within the group.

Changes in serum protein pattern in post-transplant MM

To assess the changes in serum protein pattern in MM after ASCT, we compared the post-transplant sera (day 100) with paired samples obtained in MM patients at the time of diagnosis and healthy control subjects. Comparing paired samples from MM-ASCT and MM, the most upregulated protein in post-transplant sera was sBAFF (Pcorr = .006), followed by CXCL9 (Pcorr = .041). Next twenty-one proteins were downregulated (14 proteins after adjustment for multiple comparisons) in MM-ASCT comparing to MM (Figure 2D, Supplementary Table S2D). The top-ranked proteins were: elevated sBAFF and downregulated REG4, sPECAM1, sIL6R, sPDGFβ, midkine, sHGFR, TGFBI1, sAREG, and sMICA in MM-ASCT comparing to MM (Table 1D, Figure 3). Importantly, serum levels of MM-associated pro-tumorigenic factors such as GDF15, CSF1, suPAR, and HE4 did not change after ASCT comparing to sample at the diagnosis (Supplementary Table S2D).

To exclude the influence of treatment regime on serum pattern, we assessed the protein profile in subgroups based on ASCT induction regime (IMiD-based/ bortezomib-based). We did not detect any differences in the cytokine levels as a function of the induction regime as well as the hematological response (CR, VGPR/PR) on day 100 (Pcorr > .05).

Comparing to healthy subjects, the serum of post-transplant MM patients showed permanently altered pro-tumorigenic signature characteristic by deregulation of 35 proteins (after multiple adjustments: 28 analytes) (Figure 2E, Supplementary Table S2E). Except sTGFA and TGFBI1, all deregulated proteins were elevated in MM-ASCT. The top-ranked proteins between MM-ASCT and healthy controls were as follows: sBAFF, CSF1, sTGFA, TRAP, CXCL10, sTNFR2, sTNFRSF4, Flt3L, GDF15, HE4, THPO (Table 1E, Supplementary Figure S1C). The serum protein pattern in MM-ASCT and its comparison to those of healthy controls, MGUS and MM, are presented in Figure 4.

Pattern-recognition algorithms

To facilitate the selection of the most promising circulating proteins distinguishing studied groups (MGUS, MM, MM-ASCT, healthy subjects), we applied advanced binary classification algorithm and analyzed co-occurrence of analytes in classification models. The most accurate classification model for separation of MGUS and MM utilized in classification rules most frequently sMICA in combination with other analytes (Figure 5A). In MGUS vs healthy controls, the classification rules used most often TGFBI1 and midkine (Figure 5B) and in MM vs healthy controls most often sMICA, CXCL11, and midkine (Figure 5C). The classification model for MM and MM-ASCT used in the classification rules most frequently sBAFF and CCL21 (Figure 5D) and for MM-ASCT and controls used sTGFA and sBAFF (Figure 5E).
Table 1: Serum levels of top-ranked proteins differentiating between A) MGUS vs MM, B) healthy controls vs MGUS, C) healthy controls vs MM, D) MM vs MM-ASCT and E) healthy controls vs MM-ASCT.

<table>
<thead>
<tr>
<th>Analyte</th>
<th>Mean Linear ddCq (95% CI)</th>
<th>FC</th>
<th>( P )</th>
<th>( P_{corr} )</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>A</strong></td>
<td><strong>MGUS</strong></td>
<td><strong>MM</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td>ADM</td>
<td>46.9 (28.4-65.4)</td>
<td>191 (121-262)</td>
<td>2.80</td>
<td>( 4.0 \times 10^{-6} )</td>
</tr>
<tr>
<td>TRAP</td>
<td>28.9 (22.3-35.6)</td>
<td>85.1 (42.8-127)</td>
<td>2.33</td>
<td>( 3.7 \times 10^{-5} )</td>
</tr>
<tr>
<td>GDF15</td>
<td>12.6 (9.19-16.0)</td>
<td>53.3 (22.5-84.1)</td>
<td>2.72</td>
<td>( 1.1 \times 10^{-4} )</td>
</tr>
<tr>
<td>suPAR</td>
<td>295 (251-338)</td>
<td>486 (368-603)</td>
<td>1.60</td>
<td>( 2.7 \times 10^{-4} )</td>
</tr>
<tr>
<td>REG4</td>
<td>5.96 (5.46-6.46)</td>
<td>9.58 (6.78-12.4)</td>
<td>1.30</td>
<td>( 1.2 \times 10^{-3} )</td>
</tr>
<tr>
<td>TGFB1</td>
<td>42.9 (39.0-46.8)</td>
<td>85.5 (43.7-127)</td>
<td>1.41</td>
<td>( 1.9 \times 10^{-3} )</td>
</tr>
<tr>
<td>sMICA</td>
<td>15.4 (9.98-20.8)</td>
<td>39.4 (26.8-52.1)</td>
<td>2.29</td>
<td>( 2.1 \times 10^{-3} )</td>
</tr>
<tr>
<td>IL1RA</td>
<td>9.22 (7.23-11.2)</td>
<td>60.0 (0-145)</td>
<td>1.71</td>
<td>( 2.6 \times 10^{-3} )</td>
</tr>
<tr>
<td>HE4</td>
<td>18.4 (15.6-21.3)</td>
<td>89.8 (0-190)</td>
<td>1.72</td>
<td>( 3.0 \times 10^{-3} )</td>
</tr>
<tr>
<td>sHGFR</td>
<td>259 (240-278)</td>
<td>580 (199-960)</td>
<td>1.26</td>
<td>( 3.0 \times 10^{-3} )</td>
</tr>
<tr>
<td>sVEGFA</td>
<td>892 (728-1056)</td>
<td>1628 (1082-2173)</td>
<td>1.47</td>
<td>( 3.0 \times 10^{-3} )</td>
</tr>
<tr>
<td><strong>B</strong></td>
<td></td>
<td><strong>healthy controls</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Midkine</td>
<td>30.3 (25.8-34.8)</td>
<td>75.0 (60.8-89.3)</td>
<td>2.24</td>
<td>( 8.2 \times 10^{-6} )</td>
</tr>
<tr>
<td>THPO</td>
<td>13.1 (11.7-14.6)</td>
<td>24.2 (18.8-29.6)</td>
<td>1.79</td>
<td>( 9.8 \times 10^{-5} )</td>
</tr>
<tr>
<td>sTNFRSF4</td>
<td>4.12 (3.29-4.94)</td>
<td>7.39 (6.05-8.72)</td>
<td>1.54</td>
<td>( 1.6 \times 10^{-4} )</td>
</tr>
<tr>
<td>sHER4</td>
<td>174 (163-184)</td>
<td>270 (228-311)</td>
<td>1.45</td>
<td>( 2.4 \times 10^{-4} )</td>
</tr>
<tr>
<td>IFNγ</td>
<td>1.58 (1.52-1.64)</td>
<td>2.17 (1.95-2.39)</td>
<td>1.31</td>
<td>( 3.1 \times 10^{-4} )</td>
</tr>
<tr>
<td>TGFB1</td>
<td>61.9 (54.2-69.5)</td>
<td>42.9 (39.0-46.8)</td>
<td>.71</td>
<td>( 3.7 \times 10^{-4} )</td>
</tr>
<tr>
<td>sPECAM1</td>
<td>15.6 (12.6-18.7)</td>
<td>28.7 (23.3-34.1)</td>
<td>1.56</td>
<td>( 3.7 \times 10^{-4} )</td>
</tr>
<tr>
<td>sIL17RB</td>
<td>5.56 (3.84-7.28)</td>
<td>10.9 (9.09-12.7)</td>
<td>1.81</td>
<td>( 1.1 \times 10^{-3} )</td>
</tr>
<tr>
<td>KKL6</td>
<td>28.6 (24.5-32.8)</td>
<td>44.0 (37.9-50.2)</td>
<td>1.29</td>
<td>( 1.1 \times 10^{-3} )</td>
</tr>
<tr>
<td>suPAR</td>
<td>425 (373-476)</td>
<td>295 (251-338)</td>
<td>.69</td>
<td>( 2.0 \times 10^{-3} )</td>
</tr>
<tr>
<td><strong>C</strong></td>
<td></td>
<td><strong>healthy controls</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td>PGF</td>
<td>59.2 (52.5-66.0)</td>
<td>137 (97.2-177)</td>
<td>1.72</td>
<td>( 1.6 \times 10^{-5} )</td>
</tr>
<tr>
<td>GDF15</td>
<td>8.34 (6.14-10.5)</td>
<td>53.3 (22.5-84.1)</td>
<td>3.69</td>
<td>( 9.8 \times 10^{-5} )</td>
</tr>
<tr>
<td>HE4</td>
<td>12.9 (10.4-15.5)</td>
<td>89.8 (0-190)</td>
<td>2.56</td>
<td>( 1.6 \times 10^{-4} )</td>
</tr>
<tr>
<td>sTNFR2</td>
<td>9.52 (7.93-11.1)</td>
<td>17.9 (14.3-21.6)</td>
<td>1.88</td>
<td>( 2.4 \times 10^{-4} )</td>
</tr>
<tr>
<td>CSF1</td>
<td>100 (94.4-106)</td>
<td>195 (115-275)</td>
<td>1.64</td>
<td>( 3.7 \times 10^{-4} )</td>
</tr>
<tr>
<td>Midkine</td>
<td>30.3 (25.8-34.8)</td>
<td>158 (82.9-232)</td>
<td>3.42</td>
<td>( 3.7 \times 10^{-4} )</td>
</tr>
<tr>
<td>sPECAM1</td>
<td>15.6 (12.6-18.7)</td>
<td>69.6 (17.4-122)</td>
<td>2.38</td>
<td>( 7.7 \times 10^{-4} )</td>
</tr>
<tr>
<td>CCL19</td>
<td>463 (317-609)</td>
<td>929 (733-1125)</td>
<td>1.76</td>
<td>( 1.1 \times 10^{-3} )</td>
</tr>
<tr>
<td>sVEGFA</td>
<td>765 (700-830)</td>
<td>1628 (1082-2173)</td>
<td>1.57</td>
<td>( 1.1 \times 10^{-3} )</td>
</tr>
<tr>
<td>IFNγ</td>
<td>1.58 (1.52-1.64)</td>
<td>5.17 (0-11.1)</td>
<td>1.47</td>
<td>( 1.1 \times 10^{-3} )</td>
</tr>
<tr>
<td><strong>D</strong></td>
<td><strong>MM</strong></td>
<td><strong>MM-ASCT</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td>REG4</td>
<td>9.58 (6.78-12.4)</td>
<td>7.01 (4.50-9.51)</td>
<td>.74</td>
<td>( 3.1 \times 10^{-5} )</td>
</tr>
<tr>
<td>sBAFF</td>
<td>18.4 (10.5-26.3)</td>
<td>62.3 (51.8-72.9)</td>
<td>4.38</td>
<td>( 1.5 \times 10^{-4} )</td>
</tr>
</tbody>
</table>

(Continued)
Classification of MGUS, MM, and MM-ASCT

To detect the minimum number and the best combination of serum analytes able to discriminate between MGUS and MM, and MM-ASCT, we applied Multilinear Discriminant Analysis, Naive Bayes classifiers, Random Forests, and extended Support Vector Machine (kSVM). The probability of correct classification to particular patient subgroup (intervals: >90, 90-80, 80-70, 70-60, and 60-50%) was calculated for every combination of two or three analytes from individual patients, and the misclassification error was determined. The best visual separation of studied patient groups was achieved by kSVM and therefore used in further study. The best dual-combination able to discriminate MGUS vs MM was achieved by the combination of sMICA and suPAR, able to separate these groups with a classification error of 0.062 (1 false/16 samples) (Figure 6A). The best triple-combinations for separating MGUS and MM were sMICA-ADM-GDF15 (Figure 6A) as well as the combination of sMICA-ADM-REG4, sMICA-suPAR-REG4, sMICA-suPAR-sHGFR, ADM-suPAR-REG4, TRAP-REG4-sHGFR (data not shown). The triple-combinations increased the probability of correct classification of MGUS and MM; the classification error remained 0.062 (1 false/16). For discrimination of MGUS and MM, MM-ASCT from controls and MM from MM-ASCT, several combinations of only two analytes were sufficient to classify all samples correctly (with no misclassification error). For MGUS vs controls, the combinations were as follows: midkine-sTNFRSF4 (Figure 6B) or midkine-CSF1, sTGFA, TRAP, sTNFR2, sTNFRSF4 or HE4. Also, the combination of sTGFA with CSF1, sTNFR2 or Flt3L discriminated fully between MM-ASCT and healthy controls (data not shown).

<table>
<thead>
<tr>
<th>Analyte</th>
<th>Mean Linear ddCq (95% CI)</th>
<th>FC (Fold Change)</th>
<th>P</th>
<th>Pcorr</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>sPECAM1</td>
<td>69.6 (17.4-122)</td>
<td>17.3 (13.1-21.6)</td>
<td>.41</td>
<td>2.1 × 10^-4</td>
</tr>
<tr>
<td>sIL6R</td>
<td>174 (99.3-249)</td>
<td>81.6 (64.6-98.6)</td>
<td>.55</td>
<td>5.8 × 10^-4</td>
</tr>
<tr>
<td>sPDGFB</td>
<td>544 (299-789)</td>
<td>274 (222-327)</td>
<td>.56</td>
<td>7.6 × 10^-4</td>
</tr>
<tr>
<td>Midkine</td>
<td>158 (82.9-232)</td>
<td>54.5 (42.7-66.3)</td>
<td>.54</td>
<td>1.0 × 10^-3</td>
</tr>
<tr>
<td>sHGF</td>
<td>47.1 (32.5-61.6)</td>
<td>26.9 (20.7-33.1)</td>
<td>.63</td>
<td>1.3 × 10^-3</td>
</tr>
<tr>
<td>TGFB1</td>
<td>85.5 (43.7-127)</td>
<td>41.6 (34.9-48.3)</td>
<td>.72</td>
<td>1.3 × 10^-3</td>
</tr>
<tr>
<td>sAREG</td>
<td>19.0 (5.32-32.8)</td>
<td>7.52 (6.17-8.87)</td>
<td>.67</td>
<td>1.7 × 10^-3</td>
</tr>
<tr>
<td>sMICA</td>
<td>39.4 (26.8-52.1)</td>
<td>22.5 (15.8-29.2)</td>
<td>.59</td>
<td>2.1 × 10^-3</td>
</tr>
<tr>
<td>E healthy controls</td>
<td>MM-ASCT</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>sBAFF</td>
<td>14.1 (12.9-15.3)</td>
<td>62.3 (51.8-72.9)</td>
<td>4.32</td>
<td>8.2 × 10^-6</td>
</tr>
<tr>
<td>CSF1</td>
<td>100 (94.4-106)</td>
<td>152 (131-173)</td>
<td>1.44</td>
<td>2.4 × 10^-4</td>
</tr>
<tr>
<td>sTGFA</td>
<td>24.4 (19.6-29.2)</td>
<td>11.9 (9.03-14.8)</td>
<td>.36</td>
<td>2.4 × 10^-4</td>
</tr>
<tr>
<td>TRAP</td>
<td>32.2 (25.0-39.4)</td>
<td>71.2 (42.5-100.0)</td>
<td>1.82</td>
<td>3.7 × 10^-4</td>
</tr>
<tr>
<td>CXCL10</td>
<td>130 (61.2-199)</td>
<td>667 (323-1011)</td>
<td>4.05</td>
<td>5.4 × 10^-4</td>
</tr>
<tr>
<td>sTNFR2</td>
<td>9.52 (7.93-11.1)</td>
<td>21.2 (17.6-24.9)</td>
<td>2.45</td>
<td>5.4 × 10^-4</td>
</tr>
<tr>
<td>sTNFRSF4</td>
<td>4.12 (3.29-4.94)</td>
<td>10.9 (7.21-14.6)</td>
<td>2.22</td>
<td>5.4 × 10^-4</td>
</tr>
<tr>
<td>Flt3L</td>
<td>254 (225-284)</td>
<td>522 (411-633)</td>
<td>2.07</td>
<td>7.7 × 10^-4</td>
</tr>
<tr>
<td>GDF15</td>
<td>8.34 (6.14-10.5)</td>
<td>22.7 (16.5-28.8)</td>
<td>2.50</td>
<td>7.7 × 10^-4</td>
</tr>
<tr>
<td>HE4</td>
<td>12.9 (10.4-15.5)</td>
<td>43.6 (3.61-83.5)</td>
<td>1.88</td>
<td>7.7 × 10^-4</td>
</tr>
<tr>
<td>THPO</td>
<td>13.1 (11.7-14.6)</td>
<td>28.5 (21.7-35.3)</td>
<td>2.23</td>
<td>7.7 × 10^-4</td>
</tr>
</tbody>
</table>

FC (Fold Change) between group medians of linear ddCq

*Pcorr value corrected for multiple comparisons (Benjamini-Hochberg correction)
DISCUSSION

In this study, we investigated the complexity of serum microenvironment in MGUS, MM and MM after ASCT using highly-sensitive PEA immunoassay. We hypothesized that serum of pre-cancer MGUS and MM differ by the presence of pro-tumorigenic factors. Indeed, we detected elevated levels of adrenomedullin (ADM), TRAP, GDF15, TGFB1, suPAR and other pro-tumorigenic proteins in serum of MM patients compared to MGUS. These proteins were already reported in MM but not investigated simultaneously. Pro-angiogenic factor ADM was identified as the most highly upregulated gene in hypoxia-dependent/independent fashion in MM cells, suggesting to be a major driving force for the angiogenic switch during MM evolution [11]. Highly upregulated TRAP is a marker of osteoclasts driving the bone resorption in MM [12]. The crucial role of TGFB1 in MM is supported by the observation that the inhibition of TGF-β signaling by TGF-β type I receptor kinase inhibitor causes a suppression of MM cell growth and an enhancement of bone formation [13]. Regarding GDF15 in MM, high serum levels were associated with poor prognosis [14, 15] and treatment response [16] and osteolysis [17]. GDF15 enhances the tumor-initiating and self-renewal potential of myeloma cells [18], contributes to drug resistance in both stroma-dependent/independent MM cells [14, 15], and promotes osteoclast differentiation while inhibits osteoblast differentiation [17]. Regarding suPAR, high suPAR expression in MM predicts progression, shorter survival and early extramedullary infiltration [19].

In order to detect the most promising circulating protein(s) distinguishing MGUS and MM, we analyzed co-occurrence of analytes in MGUS/MM classification models and identified sMICA as the most useful classifier. High serum levels of sMICA were already detected in MM as an adverse prognostic factor [20, 21], but not elevated in MGUS [20]. sMICA may originate from MICA-expressing MM cells, fibroblasts or other stromal cells upon stimulation [8, 22]. There is evidence that sMICA impairs the function of the NKG2D + T CD8+ and NK cells, contributing to myeloma cell immune escape [20]. Additionally, patients with MGUS, but not MM, generate high-titer anti-MICA antibodies that antagonize the suppressive effects of sMICA [20]. It has been therefore suggested that alterations in the NKG2D pathway by sMICA and anti-MICA antibodies are critically involved in the suppression of innate and adaptive immunity during the progression from MGUS to MM [20]. Importantly, some drugs may reconstitute the capabilities of sMICA-inhibited cytotoxicity of CD8+ and NK cells [23, 24], thus further highlights the potential of NKG2D + T CD8+ and NK cell-mediated immunotherapeutic interventions in MM [24, 25, 26].

Next, we investigated the minimum number and best combination of serum analytes able to discriminate between MGUS and MM. Advanced data mining methods revealed that the combination of sMICA and suPAR separates these groups with a classification error 0.062. The combination of triplets sMICA-ADM-GDF15 or sMICA-suPAR-REG4 increased the probability of correct classification of MGUS and MM with the same
Figure 2: Serum fingerprints and changes in top-deregulated proteins in MGUS, MM, and MM-ASCT. Fingerprints were presented as FC (fold-change of group medians) of serum levels of deregulated serum proteins between particular groups ($P<.05$); changes in top-deregulated proteins are presented as percentage of changes between group medians of particular groups: A. MGUS vs MM, B. controls vs MGUS, C. controls vs MM. (Continued)
classification error (1/16) significantly. Although larger cohort studies are needed to confirm our results, our study nominated sMICA, ADM, GDF15, suPAR, and REG4 as key MM-associated serum proteins able to discriminate MGUS and MM.

Despite new therapies and ASCT increasing remission rates, nearly all MM patients ultimately succumb to disease relapse and progression. Because tumor microenvironment may contribute to these processes, we investigated for the first time serum pattern from paired samples from MM patients from the time of diagnosis and after ASCT (day 100). Interestingly, the post-transplant sera possessed high levels of soluble B-Cell Activating Factor (sBAFF), a survival factor for myeloma cells [9]. Enhanced serum levels of sBAFF, found produced by MM cells, immune and stromal cells [9, 27], correlated inversely with overall survival in MM and resistance to dexamethasone and lenalidomide [27, 28]. Since the elimination of sBAFF in an MM mouse model resulted in a decrease of tumor burden and protected against lytic bone disease [29], the sBAFF signaling represents a promising therapeutic target in MM [27], especially in the setting of post-transplant sBAFF elevation. After ASCT, the MM-associated proteins sMICA and ADM were downregulated but still elevated compared to healthy controls. On the other hand, serum levels of other pro-tumorigenic factors such as GDF15, CSF1, suPAR, and others did not change after ASCT comparing to paired MM sample at the diagnosis. Similar observation was reported in treated MM patients showing that cytokine pattern in those achieving remission is not restored to physiological levels [30], thus suggesting that once an individual has MM, the microenvironment is permanently altered and primed for a relapse. These results highlight the role of microenvironment for treatment success and may explain why MM remains an incurable disease.

Figure 2: Serum fingerprints and changes in top-deregulated proteins in MGUS, MM, and MM-ASCT. D. MM vs MM-ASCT and E. controls vs MM-ASCT. MGUS is colored yellow, MM red, MM-ASCT blue, and control subjects green.
Figure 3: Changes in serum protein levels between paired samples from MM at the diagnosis and after ASCT (day 100). Pcorr values for differences between two groups of patients after multiple corrections are stated.

Figure 4: Comparison of protein fingerprints in MGUS, MM, MM-ASCT and healthy subjects for selection of top-deregulated proteins.
We were also interested in MGUS associated serum pattern comparing to healthy controls. Our analysis revealed for the first time that MGUS is characteristic by low levels of TGFB1 and high levels of midkine, a heparin-binding growth factor involved in angiogenic and anti-apoptotic functions and tumor expansion in various cancers [31, 32]. Enhanced gene expression of midkine and other angiogenic factors were already reported in MM [33, 34] and also in this study we detected higher serum levels of midkine in MM vs MGUS. Importantly, lower gene expression of midkine and other angiogenic genes was detected in IMiD-responders compared to non-responders [34]. Elevation of midkine, produced by normal and malignant B-cells, tumor and stromal cells [35, 36], was also reported in other B-cell malignancies such as chronic lymphocytic leukemia and lymphomas [35]. Regarding TGFB1, low levels of TGFB1 were shown to control MM cell growth [13]. Our observations highlight the role of TGFB1 and midkine in the progression of MGUS to MM thus deserving further investigation.

We are aware that this study has several limitations. Because this study was focused on determination of serum protein fingerprinting, we did not analyze plasma bone marrow and did not investigate the functional effect of deregulated proteins. This should be performed in future studies.

Figure 5: Network visualization of classification models obtained by pattern-recognition analysis that identified key serum biomarkers distinguishing between MGUS, MM, and MM-ASCT based on co-occurrence of analytes in classification models. A. MGUS vs MM, B. controls vs MGUS, C. controls vs MM, D. MM vs MM-ASCT and E. controls vs MM-ASCT. The size of the vertices (font-size) and connections among vertices show those proteins, which were used in classification rules of the particular patient group in the most accurate classification model.
Figure 6: Extended Support Vector Machine (kSVM) analysis for identification of the minimum number and the best combination of proteins distinguishing MGUS, MM, and MM-ASCT. The dots represent the individual patient data (combinations of two or three analytes). The contour plots show the probabilities (intervals: > 90, 90-80, 80-70, 70-60, and 60-50%) for input data from individual patients to belong to the particular patient group. MGUS is colored yellow, MM red, MM-ASCT blue, and control subjects green. The more saturated color the higher probability of correct classification. A. MGUS vs MM, B. controls vs MGUS, C. controls vs MM, D. MM vs MM-ASCT, E. controls vs MM-ASCT.
In conclusion, we identified serum protein fingerprints associated with MGUS and MM as well changes ongoing in MM after ASCT. The knowledge of serum pattern may contribute to the identification of key myeloma cell survival factors, which may in turn influence treatment response and disease development.

**MATERIALS AND METHODS**

**Study population and materials**

The study cohort includes patients with MGUS (n=16) and MM (n=16); all patients were diagnosed according to the criteria of International Myeloma Working Group [37, 38]. Serum samples were taken at the time of diagnosis from previously untreated patients, aliquoted and stored at -80°C until analysis. In all enrolled MM patients, paired serum sample collected at day 100 after ASCT (MM-ASCT) was also analyzed. Patient characteristics are described in Table 2. Serum samples from healthy control subjects (n=7, mean age 51 yrs; range 45-72 yrs, 4 males/3 females) were taken from members of medical staff; all completed a questionnaire regarding their health status. All patients provided written informed consent about the usage of peripheral blood for the purpose of this study. The study was approved by the ethics committee of University Hospital and Palacky University Olomouc.

**Proximity extension immunoassay**

Serum profiles of ninety-two cancer-related proteins were assessed by the Proseek Multiplex Oncology 1 kit (Olink Bioscience, Uppsala, Sweden) according to the manufacturer’s recommendation. Briefly, serum samples (1 µl) were incubated in the presence of 92 proximity antibody pairs tagged with DNA reporter molecules. Once the pair of antibodies bound to their corresponding antigens, the respective DNA tails formed by proximity extension an amplicon that was quantified by high-throughput real-time PCR (BioMark™ HD System, Fluidigm Corporation). The generated fluorescent signal directly correlates with protein abundance. Olink Wizard (Olink) was used for data normalization: the raw Cq-value (log2 scale) for each data point was normalized by subtracting the Cq-value for the extension control and compared to that of the corresponding background reaction resulting in a ddCq-value [39]. For further analysis, linearized values (2^-ddCq) were used. For panel description see Supplementary Table S1, for sensitivity and specificity parameters of PEA analysis see Assarsson et al. [39].

**Pattern-recognition algorithms**

Binary classification by a stochastic nature-inspired symbolic regression method and evolutionary fuzzy-rules [40] was conducted in order to learn symbolic models for particular patient groups (MGUS, MM, MM-ASCT, healthy subjects) based on their serum protein pattern. For each tested patient group pair, the procedure was repeated more than 500 times to accommodate the stochastic nature of the algorithm and to obtain representative results. The most accurate classification models, separating all patients in the correct patient groups, were utilized for the identification of key molecules and those co-occurring in the classification rules characteristic for the particular patient group.

The proteins from the classification rules were further used to form a network model of patient classification with molecules as vertices and co-occurrences in successful classification models as edges. An algorithm based on analysis of the nearest neighbors between the studied molecules was applied to determine vertex and edge weights in the network model [41]. The size of the vertices (font-size) and connections among vertices show those proteins, which were used in classification rules of the particular patient group in the most accurate classification model.

**Classification methods for separation of patient groups**

To assess the minimum number and best combination of serum analytes distinguishing between the patient groups (MGUS vs MM, MM vs MM-ASCT), we applied several classification methods from the area of information retrieval such as Multilinear Discriminant Analysis, Naive Bayes classifiers, Random Forests, and kSVM to calculate the class probabilities for every input data (combinations of two or three analytes) and to determine the misclassification error [42, 43]. All methods were calculated using R statistical software with package Caret (http://topepo.github.io/caret/index.html).

**Statistical analysis**

Statistical analyses (Mann-Whitney-Wilcoxon and paired Wilcoxon tests) were performed using R statistical software package (http://www.r-project.org/). All data analyses were performed on linearized expression data (2^-ddCq) for each protein. A combination of box plots and swarm plots (a one-dimensional hybrid between scatter plot and strip chart) was employed to visualize the distribution of signals across subjects in particular subgroups. Radar charts were created for each pair of compared subgroups to visually assess the quantitative changes in levels of the most significant molecules, determined for each group pair by the Mann-Whitney-Wilcoxon test, and for paired samples (MM and MM-ASCT) by paired Wilcoxon test. P-value for each protein was adjusted for multiple comparisons using the False Discovery Rate (FDR) by the Benjamini-Hochberg procedure. A P-value < .05 was considered as significant.
# Abbreviations

ADM, Adrenomedullin; ASCT, Autologous stem cell transplantation; CA125, Cancer antigen 125; CA242, CA 242 tumor marker; CCL19, Chemokine (C-C motif) ligand 19; CCL21, Chemokine (C-C motif) ligand 21; CEA, Carcinoembryonic antigen; CR, Complete remission; CSF1, Colony stimulating factor 1; CXCL5, C-X-C motif chemokine 5; CXCL9, Chemokine (C-X-C motif) ligand 9; CXCL10, C-X-C motif chemokine 10; CXCL11, C-X-C motif chemokine 11; CXCL13, Chemokine (C-X-C motif) ligand 13; EPO, Erythropoietin; FABP4, Fatty acid binding protein 4; FDR, False Discovery Rate; Flt3L, FMS-like tyrosine kinase 3 ligand; GDF15, Growth differentiation factor 15; GM-CSF, Granulocyte-macrophage colony-stimulating factor; HE4, Human Epididymis Protein 4; hGH, Human Growth Hormone; IFNγ, Interferon gamma; Ig, Immunoglobulin; IL1RA, Interleukin-1 receptor antagonist; IL2, Interleukin 2; IL4, Interleukin 4; IL6, Interleukin 6; IL12, Interleukin 12; IMiD, Immunomodulatory drug; KLK6, Kallikrein-6; KLK11, Kallikrein-11; kSVM, Extended Support Vector Machine; MIA, Melanoma-derived growth regulatory protein; MGUS, Monoclonal gammopathy of undetermined significance; MM, Multiple myeloma; MYD88, Myeloid differentiation primary response 88; OPG, Osteoprotegerin; PEA, Proximity extension immunoassay; PGF, Placental growth factor; PR, Partial remission; PRL, Prolactin; REG4, Regenerating islet-derived protein 4; SMM, Smoldering multiple myeloma; TGFβ1, Transforming growth factor beta 1; THPO, Thrombopoietin; TRAP, Tartrate-resistant acid phosphatase; sAREG, soluble Amphiregulin; sBAFF, soluble B-cell activating factor; sBTC, soluble Betacellulin; sCAIX, soluble Carbonic anhydrase 9; sCD30L, soluble CD30 ligand; sE selectin, soluble E-selectin; sEGFR, soluble Epidermal growth factor receptor; sEMMPRIN, soluble Extracellular matrix metalloproteinase inducer; sEpCAM, soluble Epithelial cell adhesion molecule; sEpiregulin, soluble Epiregulin; sER, soluble Estrogen receptor; sFasL, soluble Fas ligand; sFOLR1, soluble Folate receptor alpha; sHBEGF, soluble Heparin-binding EGF-like growth factor; sHER3, soluble Receptor tyrosine-protein kinase erbB-3; sHER4, soluble Receptor tyrosine-protein kinase erbB-4; sHGF, soluble Hepatocyte growth factor/scatter factor; sHGF, soluble Hepatocyte growth factor receptor; sIL2RA, soluble Interleukin-2 receptor alpha chain; sIL6R, soluble Interleukin 6 receptor; sIL17RB, soluble Interleukin-17 receptor B; sMICA, soluble MHC class I polypeptide-related sequence A; sPDGFB, soluble Platelet-derived growth factor subunit B; sPECAM1, soluble Platelet endothelial cell adhesion molecule; sSCF, soluble Stem cell factor; sTGF, soluble Tissue Factor; sTGFα, soluble Transforming growth factor alpha; sTIE2, soluble Receptor tyrosine kinase Tie2; sTNFR1, soluble Tumor necrosis factor receptor 1; sTNFR2, soluble Tumor necrosis factor receptor 2; sTNFRSF4, soluble Tumor necrosis factor receptor superfamily member 4; suPAR, soluble Urokinase plasminogen activator receptor; sVEGFA, soluble Vascular endothelial growth factor A; VGPR, Very good partial remission

<table>
<thead>
<tr>
<th>Parameter</th>
<th>MGUS (n=16)</th>
<th>MM (n=16)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Age, years, median (min-max)</td>
<td>59 (46-83)</td>
<td>57 (39-64)</td>
</tr>
<tr>
<td>Gender, n, male/female</td>
<td>8/8</td>
<td>5/11</td>
</tr>
<tr>
<td>Paraprotein type, n (%)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>IgG</td>
<td>11 (69)</td>
<td>7 (44)</td>
</tr>
<tr>
<td>IgA</td>
<td>4 (25)</td>
<td>4 (25)</td>
</tr>
<tr>
<td>IgD</td>
<td>1 (6)</td>
<td>0 (0)</td>
</tr>
<tr>
<td>Light chain only</td>
<td>0 (0)</td>
<td>5 (31)</td>
</tr>
<tr>
<td>Bone lesions, n, yes/no</td>
<td>0/16</td>
<td>12/4</td>
</tr>
<tr>
<td>ASCT Induction regime, n (IMiD-based/bortezomib-based)*</td>
<td>NA</td>
<td>16 (8/8)</td>
</tr>
<tr>
<td>Time difference from diagnosis to day 100 after ASCT, days, mean (min-max)</td>
<td>NA</td>
<td>314 (231-567)</td>
</tr>
<tr>
<td>Response on day 100 after ASCT, CR/VGPR/PR, n (%)</td>
<td>NA</td>
<td>11/4/1 (69/25/6)</td>
</tr>
</tbody>
</table>

*1 patient received tandem ASCT

NA not applicable, CR complete remission, VGPR very good partial remission, PR partial remission
CONFLICT OF INTERESTS

The authors declare no conflicts of interest.

GRANT SUPPORT

This work was supported by grant of Ministry of Health of Czech Republic (MZ CR VES16-32339/A), and in part by the Student Grant System, VŠB – Technical University of Ostrava (SP2016/68).

Authorship

P.S., T.P., and R.F. performed experiments; P.G., M.K., and P.K. analyzed the results and made the figures; T.P., V.S., J.M. collected patient material and provided clinical data; E.K., and T.Pa. designed the research; P.S., T.P., and E.K. wrote the manuscript; and T.Pa. suggested improvements to the manuscript.

REFERENCES

20. Jinushi M, Vanneman M, Munshi NC, Tai YT, Prabhala RH, Ritz J, Neuberg D, Anderson KC, Carrasco DR, Dranoff G. MHC class I chain-related protein A antibodies and


Utilization of the Discrete Differential Evolution for Optimization in Multidimensional Point Clouds

Vojtěch Uher and Petr Gajdoš and Michal Radecký and Václav Snášel

Abstract— The Differential Evolution (DE) is a widely used bio-inspired optimization algorithm developed by R. Storn and K. Price. It is popular for its simplicity and robustness. This algorithm was primarily designed for real-valued problems and continuous functions, but several modified versions optimizing both integer and discrete valued problems have been developed. The discrete-coded DE has been mostly used for combinatorial problems in a set of enumerative variants. However, the DE has a great potential in the spatial data analysis and pattern recognition. This paper formulates the problem as a search of a combination of distinct vertices which meet the specified conditions. It proposes a novel approach called the Multidimensional Discrete Differential Evolution (MDDE) applying the principle of the discrete-coded DE in discrete point clouds (PCs). The paper examines the local searching abilities of the MDDE and its convergence to the global optimum in the PCs. The multidimensional discrete vertices cannot be simply ordered to get a convenient course of the discrete data, which is crucial for a good convergence of a population. A novel mutation operator utilizing a linear ordering of spatial data based on the space filling curves is introduced. The algorithm is tested on several spatial datasets and optimization problems. The experiments show that the MDDE is an efficient and fast method for discrete optimizations in the multidimensional point clouds.

I. INTRODUCTION

The Differential Evolution (DE) has been successfully applied to many continuous, combinatorial and design optimization problems. The measuring devices, cameras, laser devices or sensors produce discrete multidimensional vertices [1][2][3]. The big spatial data is analysed in research areas like robotics, pattern recognition and/or computer vision. In most of these areas, good results have been achieved with the DE (see e.g. [1][4][5][6][7]). This paper proposes a novel DE based algorithm solving the combinatorial tasks with discrete vertices. The abilities of the discrete Differential Evolution to search the optimal combinational solutions in the multidimensional discrete point clouds (PCs) are discussed. Our modified method called the Multidimensional Discrete Differential Evolution (MDDE) uses a vertex hashing function to strengthen the local properties of an n-dimensional discrete dataset.

The Differential Evolution has been introduced by Storn and Price [8]. It is an evolutionary method, which has become popular for its simplicity, robustness, and good convergence properties [9]. It is based on the population of individuals which represent the temporary solutions that are iteratively refined during the generations. Each individual consists of several variables. The quality of individuals is evaluated by an objective function. After the successful DE application for real-valued problems on a continuous space, some combinatorial or design optimization applications on integer or discrete valued problems were presented, such as load dispatch problem [10], unit commitment problem [11], 0-1 knapsack problem [12], generalized traveling salesman problem [13], different NP-hard scheduling problems [14][15][16][17][18], form-finding of tensegrity structures [19], assembly line balancing problem [20] or robots path planning problem [21][22]. Survey of discrete-valued problems and applications of evolutionary algorithms was published in the papers [23][24][25].

There are several basic categories of variables according to the paper [26]. The discrete integer variables bounded within a range are primarily discussed in this paper. We will call this category the discrete-valued variables. The value of such a variable is an integer pointer addressing an enumerative sample from the set of discrete elements. The elements should be arranged to get a better convergence of population [9], otherwise the DE leads to the random search. The existing discrete methods can be divided into a) indirect and b) direct methods. The indirect methods operate with the standard real-valued variables. The values are progressively recalculated to/from the integer ones by some transformation function (see e.g. [27][28][29][30]). The direct methods operate directly with integer values without any transformation, which eliminates the rounding error. The advantages of the indirect methods are that they utilize the robustness of the real-coded DE and require minimal intervention to the original DE. In the paper by Lampinen and Zelinka [27], a simple truncation of the real-valued parameters is proposed. But this simple approach worsens the diversity of the population and the robustness of the algorithm [31]. Other methods using improved rounding techniques involving some additional conditions, constraints and thresholds were published by Angira and Babu [32], Liao [26] or Schmidt [33]. Tatsgetren et al. introduced several approaches using a discrete DE algorithm for a flowshop scheduling problem [34][14]. A novel indirect method called the discrete Differential Evolution (DDE) [29] was proposed by Onwubolu and Davendra. In this case, the whole evolution
is managed with the integer values that are transformed into the real ones only for the mutation phase of the DE. This approach uses the Forward Backward transformation and it has better convergence properties than the simple real/integer rounding techniques [29]. Datta and Figueiredo described a new mutation operator for discrete-valued variables [35]. Their approach called the ridDE is a direct method based on a bit mutation of integer values to avoid the real/integer transformation.

This paper primarily aims at the problems addressing the optimizations in the sparse discrete data represented by distributed vertices in a vector space. The Differential Evolution is often used for the pattern recognition [36][7], clustering [37], classification or feature extraction [38]. All these disciplines find a utilization in the bio-inspired systems and robot automation [4][22][5] or computer vision [36][39]. The article [38] summarizes different applications of evolutionary algorithms in the pattern recognition and machine learning including the Differential Evolution. The DE has been utilized for a human body pose estimation from the point clouds [6][40][36], circles detection [7], ellipses detection [41], a recognition of leukocytes in images or a 3D face model reconstruction utilizing multi-view 2D images [42]. Most of the referenced algorithms optimize analytically a temporary pattern shape, deformable or active shape models. The intersection rate between the proposed model and vertices represents the quality of a solution. However, this means a complete passage of whole dataset every time when a solution is evaluated by the objective function. Our further vision is to apply our novel approach to the direct pattern or feature recognition, where an optimized set of discrete vertices represents the required pattern or its estimate.

To do so, some modifications have to be done in the discrete-coded DE. This paper conducts the basic model of the MDDE. The multidimensional vertices are numbered by their indices in the memory. The discrete-valued variables of individuals store the integer indices addressing the vertices in the memory. Thus, the stochastic optimization iteratively refines the vertex indices to find the required combination of vertices. The local searching abilities of the MDDE in the static point clouds are examined. The DE can efficiently handle a non-linear and a non-differentiable objective function. Thus, it is expected that it should be applicable to the global optimization problems in the sparse point clouds as well. The main problem is that the discrete vertices are unordered and the optimization is very slow and unstable [9]. The 1D enumerative datasets can be ordered by their values. But in the multidimensional space it is necessary to define some hashing function for the n-dimensional vertices. The three space filling curves (SFCs) are tested for the vertex hashing to obtain partly sequenced spatial data (Section II-B).

First, the used real-coded Differential Evolution and the selected SFCs are introduced in Section II. Section III describes the whole method, its input parameters and the utilization of the SFCs. It also solves the problem of duplicate indices generated during the evolution. Section IV tests the proposed method on several optimization problems and datasets. It proves that our novel MDDE efficiently works in the spatial discrete data and the more sophisticated SFCs considerably improve the convergence of a population.

II. RELATED WORK

First, the reference model of the Differential Evolution is reminded (Section II-A). Next, several types of the space filling curves (SFCs) are mentioned in the Section II-B.

A. Real-coded Differential Evolution

The first differential evolution algorithm was presented by Price and Storn in 1995 [43] and then improved in 1997 [8]. It is a simple evolution strategy for a global optimization problem [43][8]. The paper [44] defines several variants of the DE, but the DE/best/b/bin variant is explained here, because it provides better results for most of the tested optimization problems. The basic algorithm is briefly described as follows:

A population consists of \( P \) individuals representing the potential solutions of the selected optimization problem. The objective function \( f(X) \) evaluating the quality (objective value) of an individual is defined as \( f(X) : \mathbb{R}^k \rightarrow \mathbb{R} \). An individual consists of the \( k \) real-valued variables that are represented by a vector \( X = (x_1, \ldots, x_k) \). The problem dependent constraints defining the search-space limiting the values of the variables can be established as well [45][26][46]. Mostly, the minimal value \( \min(f(X)) \) is searched. The process of the evolution is done by generating a new population of individuals with improved objective values. The normalized objective value is usually called the fitness value. The number of generations is limited and labeled as \( g \). The individual with the minimum objective value found during \( g \) generations is returned as the result of the optimization. The appropriate setup of the DE input parameters is discussed in [8][44]. The process of the DE/best/b/bin algorithm can be described as follows:

1) At the beginning of the DE, the random population respecting defined constraints is generated.
2) A \( x_{G}^{i,j} \) is a variable value of an individual from an actual population, where \( i \in [1, P], \ j \in [1, k] \) and \( G \) is a generation counter \( G \in [1, g] \).
3) For \( i = 1, \ldots, P \) of generation \( G \):
   a) Different individuals \( X_{A}^{G} \) and \( X_{B}^{G} \) are selected in the population randomly, where \( i \neq A \neq B \). The third one is the \( X_{G}^{best} \), which represents the best-known solution so far.
   b) A mutant vector is computed by the mutation operator: \( v_{i,j}^{G} = \beta_{G}^{i,j} + F \cdot (x_{A,j}^{G} - x_{B,j}^{G}) \), where \( F \in [0, 2] \) is the mutational factor and \( j = 1, \ldots, k \).
   c) A new individual is computed from the mutant vector by the crossover operator: \( u_{i,j}^{G} = x_{G}^{best,j} \) if \( r_{i,j}^{G} \leq C \) or \( j = D_{i}^{G} \), otherwise \( u_{i,j}^{G} = x_{i,j}^{G} \), where \( j = 1, \ldots, k, C \) is the crossover constant \( (C \in [0, 1]) \), \( r_{i,j}^{G} \) is a random number \( (r_{i,j}^{G} \in [0, 1]) \) and \( D_{i}^{G} \) is a randomly chosen index of individual variable.
4) The step 3 is repeated \( g \) times.

### B. Space filling curves

The algorithm proposed in this paper uses space filling curves (SFCs) to represent the multidimensional discrete data. Three variants of the SFCs were selected: linear indexing (C-curve), Z-order, and Hilbert curve (see e.g. [47][48] and the Figure 1). Generally, SFCs connect the points that are close to each other in the space and thus transform a general n-dimensional problem into one dimensional (1D). Any SFC is usually based on a bounded space division. The bounding box of the dataset is computed. For each vertex a code representing its location in the subspace hierarchy is computed, and the vertices are sorted according to these codes. Thus, the ordered linear array grouping the discrete vertices with a similar space character is created. All the three mentioned SFCs are based on the Octree structure, so that they are universally applicable for the n-dimensional space. The construction of the SFCs is described in [49][50]. The SFCs are very straightforward and efficient methods for sparse space clustering [51]. The C-curve is the basic approach for the linearization of the n-dimensional data. It can be simply constructed, but the local properties are very basic in comparison with the other two SFCs. The Z-order curve is a very popular curve with good local properties and fast construction times. The Hilbert curve fills the space conveniently without any unnecessary crossings or space leaps (see Figure 1), and thus it is considered to be one of the best Octree based SFCs (see [49][51]).

![Fig. 1. The basic types of the space-filling curves (A: C-curve, B: Z-order, C: Hilbert)](image)

### III. DISCRETE DIFFERENTIAL EVOLUTION IN N-DIMENSIONAL SPACE

This section describes a novel approach based on the DE for the discrete multidimensional data analysis. The method is explained on the DE/best/b/bin variant (described in Section II-A), because it seems to be efficient for a distance function minimization, but any other variant can be used [44]. The two discrete-coded methods were tested with spatial data: DDE by Onwubolu et al. [29] and ridDE by Datta [35]. However, the ridDE cannot be parametrized conveniently, thus the DDE was selected as the reference model, as it is introduced in Section III-A. The problem of discrete vertex optimization is described in Section III-B. The multidimensional discrete Differential Evolution (MDDE) utilized for the \( k \) distinct solutions search in spatial data is explained in detail in Section III-C.

#### A. Utilized discrete model of the DE

The DDE by Onwubolu et al. [29] was selected as the reference discrete model, because it works with individuals that consist of the discrete-valued variables. The internal crossover and mutation operators invariably change any applied value to a real number. This leads to in-feasible solutions. Therefore, it is necessary to progressively convert the values from integers to real ones and then back to the integers. The DDE uses the so-called Forward Backward Transformation of values. The Forward (integer/real) transformation is computed only for the mutation and crossover phases of the DE, so that the operators are applied to the real values. The variable values of the new individual are then transformed back to the integers by the Backward (real/integer) transformation and the evolution continues with the integer values. This model is very convenient for combinatorial problems, where the real values make no sense, and for the detection and elimination of the found duplicate values of an individual. The individual is represented by a vector \( X = (x_1, \ldots, x_k) \). The Forward transformation is defined as

\[
(x_j)' = -1 + x_j \times \frac{500}{999},
\]

The Backward transformation is defined as

\[
x_j = \frac{(\text{INT}) \left( (x_j)' + 1 \right) \times 999}{500},
\]

where the \( x_j \) is an integer value, the \( (x_j)' \) is the corresponding real value for \( j = 1, \ldots, k \). The constants were established after an extensive experimentation [29].

The transformations (1) and (2) are mutually inverse.

#### B. Direct MDDE

The modified multidimensional discrete Differential Evolution (MDDE) is very similar to the DE from the Section II-A. The most important differences are in the mutation and the evaluation parts. The MDDE optimizes a set of indices addressing the static vertices of the dataset. The vertices are stored in a linear array in the memory. An individual consists of \( k \) discrete-valued variables. The final solution is defined as a combination of indices addressing the vertices meeting the required conditions. The conditions depend on a specific optimization problem. The objective function can be formulated as a distance function defining some vertex distribution representing e.g. the outline of a required shape.

The main problem is that the real discrete datasets are non-uniformly distributed in the space. Thus, the indices addressing the vertices in the array represent no information about the spatial character of the vertices. Application of the DDE model to the set of unordered vertices leads to
the random search. The dataset has to be ordered to get a better convergence of the population. However, this is not that straightforward in the n-dimensional space, thus a smart vertex hashing has to be applied. The three space filling curves are tested in this paper (Section II-B). A SFC makes the n-dimensional discrete data partly sequenced, so that the close indices address the spatially close vertices. The specific vertex order affects the diversity of the population and the robustness of the algorithm (see the Section IV). The order of vertices is primarily important for the mutational phase of the evolution.

As the MDDE is a randomized algorithm, it is possible that a new generated individual contains some duplicate indices. Generally, a resulting solution consisting of distinct vertex indices is expected to obtain the set of vertices representing the searched pattern or feature. The duplicities have to be eliminated to obtain the duplicity free individuals at the end of every generation. The basic algorithm works as it follows:

1) The input parameters and data are set.
2) The SFC representation of a point cloud is computed.
3) The initial population of \( P \) individuals is generated. Each individual consists of \( k \) discrete-valued variables, which are randomly initialized, so that there are no duplicities.
4) All individuals are evaluated by the objective function.
5) For each individual of a population:
   a) Three different individuals are randomly selected from the current population.
   b) The best known individual and the two of the randomly selected individuals are combined:
      i) The Forward transformation (1) of the variable values is computed for all parent individuals.
      ii) The mutation operator and the crossover operator are applied to the corresponding variables.
      iii) The variable values of the new individual are transformed to the integers by the Backward transformation (2) and validated afterwards.
   c) The duplicate variable values of the new individual have to be resolved. The duplicities are replaced by distinct values from the third randomly selected individual.
   d) An individual is evaluated by the objective function according to the total objective value (e.g. sum of separate distances). The new individual is compared with the corresponding one from the current population and the better one is selected for the new population.
   e) The best known solution is replaced with the new individual and replaced eventually.
6) The point 5) is repeated in each of \( g \) generations.
7) Finally, the resulting vertices are read according to the found integer indices stored in the discrete-valued variables of the best found individual.

C. The \( k \) distinct solutions search

This section describes the parts of the MDDE algorithm in more detail. The utilization of the SFCs, the mutation and the duplicity elimination are explained here.

1) Initialization: The input parameters of the MDDE are almost the same as those mentioned in the Section II-A:

\[
l \quad \text{number of vertices in the dataset} \\
f(X) \quad \text{total objective function, where } X = (x_1, \ldots, x_k) \\
f_k(x_i) \quad \text{separate objective function} \\
P \quad \text{number of individuals of a population} \\
k \quad \text{number of individual variables} \\
n \quad \text{dimension of the discrete vertices and the separate objective function} \\
g \quad \text{maximum number of generations} \\
F \quad \text{constant mutational factor, } F \in [0, 2] \\
C \quad \text{crossover constant, } C \in [0, 1]
\]

2) Individual representation: Each individual of the population consists of \( k \) discrete-valued variables storing the vertex indices. One array containing the \( 2^k \) individuals is allocated. The alternation of populations is done by the double buffering of \( P \) individuals and the populations are switched simply by the exchange of pointers addressing the \( 0th \) and the \( P−1 \)th individual. The individual variables are aligned in the memory as well, thus \( 2 \cdot P \cdot k \) values (32-bit) are stored in a row.

3) Initial population: The first duplicity free population has to be generated. The range of the vertex indices \((0, l)\) is divided into \( k \cdot P \) blocks. One random index is selected from each block, thus the \( k \) different initial values are generated randomly for each of \( P \) individuals. A random permutation of the values is computed afterwards. Therefore, the variable values of all individuals are completely distinguished.

4) Evaluation: The evaluation of the objective function with an individual is done similarly as it is in the case of casual 1D discrete data. The whole MDDE works with vertex indices assigned by the SFC. The separate objective function is called with a vertex addressed by the integer index. If an individual consists of more variables, a multidimensional objective function will be utilized. Generally, the variables are evaluated by a separate objective function and the sum of \( k \) particular objective values is used to compute the total objective value of an individual. However, this can be done only if the particular objective value converges by itself (e.g. Euclidean distance). Otherwise, a sophisticated objective function must be used.

5) Mutation operator: The MDDE operates with vertex indices addressing the ordered vertices on the SFC (Figure 2). The mutation operator computes a mutant vector as a linear combination of three different individuals (Section II-A): two from the current population and the best known one (see Figure 2). According to the DDE model, the mutation operator already calculates with the transformed real values. The computation of the mutant vector is done for each individual variable:

\[
v_{i,j}^G = x_{\text{best},j}^G + F \cdot (x_{A,j}^G - x_{B,j}^G),
\]
where \( i = 1, \ldots, P, \; j = 1, \ldots, k \) and \( G \) is a generation counter. Obviously, the mutation operator can be simply reformulated to e.g. DE/rand/1/bin and other variants [44] if it is needed. Figure 2 shows that the order of vertices is crucial for the convergence of population. The SFC better secures that the mutant index \( v^G_{a,i} \), computed from the parent indices (3) addresses the vertex that is placed nearby the vertices addressed by the parent indices \( x^G_{\text{best},j}, x^G_{A,j} \) and \( x^G_{B,j} \). In the case of unordered point clouds, the mutation would practically lead to a random selection of a mutant vector without any spatial logic (see Figure 2).

6) Crossover operator: The traditional crossover operator described in the Section II-A is applied. A proposed (mutant) solution is accepted with the probability \( C \). If \( k > 1 \), the operator will be applied separately for each variable. The variable values of the new individual are transformed to the integers by (2). Additional constraints and the condition that the values (indices) belong to \((0,1)\) have to be validated afterwards. If a variable value is placed out of the interval, a random value in the interval \((0,1)\) will be selected.

7) Separate selection operator: If \( k > 1 \), and it is possible to assess the quality of the variable values separately, the selection can be made on the level of separate variables. This pretends the average results generated by the simple optimization of the sum of values and improves the convergence of the population. E.g. the vertex distance from the proposed pattern can be used as a separate metric.

8) Elimination of duplicities: The various combinations of distinct variable values (vertex indices) may lead to the same resulting value due to the convergence to the global optimum. The duplicities have to be found and replaced to get a better diversity of a discrete solution. A point cloud is a finite set of vertices, thus a subset of \( k \) sufficient vertices can fulfill a condition resulting in some pattern or feature recognition. Therefore, the duplicity free solutions are required. All individuals are checked for duplicities before the final individual selection to preserve this demand for the new population.

For each newly generated individual \( U^G_a \) another one \( X^G_b \) is randomly selected from the current population (the new one is not finished yet), where \( G \) is a generation number and \( a \neq b \). A new individual \( U^G_a \) is checked for duplicities at first and the number of recurrences \( r \) is obtained, where \( r \in (0,k) \). The mentioned facts mean that the \( U^G_a \) and the \( X^G_b \) can have maximally \( k-r \) identical indices after the elimination of \( r \) recurrences from the \( U^G_a \). Thus, having the certainty of the duplicity free \( X^G_b \), the remaining \( r \) indices can replace the recurrences of the \( U^G_a \).

The implementation of this algorithm is based on a convenient flagging of the indices followed by their sorting (Figure 3). The index arrays of both individuals are copied to the temporary array one by one. The \( U^G_a \) is stored at first followed by \( X^G_b \). Another array holds the corresponding flags of \( 2k \) indices. The flagging is done by the sequential comparison of unmarked indices. The indices of the \( U^G_a \) are flagged at first and the \( r \) recurrences are found. The unique indices are flagged with 1 and the duplicities with 3. Next, the \( r \) distinct indices have to be found in the \( X^G_b \), so that the indices of the \( X^G_b \) are compared with the preceding ones. The unique indices are flagged with 2 and the search will be terminated when \( r \) indices are found. The remaining indices are flagged by 3. The \( 2k \) indices are sorted by Quick sort algorithm according to the flags, thus the first \( k \) indices represent the new duplicity free individual. The flagging can be also used for penalization of undesired solutions so that the penalized indices are sorted out.

9) Finally: The new proposed individual is compared with the best known one. The total objective value is used to assess the best ascertained individual. The whole computation is terminated after \( g \) generations, or when a terminating condition is met. The ascertained individual with the best total objective value is returned.

IV. EXPERIMENTS AND DISCUSSION

In this section, the proposed MDDE method is tested. The main aim of the experiments is to test the local behaviour of the MDDE on the three space filling curves (SFCs) and its convergence to the global optimum in the discrete point clouds (PCs). The C-curve was selected as a naive vertex hashing algorithm for comparison to show that the MDDE running on more complex SFCs with better local properties converges faster to the searched extreme. It seems there is no
comparable method addressing the combinational problems on the level of discrete multidimensional vertices. The SFCs are constructed by a hierarchical vertex hashing followed by sorting of the vertices according to the hashes/codes (see Section II-B). A code represents an octant that contains the hashed vertex. The order of the octant written to the code distinguishes the different variants of the SFCs. The codes are usually represented by a bit sequence of octant coordinates. The SFCs of all the tests and datasets were constructed for the maximum hierarchical level allowed by the 64-bit integer. The bit length of the hash is the main limitation of our method, because the greater is the dimension \( n \) of the discrete vertices the lower is the maximum level of clustering and the ability of the SFCs to distinguish location of two close vertices. That is why the experiments are focused on 2D and 3D problems and datasets. But the MDDE is generally applicable for \( n \)-dimensional spaces if the longer hashes are used.

This paper primarily aims at the problems addressing the optimizations in the sparse discrete data represented by distributed vertices in a vector space. It is assumed that the observed property or the pattern are locally bound to the spatial data. Several discrete methods were tested, but the DDE by Onwubolu and Davendra [29] has been chosen. In comparison with the riDE [35], the DDE provides the option of the \( F \) parameter setting that allows one to define the sampling step of the evolution. All the tests were performed with the DE/best/l/bin variant, as this seems to be the best one after extensive experimentation.

\section{A. The definition of the tested problems}

The algorithm was tested on several common optimization problems:

1) point-to-point and point-to-line distance minimization problem
2) discrete optimization of Schwefel and Rastrigin functions
3) maximum distance search in 3D datasets

These problems have been selected, because they are applicable for all kinds of point clouds and space dimensions and they mostly represent the basic tasks in the area of the spacial data analysis. They can be precisely solved analytically by the brute force vertex comparison as well, thus it is possible to compare the results of the analytical and the evolutionary approaches. The problems are described in the following subsections.

1) Point-to-point and point-to-line distance minimization: The objective function of the point-to-point problem is defined as the Euclidean distance between a randomly chosen vertex \( p \) from the dataset and the vertices proposed by an evolution. The objective function of the point-to-line problem is defined as the Euclidean distance between the line constructed by two different vertices randomly chosen from the dataset and the vertices proposed by an evolution [52]. The distance is the basic metric that is generally minimized to recognize some shape or pattern. The evolution converges locally to the global extreme in this case, thus it is a good example that can be tested with the MDDE. It is obvious that the randomly selected vertices have to be consistent during the whole evolution process. The distances of the \( k \) vertices of each individual are optimized separately and the total fitness (objective) value of an individual is computed as a sum of \( k \) distances. In both cases, the zero distance solutions are heavily penalized in order to provide the comparison rating between the analytical and the evolutionary approach.

2) Discrete optimizations of test functions: The evolutionary algorithms are usually checked on several continuous test functions [53]. The well-known Schwefel and Rastrigin functions have been selected for the tests of the MDDE, because they are both very complex functions with many local minima and they are applicable for any dimensions (see [53]). These continuous functions represent the corresponding objective functions evaluating the quality of the ascertained vertices. The discrete vertices of the dimension \( n \) are randomly generated in the typical input domains defined e.g. in [53]. Thus the optimization is based on the search of the \( k \) distinct vertices with the minimal objective value.

Two different distributions of random samples were tested to better distinguish the properties of the space filling curves (see Figure 4). The Gaussian distribution consists of \( 10^5 \) vertices sampled randomly according to the standard normal distribution recalculated to the intervals of the input domain. Similarly, the Gaussian islands are the ten randomly chosen vertex groups distributed according to the standard normal distribution (Figure 4B) containing together \( 10^4 \) vertices. The distributions are the same for all measurements.

\begin{figure}[h]
\centering
\includegraphics[width=\textwidth]{fig4}
\caption{Vertex distributions: A) Gaussian, B) Gaussian islands}
\end{figure}

3) Maximum distance search: The problem is defined as a search of the two most distant vertices of the dataset. This can be used e.g. as an approximative solution of the minimum sphere problem, which is defined as a search of the minimum sphere containing all the vertices of the dataset [54]. The minimum sphere problem is more complex, because the maximum Euclidean distance used as a perimeter of the sphere does not guarantee that all the vertices are contained inside the sphere. However, in many cases the maximum distance can be used as a good estimate of the minimum sphere problem solution, which can be further improved. We reformulated it to a minimization problem, so that the difference

\[ \Delta Dist = \text{diagonal} - \text{maxDist} \] (4)
TABLE I
THE DE PARAMETERS FOR POINT-TO-POINT AND POINT-TO-LINE MINIMIZATION PROBLEMS AND DIFFERENT DATASETS.

<table>
<thead>
<tr>
<th>Dataset</th>
<th>Vertex num. (l)</th>
<th>Fitness rate</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>PP</td>
</tr>
<tr>
<td>Uniform rand.</td>
<td>$10^6$</td>
<td>2.0 (PL)</td>
</tr>
<tr>
<td>Gauss rand.</td>
<td>$10^6$</td>
<td>2.0 (PL)</td>
</tr>
<tr>
<td>Gauss islands rand.</td>
<td>$10^4$</td>
<td>1.5 (PL)</td>
</tr>
<tr>
<td>Stanford Bunny$^1$</td>
<td>35947</td>
<td>1.8 (PL)</td>
</tr>
<tr>
<td>Stanford Dragon$^1$</td>
<td>437645</td>
<td>2.2 (PL)</td>
</tr>
<tr>
<td>Stanford Buddha$^1$</td>
<td>543652</td>
<td>2.2 (PL)</td>
</tr>
</tbody>
</table>

Common parameters:
- $k = 16$ (PP$^*$) or $k = 50$ (PL$^*$),
- $F = 0.01$ (PP$^*$) or $F = 0.005$ (PL$^*$),
- $P = 30$ (PP$^*$) or $P = 80$ (PL$^*$)

* Point-to-point (PP), Point-to-line (PL)
$^1$ Models from The Stanford 3D Scanning Repository [55]

is minimized, where diagonal is the diagonal length of the bounding box and the maxDist is the maximum distance between two vertices found in the dataset. The bounding box diagonal represents the possible maximum distance of two vertices, thus the $\Delta Dist$ is always positive.

This problem is different from the others, which locally converge to the extremes. But the maximum distance can be found by the local search of two distant areas, which leads to finding of greater distances. Therefore, the MDDE algorithm converges to the global extreme as well.

B. Achieved results

This section discusses the achieved results of the MDDE tested on the defined problems. The three artificial and the three real standard datasets were chosen for the tests, as they are mentioned in Table I. The random Gaussian datasets were generated according to standard normal distribution. The Gaussian islands were explained in Section IV-A.2. For all optimization problems and datasets the best solutions are computed analytically in advance.

1) Sufficient solution search: First, the point-to-point and point-to-line problems were tested (see Section IV-A.1). The corresponding DE parameters for both problems can be seen in Table I and they were established after extensive experimentation. Figure 5 shows the comparison of the SFCs on the six different 3D datasets. These tests measure the number of DE generations needed to obtain a sufficient result, so that all k vertices must have the sufficient distance. The sufficient result $X_{best}^{G}$ has to meet the condition

$$f_s(x_{best}^{G}) < f_{best} \cdot \text{fitnessRate}$$

for $j = 1, \ldots, k$, where k is the number of individual indices, $G$ is a generation counter, $f_s$ is a separate objective function that returns the distance of the $j$-th individual vertex from the reference point/line, the $f_{best}$ is the best analytically computed solution and the fitnessRate is the corresponding accuracy rate according to Table I. Each measurement was performed 50 times for different randomly selected vertices, which define the reference vertex or line. Thus, the graphs represent the convergence metrics examining various areas of the distributed datasets. Figure 5 shows that the MDDE utilizing the Z-order and the Hilbert curve converges faster to the global optimum than in the case of the C-curve. The Z-order generally shows better results than the Hilbert curve especially in sparse and non-uniformly distributed datasets.

2) Convergence tests: The next measurements are focused on the evolution convergence during the generations. The figures 6, 7, 8 and 9 show the MDDE progress measured on different problems, datasets and dimensions. These measurements are visualized by the ribbon plots or curves of medians constructed from 20 preformed measurements. The vertical axis represents the corresponding fitness value expressed by a multiple of the best analytical solution.

Figure 6 shows the comparison of the ribbon plots displaying the median, the first and the third quartile of the measured fitness for point-to-point and point-to-line problems. These tests were performed on the artificial datasets with $10^6$ vertices with the Gaussian distribution according to the parameters in Table I. The Z-order shows its supremacy again, the C-curve has the worst convergence in this measurement. The accuracy is much better in the case of the point-to-point distance problem, because the line crosses the whole point cloud, thus there are many very close vertices. The vertices with the zero distance metric are eliminated in both cases.

The figures 7 and 8 show similar convergence metrics for the Rastrigin (Figure 7) and the Schwefel (Figure 8) test functions. Only the medians are displayed to obtain a better legibility of the plots, Table II summarizes the MDDE parameters for all tests. The tests on both functions were performed on artificial datasets with the Gaussian distribution ($10^5$ vertices) and Gaussian islands ($10^4$ vertices), as it was explained in Section IV-A.2. The results are more comparable in contrast with the distance functions especially in the case of the Gaussian islands. However, the Z-order mostly shows the fastest convergence and the best accuracy in comparison with the other SFCs.

Finally, the Figure 9 represents the convergence metrics of the maximum distance problem reformulated to the minimization problem (see Section IV-A.3). These tests were performed on the three Stanford datasets mentioned in Table I according to the parameters in Table II. The plots show the progress of the fitness rate during the 100 generations. The results are quite comparable again, but the Z-order converges faster than the Hilbert curve and C-curve.

3) Completeness tests: The MDDE returns a vector of vertex indices as a result of the optimization. The discrete optimal solutions can be found analytically in the datasets with the finite number of vertices, so that the intersection of the stochastically found solution and the best solution can be computed. Thus, the completeness is defined by the rate

$$c = \frac{a}{k}$$

where $a$ is the number of correctly found vertex indices of an individual and $k$ is the total number of individual indices.
The completeness was measured after 100 generations of the evolution on the Rastrigin (Table III) and the Schwefel (Table IV) test functions, because they are very complex functions with many local minima. The measurements were performed with the DE parameters summarized in Table II. The tables III and IV represent the completeness comparison for the three SFCs and two vertex distributions. The tables show that the completeness is better in the case of Gaussian islands and 3D space. The same number of vertices distributed in the 2D space leads to the greater density of sampling, thus there are more vertices with good fitness than in the 3D space, where the distances between samples are greater. Therefore, the distinction of two very close solutions is very complicated for such a bio-inspired method. However, the results are still very good especially in the case of Z-order and Hilbert curves.

4) Performance tests: This section briefly introduces the performance of the proposed MDDE algorithm. The evolutionary times of 100 generations including the duplicity elimination are summarized in Table V. Each measurement was performed 50 times on all the mentioned optimization problems according to the DE parameters in Table I and Ta-
Fig. 6. The ribbon plots comparing the evolutions on point-to-point (left) and point-to-line (right) distance minimization problems for different SFCs (red: Z-order, yellow: Hilbert, green: C-curve) and artificial datasets with the Gaussian distribution (2-3 dimensions). The vertical axis (log10 scale) shows the fitness value expressed by a multiple of the best solution precomputed analytically. Each measurement was performed 20 times for the same parameters (see Table I).

Fig. 7. The convergence of the discrete evolution on the Rastrigin function utilizing the different SFCs (red: Z-order, yellow: Hilbert, green: C-curve). The vertical axis (log10 scale) shows the fitness value expressed by a multiple of the best solution precomputed analytically. The measurements were performed 20 times for two different vertex distributions (left: Gaussian, right: Gaussian islands) and the same parameters (Table II).

V. CONCLUSION

A novel modification of the DE called the Multidimensional Discrete Differential Evolution (MDDE) addressing the combinatorial problems in n-dimensional point clouds is presented. Our method aims at the discrete-valued problems, where a combination of multidimensional vertices represents...
the required solution. The convergence of the evolution is improved by a spatial data linearization by the space filling curves (SFCs). The algorithm efficiently eliminates the problem of the duplicate values in an individual. The paper examines the local searching abilities of the MDDE and the convergence to the global extreme in the discrete point clouds. The method is tested on several spatial optimization problems and the three SFCs (Z-order, Hilbert, C-curve). The tests on the convergence and completeness of the discrete solution show that the Z-order curve can be recommended as the best variant from the tested SFCs. The completeness of the best found solutions mostly balances between 60% to 100% depending on the used SFC. The evolution converges fast especially during the first 50 generations. The computation times of 100 generations measured on the test problems are maximally several milliseconds. Our MDDE
TABLE V
COMPUTATION TIMES (S) OF 100 GENERATIONS OF THE DEFINED PROBLEMS

<table>
<thead>
<tr>
<th>Problem</th>
<th>2D</th>
<th></th>
<th>3D</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Point-to-point</td>
<td>0.0075</td>
<td>0.0005</td>
<td>0.007</td>
<td>0.0077</td>
</tr>
<tr>
<td>Point-to-line</td>
<td>0.0053</td>
<td>0.0005</td>
<td>0.0065</td>
<td>0.0064</td>
</tr>
<tr>
<td>Rastrigin (Gaussian)</td>
<td>0.0059</td>
<td>0.0005</td>
<td>0.006</td>
<td>0.0073</td>
</tr>
<tr>
<td>Rastrigin (Gauss. islands)</td>
<td>0.004</td>
<td>0.0004</td>
<td>0.004</td>
<td>0.0044</td>
</tr>
<tr>
<td>Schwefel (Gaussian)</td>
<td>0.0067</td>
<td>0.0004</td>
<td>0.007</td>
<td>0.0083</td>
</tr>
<tr>
<td>Schwefel (Gauss. islands)</td>
<td>0.0044</td>
<td>0.0005</td>
<td>0.004</td>
<td>0.0053</td>
</tr>
<tr>
<td>Maximum distance</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>0.0019</td>
</tr>
</tbody>
</table>

TABLE III
RASTRIGIN FUNCTION: COMPLETENESS OF RESULTS AFTER 100 GENERATIONS

<table>
<thead>
<tr>
<th>Distribution</th>
<th>2D</th>
<th></th>
<th>3D</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Gaussian</td>
<td>0.61</td>
<td>0.25</td>
<td>0.54</td>
<td>0.54</td>
</tr>
<tr>
<td>Gauss. islands</td>
<td>0.94</td>
<td>0.09</td>
<td>1.0</td>
<td>0.90</td>
</tr>
<tr>
<td>C-curve</td>
<td>0.04</td>
<td>0.05</td>
<td>0.00</td>
<td>0.01</td>
</tr>
<tr>
<td>Gauss. islands</td>
<td>0.22</td>
<td>0.12</td>
<td>0.2</td>
<td>0.81</td>
</tr>
</tbody>
</table>

TABLE IV
SCHWEFEL FUNCTION: COMPLETENESS OF RESULTS AFTER 100 GENERATIONS

<table>
<thead>
<tr>
<th>Distribution</th>
<th>2D</th>
<th></th>
<th>3D</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Gaussian</td>
<td>0.04</td>
<td>0.06</td>
<td>0.00</td>
<td>0.16</td>
</tr>
<tr>
<td>Gauss. islands</td>
<td>0.21</td>
<td>0.11</td>
<td>0.2</td>
<td>0.93</td>
</tr>
</tbody>
</table>

The work was also supported by Grant Agency of the Czech Republic, under the grant no. GACR GA15-06700S: Unconventional Control of Complex Systems.

ACKNOWLEDGMENT

This work was supported by SGS project, VSB-Technical University of Ostrava, under the grant no. SP2016/97. This is an efficient and fast method for discrete optimizations in the multidimensional point clouds. The main disadvantage of the MDDE is the limited precision of the SFCs, which are limited by the bit length of the vertex hashes. This is considerable especially in higher dimensions.

The MDDE represents a basic discrete model for a pattern recognition and a feature extraction especially in the 2D and 3D discrete datasets. The difficult task is to formulate the real problems for the MDDE, thus this will be the direction of our future work. We have promising results in the area of primitives detection, where the MDDE can accelerate the convergence of evolution.

REFERENCES


Regular Paper

A parallel Fruchterman–Reingold algorithm optimized for fast visualization of large graphs and swarms of data

Petr Gajdoš, Tomáš Ježowicz, Vojtech Uher, Pavel Dohnálek*

Department of Computer Science, FEECS VŠB – Technical University of Ostrava, 17. listopadu 15, 708 33 Ostrava, Czech Republic

A R T I C L E   I N F O
Article history:
Received 29 October 2014
Received in revised form 16 July 2015
Accepted 22 July 2015
Available online 14 August 2015

Keywords:
Graph layouts
Nearest neighbors
Fruchterman–Reingold
Fast graph visualization
GPU
CUDA

A B S T R A C T

Graphs in computer science are widely used in social network analysis, computer networks, transportation networks, and many other areas. In general, they can visualize relationships between objects. However, fast drawing of graphs and other structures containing large numbers of data points with readable layouts is still a challenge. This paper describes a novel variant of the Fruchterman–Reingold graph layout algorithm which is adapted to GPU parallel architecture. A new approach based on space-filling curves and a new way of repulsive forces computation on GPU are described. The paper contains both performance and quality tests of the new algorithm.

1. Introduction

Graphs can bring a new view of the data structure and relationships between its elements. They can visualize some additional information, e.g. clusters, that are hidden by default. However, graphs become more complex and unreadable with a growing amount of data and it is evident that a naive visualization of complex graphs leads to loss of information, e.g. the user can see just a clutter of nodes and edges. Some layout algorithm must be used to provide a readable form of the graph structure. The main disadvantage of using layout algorithms lies in the additional computation time, which can be very long in the case of a large amount of data. Fast graph drawing with a readable layout is still a challenge [1]. For example, it is practically impossible to visualize large social networks [2] like Facebook and Twitter, and illustrate the dynamic of such networks. The same holds true for the visualization of WWW pages connected by links, computer networks [3], or protein similarity [4], or even visualizing large quantities of particles in Particle Swarm Optimization [5], ants during Ant Colony Optimization [6], genes in Genetic Algorithms [7], etc.

Several layout algorithms exist [8–10] that can be used to create an acceptable graph visualization. Usually, more layout algorithms can be applied on the same graph to provide variants for the readers. The layout should ideally help the reader to better understand the information contained in the graph. Several aesthetic techniques or metrics like edge crossing, line bends, symmetry, minimum angle or orthogonality are presented in [11]. Ref. [12] shows which of the aesthetic criteria have the greatest influence for human understanding. Note that for some graphs, even if one layout has no edge crossings, a different layout with more crossings may be considered as a better one (showed in [8]). The more aesthetic criteria are required, the more computation time is usually needed.

The paper is organized as follows. Section 2 presents the related work. In Section 3 the nearest neighbors (NN), space-filling curves (SFCs), graph layout problem and Fruchterman–Reingold (FR) algorithm are described. In Section 4, the proposed algorithm is discussed as well as the obtained speed-up in the final implementation. Finally, the last section summarizes the performance and quality of the results and experiments.

2. Related work

Some of the graph layout algorithms represent a class of the so-called force-directed layouts, e.g. Fruchterman–Reingold [9] or Kamada–Kawai [8]. These algorithms iteratively change the positions of vertices/nodes to reduce a defined energy function, also called the temperature $T$. These layouts are generally considered to be aesthetic. The problem related to the usage of the force-
directed algorithms is that in general they are computationally expensive. The paper [2] describes the speed-up of the Fruchterman–Reingold by computing the most expensive part on GPU.

Our research focused on the method of finding the nearest neighbors for every vertex to achieve more effective computation. Thus the solution of the nearest neighbors problem described for example in [13,14] became the primary goal of our research. One of the fast methods for nearest neighbors searching is based on the space-filling curves. The space-filling curves used in the graph layout can be found in [15], where a new approach dealing with dense graphs was presented. Moreover, searching for new layout algorithm leads to the consequent challenge that consists of comparison techniques. When a new layout algorithm is found, the natural question is how to compare it with others. Ref. [11] shows several quality measurements for graph layouts. Next, in [12], quality measurements were studied from the point of view of human reading.

### 3. Background of the suggested visualization

In this section a selected space-filling curve is briefly described with primary focus on the problem of computation of graph vertices/nodes coordinates. Also the Fruchterman–Reingold algorithm is mentioned.

#### 3.1. Nearest neighbors and space-filling curves

Space-filling curves (SFCs) were used in order to get the nearest neighbors, because they can be computed very fast in parallel way. Several variants exist such as Peano, Z-order or Hilbert space-filling curve [16–18] (Fig. 1).

Note that SFCs are very inaccurate in classifying nearest neighbors. Generally, SFCs connect the points that are close to each other and thus transform the n-dimensional problem into one dimensional (1D). Unfortunately, many points should be considered as nearest neighbors but ultimately they are far from each other on the SFC. On the other hand, some vertices lie very close together on the curve but in reality they are far away. The mentioned disadvantage is balanced with the fact that the SFC can be computed very easily and in parallel. It requires just computing a single index for each point \((\theta(N))\) and then reordering such indices, e.g. with quick-sort or with parallel radix sort using CUDA. An example of Z-order SFC can be seen in Fig. 2.

![Fig. 1. Example of Z-order space-filling curve.](image1)

#### 3.2. Graph layout and Fruchterman–Reingold

Let a graph \(G = (V,E)\) be a set of vertices \(V\) and edges \(E\) that connects those vertices. One of the well known algorithms for the graph layout is the Fruchterman–Reingold, which belongs to the family of force-directed graph layout algorithms. Vertices connected by an edge attract each other. It also defines an ideal distance for each vertex. The vertices should be drawn near each other, but not too close. To lay out a graph, the vertices are replaced by steel ring and each edge with a spring mechanical system [9]. The complete algorithm is shown below:

- \(area \leftarrow W \times L\) (width and length of the frame)
- \(G = (V,E)\) \(\{\text{random initial positions for the vertices}\}\)
- \(k \leftarrow \sqrt{area/|V|}\)
- function \(F_d(z)\): \(\{\text{return} \ x^2/k\}\)
- function \(F_t(z)\): \(\{\text{return} \ k^2/z\}\)
- for \(i \leftarrow 1\) to iterations \(\{\text{calculate repulsive forces}\}\)
  - for all \(v \in V\) \(\{\text{each vertex has two vectors: .pos and .disp}\}\)
    - \(v\).disp \(\leftarrow 0\)
    - for all \(u \in V\) \(\{\text{if} \ u \neq v \text{ then}\}\)
      - \(\Delta \leftarrow v\).pos - u.pos\)
      - \(v\).disp \(\leftarrow v\).disp + \(\Delta/|\Delta|\) \(\times F_d(|\Delta|)\)
    - end if
  - end for
- for \(u \in E\) \(\{\text{calculate attractive forces}\}\)
  - \(\Delta \leftarrow e\).v.pos - e.u.pos\)
  - \(e\).t.disp \(\leftarrow e\).v.disp + \(\Delta/|\Delta|\) \(\times F_t(|\Delta|)\)
  - \(e\).u.disp \(\leftarrow e\).u.disp + \(\Delta/|\Delta|\) \(\times F_t(|\Delta|)\)
  - end for
- \(\text{[limit the max displacement to the temperature T]}\)
- \(\text{[and then prevent from being displaced outside frame]}\)
- for all \(v \in V\) \(\{\text{do}\}\)
  - end for

![Fig. 2. 8 vertices (A, B, C, D, E, F, G, H) randomly placed and ordered by Z-order space-filling curve.](image2)
\[ v.\text{pos} \leftarrow v.\text{pos} + (v.\text{disp} / |v.\text{disp}|) \min(v.\text{disp}, T) \]
\[ v.\text{pos.x} \leftarrow \min(W/2, \max(-W/2, v.\text{pos.x})) \]
\[ u.\text{pos.y} \leftarrow \min(L/2, \max(-L/2, v.\text{pos.y})) \]

\textbf{end for}

\textbf{end for}

The algorithm shown above basically consists of three independent parts. The computation of repulsive forces is the bottleneck of the whole algorithm due to its \(\Theta(N^3)\) time complexity; more details about this is in Section 4.2. The remaining parts work with edges and with all vertices again:

1. Calculate repulsive forces (\(\Theta(N^2)\)).
2. Calculate attractive forces (\(\Theta(E)\)).
3. Iterate all vertices (\(\Theta(N)\)).

3.3. GPU computing

CUDA environment [19,20] allows a heterogeneous programming approach. Simply said, one part of the code is processed by GPU and another part by CPU. The graphic card has basically five types of memory. The most important is global memory (DRAM), which serves as a communication point between GPU and CPU. Since global memory is slow, a shared memory (shared by threads in one block) is often employed to achieve better performance. Registers are the fastest memory. A constant and texture memory are used too. Functions performed on GPUs are called kernels. Each kernel needs configuration of a grid of blocks and each block requires a number of threads to be specified. Both the grid and blocks can have up to three dimensions. The computation is performed on several streaming multi-processors (SM) independently. The maximum number of blocks, threads and SM depends on the specific GPU.

4. Our approach

The main goal of our research was to develop a variant of the Fruchterman–Reingold algorithm graph layout algorithm which can run fast on the GPU and provides similar results as the original algorithm at the same time. The whole computation should be performed on GPU to avoid unnecessary memory transfers. So the proposed novel approach described below was performed only on GPU.

Fruchterman–Reingold is an iterative algorithm and it requires several thousand iterations to complete. It brought us to the idea to compute a different space-filling curve at each iteration. It is based on the fact that the layout algorithm induces dynamic changes. Thus the set of \(k\)-nearest neighbors of a selected vertex in a single iteration is changed in the consequent iteration. Finding the neighbors in each iteration is achieved by a small random move of each vertex (by jitter factor). This random shift causes that nearest neighbors are different in every iteration. The jitter factor is a small number dependant on SFC boundary box. Note that the bounding box must be large enough to ensure that all vertices stay inside. Then all repulsive forces are computed between vertex pairs in every part of the SFC and finally attracted forces between connected vertices are computed and vertices are visualized. Every single iteration consists of the following steps:

1. Random shift each point limited by jitter factor.
2. Compute SFC index for every single \(v \in V\).
3. Reorder \(V\) according to the SFC index (CUDA CUBB library was used).
4. Split the set of reordered vertices in \(V\) into SFC parts.
5. Within every SFC split part, the repulsive forces are computed for the set of vertices.
6. Calculate attractive forces and update the positions of all vertices \(V\).

4.1. Nearest neighbors with SFC in CUDA

The goal of every iteration is to adjust positions of each vertex. The positions of vertices are stored in the array. The first step in iteration is to order the vertices according to Hilbert (or Z-order) code as described [21]. Note that vertices are not physically moved, but instead only ordered indices are needed. The ordered list of vertices that lie on the SFC is split into several parts in the next step of the proposed method.

Fig. 2 shows 8 randomly placed vertices \(A, B, C, D, E, F, G\) and \(H\). In the beginning the vertices are stored in one dimensional array. The vertex \(A\) is at 0 position, vertex \(B\) is at the 1st position, etc. When the SFC is given then for each vertex position the SFC code can be easily computed. When the vertices have their SFC codes they can be ordered (for example by parallel quick sort or fast CUDA radix sort, etc.). When vertices (their SFC codes) are ordered in the array the nearest neighbors (according to SFC) lie next to each other.

4.2. Repulsive forces

Computation of the repulsive forces is the most time-consuming part of the Fruchterman–Reingold algorithm \(\Theta(N^3)\). A single repulsive force between two vertices decreasing when the distance between the vertices is increasing. So the forces that modify the positions of vertices can be neglected for all such vertex pairs where the vertices are far from each other. Omitting such forces and focusing on the forces between close vertices only significantly reduce the computation time and the time complexity in general. There exist many algorithms dealing with searching for all nearest neighbors (NN) or \(k\)-nearest neighbors. We refer to [14,13,22] for information on the NN problem.

Our proposed approach to speed up this part on GPU is a combination of the method described [21] and speed-up of the FR algorithm described in [2]. The details of our algorithm are shown in Fig. 2. At first the positions of vertices are copied to texture memory (read-only memory, but faster access than global memory). Another input to the kernel are indices which provide information on vertex order according to Hilbert/Z-order curve as described [16–18,22]. Every block then loads to shared memory (sData) vertex positions from texture (tex2D) based on indices. Each thread then performs the computation of repulsive forces in the shared memory context. Results are then stored to the sForces shared memory and then back to the global memory. Note that sData shared memory array is volatile to ensure that the compiler will not make any optimization such that some data will be stored in local caches (everything must be visible for all threads). The sForces array is not volatile because every thread accesses only its own memory address.

The pseudocode below illustrates the modification of the original Fruchterman–Reingold. The main difference lies in the part where the repulsive forces are computed. The graph is divided into parts as described above. Such division leads to inaccuracies in the computation of repulsive forces due to lower number of vertices in subgraphs. This side effect is eliminated by jittering, which enables certain vertices to fall in different parts than they would without jittering. The computation of the attractive forces and the final displacement is the same but it is done in parallel (Figs. 3 and 4).
area = W × L {width and length of the frame}
G = (V, E) {random initial positions for the vertices}
k = \sqrt{\text{area} / |V|}
function \( F_r(z) \) {return \( x^2 / k \)}
function \( F_a(z) \) {return \( k^2 / z \)}
{do it in parallel}
for i = 1 to iterations do
    for all \( v \in V \) do
        \( v = v + \text{jitterFactor} \)
        \( vIndex = SFC(v) \)
    end for
    \( V = \text{reorder}(V) \)
{calculate repulsive forces; a single CUDA Block processes a set of parts of the SFC; every part represents a subgraph \( V_i \subseteq V \), where |\( V_i \)| = SFC_SIZE.}
    for all \( i \in \text{noPartsPerBlock} \) do
        for all \( v \in V_i \) do
            \( v.disp = 0 \)
            for all \( u \in V_i \) do
                if \( u \neq v \) then
                    \( \Delta = v.pos - u.pos \)
                    \( \Delta \text{ is short hand for the difference} \)
                    \( (\Delta \text{ between positions of the two vertices}) \)
                    \( v.disp = v.disp - (\Delta / |\Delta|) \times F_a(\Delta) \)
                end if
            end for
        end for
    end for
{calculate attractive forces}
{limit the max displacement to the temperature \( T \)}
{and then prevent from being displaced outside frame}
end for

5. Experiments

In this section, achieved graph layouts are shown in Figs. 5–8. Hardware configuration and all parameters of the proposed algorithm are mentioned as well.

5.1. SFC Part Size

SFC Part Size is the number of vertices in a single cluster where all vertices in that cluster are considered to be near to each other. This number defines the size of GPU thread block; that is why it should be set with respect to the design of GPU kernels. Usually, the block size is a multiple of the warp size (32 threads) [19]. The SFC Part Size affects the speed and its optimal value depends on a particular GPU architecture. In the case of the smaller SFC Part Size, the algorithm requires more iterations to achieve the acceptable result. According to the experiments, the acceptable SFC Part Size ranges between 32 and 256 (see Fig. 5).

5.2. Iterations

The proposed algorithm as well as the original FR algorithm requires many iterations to achieve readable graph layout. The number of iterations is determined experimentally and it usually varies by thousands. Fig. 6 shows the certain number of iterations for the same graph. When the vertices stop moving or they slowly oscillate, the graph is in the stable position and the computation can be stopped.

5.3. Jittering

Jittering is one of the most important elements in the proposed algorithm. It could be informally defined as a random distortion of borders, e.g. resizing and moving them. If the computation is done without jittering, the rectangular patterns/clusters can be seen in the final layout (see Fig. 7). This effect depends on the level of details (LOD) [16] of the particular space-filling curve. If one vertex
Fig. 5. Graph layouts with different settings of SFC Part Size.

Fig. 6. Graph layouts after given numbers of iterations.
Fig. 7. The comparison between enabled and disabled SFC jittering.

Fig. 8. The comparison between different settings of the gravitation factor.
moves outside the rectangular bounding box (e.g. by applying repulsive forces), vertices in the neighboring cluster push this vertex back to the common border. These borders are regular and are defined by LOD. With the usage of the jittering, the common borders are disturbed randomly, which leads to more random nearest neighbors selection. Higher jittering factor causes more randomness in border distortion. The value of the jittering factor needs to be determined experimentally for the given data.

5.4. Gravitation

The gravitation factor (GF) affects the expansion of the graph layout. The default value is 1.0, which means that the vertices are not affected by the gravitation and only repulsive and attractive forces are taken into account. In the case of $GF < 1.0$, all vertices are affected by the gravitation that moves them closer to the space origin. In the case of $GF > 1.0$, the graph layout expands. The GF is set experimentally and one should avoid extreme values (see the right bottom layout in Fig. 8). The extreme setting leads to suppression of repulsive and attractive forces.

5.5. Performance

The last experiments were focused on the performance of the proposed algorithm. All the times mentioned in Figs. 9 and 10 are in milliseconds. Every test was run 10 times and the final computation times (including visualization) were averaged.

The source data were generated by Stanford Network Analysis Project (SNAP) [23]. In total, there were 20 experiments that differ in the number of vertices ($|V| \in \{10K, 100K, 1M\}$) and edges ($|E| \in \{1M, 2M, 4M, 8M, 16M\}$). Fig. 9 shows a computation time for every combination of $|V|$ and $|E|$.

Fig. 10 shows a computation time for every combination of $|V|$ and $|E|$.

5.6. Hardware and software configuration

All experiments run on the following hardware: AMD FX-8150 Eight-Core CPU @ 3.61 GHz, 32 GB RAM, Windows 8.1 64-bit. CUDA capable device NVIDIA TITAN (Kepler architecture) with CUDA 6.8.

6. Conclusion and future work

This paper described a novel algorithm based on the Fruchterman-Reingold force-based graph layout algorithm. The algorithm was implemented in parallel using CUDA architecture. From the visual point of view, the proposed algorithm achieved almost the same results as the original one. The novelization of the algorithm, especially the reduced cost of repulsive force computation which is now less than $o(N^2)$, makes sure the whole computation is now performed fast enough to compute layouts for graphs even with a million vertices in an acceptable time, which was not possible with the original version. Now we can quickly visualize large datasets like particle swarms, colony growth visualization in colony optimization algorithms, displaying huge numbers of generations of individuals in evolutionary algorithms, or even things outside the swarm intelligence research field like parts of social, communication or citation networks, complex molecules or web page interconnections. Our novel method for finding the nearest neighbors implemented on GPU also brings promising results in solution of the general $N$-body problem [24] and therefore our future work will go in this direction.

Acknowledgment

This work was supported by Project SP2015/146 “Parallel processing of Big data 2” of the Student Grant System, VŠB Technical University of Ostrava and the GACR Grant no. P103/15/067005 “Unconventional Control of Complex Systems”.

References

Bibliography


[121] M. Quinn. Parallel Programming in C with MPI and OpenMP. Mcgraw Hill Education (India) Private Limited, 7 2003.


