Distributive and parallel problem solving - methods, data structures and algorithms

Antal Tátrai

PhD Dissertation
Department of Computer Algebra
Eötvös Loránd University

Supervisor: Dr. Attila Kovács PhD
Associate Professor
PhD School of Informatics
Dr. András Benczúr D.Sc.
PhD Program of Basics and methodology of informatics
Dr. János Demetrovics
member of the Hungarian Academy of Sciences

2012
Acknowledgements

I would like to thank my PhD advisor Attila Kovács for supporting me during the past five years. Thanks for the project opportunities, for the technical and English supports, for the advice and hints that cannot be learnt from books.

I would like to thank my colleague Péter Burcsi for his help in my research and writing up my results.

I would like to thank Dr. Judit Gaizler Nyékyné, Prof. Tamás Roska and Prof. Péter Szolgay for their help and patience that they presented to me at Pázmány University.

Finally, I would like to say thank you to my wife for her patience and endless support.
## Contents

### List of Figures

1 Introduction
   1.1 Amdahl’s law .................................................. 1
   1.2 Communication overhead ...................................... 2
   1.3 Parallel architectures ....................................... 3
      1.3.1 Symmetric multiprocessing (SMP) architecture ........ 4
      1.3.2 GRID architecture ....................................... 6
      1.3.3 SMP with special devices ................................ 6
   1.4 The problem of concurrent start-up in Erlang systems ... 7
   1.5 The problem of cache cleaning ............................... 10
   1.6 The problem of classifications through generalized number systems ... 11

2 Concurrent start-up of Erlang systems
   2.1 Erlang/OTP .................................................... 13
      2.1.1 Code structure and execution ........................... 13
      2.1.2 Design principles ...................................... 14
         2.1.2.1 Supervision trees .................................. 14
         2.1.2.2 Behaviours ......................................... 15
   2.2 Basic idea of the solution ................................... 15
      2.2.1 Dependency graph ...................................... 16
      2.2.2 Concurrent startup of a supervisor’s children .......... 17
   2.3 The prototype of the solution ................................ 19
      2.3.1 Realization of the dependency graph .................... 19
      2.3.2 Condition server ........................................ 21
<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>4.3.4 Implementation</td>
<td>75</td>
</tr>
<tr>
<td>4.3.4.1 Concurrent hash sets</td>
<td>75</td>
</tr>
<tr>
<td>4.3.4.2 Barriers</td>
<td>76</td>
</tr>
<tr>
<td>4.3.4.3 Arithmetic vector operations</td>
<td>76</td>
</tr>
<tr>
<td>4.3.5 Results</td>
<td>76</td>
</tr>
<tr>
<td>4.3.6 Conclusion</td>
<td>79</td>
</tr>
<tr>
<td>Summary</td>
<td>80</td>
</tr>
<tr>
<td>Összefoglalás</td>
<td>82</td>
</tr>
<tr>
<td>Publications of the author</td>
<td>84</td>
</tr>
<tr>
<td>References</td>
<td>85</td>
</tr>
</tbody>
</table>
List of Figures

1.1 An SMP system with 3 processors connected to the same memory module through a system bus or a crossbar. .................................................. 4
1.2 The CELL processor architecture .................................................. 7

2.1 Inserting a dummy supervisor node for (1) preserving the supervisor’s restarting behaviour, and (2) enabling fast parallel start-up. Supervisors are denoted by squares, permanent processes by continuous border, and temporary processes by dashed border. The living processes \( (s_2, w_2) \) can be seen in the middle of the tree after the termination of the dummy functions. ................................................................. 18
2.2 Startup speed of the sequential and concurrent versions (in seconds). In concurrent case we can put different number of fork points into different places in the process tree. The author created several concurrent cases for each kind of trees. The worst and best case values represent the slowest and the fastest concurrent start-up time in the proper kind of trees. . . . . . 26
2.3 Start-up speed (in seconds) depending on the number of running processes, which can be set by the fork points. The levels show the depth of the fork points. ................................................................. 27
2.4 Start-up speed (in seconds) depending on the depth of the fork points. . 27

3.1 A schematic picture about a cache implementation. Threads try to process insert and find instructions on a hash table guarded by a lock. The clearer thread tries to clear the old elements from the same hash table. . 30
3.2 A cache with binary heap as secondary structure ............................. 33
3.3 A cache with bucket list as secondary structure ............................. 35
LIST OF FIGURES

3.4 Performances .................................................. 37
3.5 Transactions vs. Size of Cache (iMac) .......................... 38
3.6 Transactions vs. Number of Threads (iMac) ..................... 38

4.1 On the picture a rectangle, a line or a point represent a brick in the second dimension. a) the numbers mean the order of bricks. b) the numbers show the new bricks of a prism. For example the bricks numbered 1, 2 and 3 create a prism, and during calculating of bricks with number 3, no more iterations are needed, if the $\phi$ function arrived in a prism created in bricks numbered 1 and 2. ............................................. 50
4.2 ........................................................................... 50
4.3 ........................................................................... 51
4.4 ........................................................................... 51
4.5 ........................................................................... 51
4.6 ........................................................................... 51
4.7 ........................................................................... 51
4.8 ........................................................................... 52
4.9 Speed-up ratio between the two algorithms ................. 58
4.10 Measured running time on the LS22 ......................... 77
4.11 Measured running time on both platforms (LS22, QS22) ................................. 78
4.12 Measured running time depending on the number of points sent ........ 79
Chapter 1

Introduction

Nowadays multiprocessor computers become more and more conventional. Computers with hundreds or thousands of processors are common in the industry, and probably your desktop PC at home has more than one processor as well. On the other hand, there are several problems regarding the sequential solutions, which often do not satisfy the needed performance issues. One of the main challenges in the future’s computer science is developing efficient and safe parallel algorithms instead of the sequential ones. The methodologies, practices and algorithms that are never-falling in the sequential world usually waste the power of the parallel architectures. Well-known examples are reference counting and garbage collecting.

There are several challenges in the field of multiprocessor programming. The existing solutions depend on the details of the hardware that were invisible in sequential case. For example programs for multiprocessor chips use threads and the cache coherency problem is resolved by the hardware without any software trick. In case of large scale systems usually we have to develop solutions for routing or other jobs that are independent from the original problem, consequently, our algorithms can be scalable very hardly.

Parallel computers are asynchronous: threads can be halted or delayed without warning by interrupts, preemption, cache misses, page faults or other events. These events are unpredictable. While in sequential case a killed or a delayed activity does not disturb other activities, in parallel case a halted thread may stop the whole task, and the interruption of a thread can produce delay in several other threads. Sequential programming has a history of six decades, precisely defined methodologies, data structures, primitive statements such as loops and conditions. In the parallel case we
have several well-defined building blocks but usually we just have complicated methodologies, half-defined data structures and algorithms that are bounded by the number of threads or having other restrictions. We design and implement sequential programs step-by-step: if we want to extend out existing code with a new component then we can do that without breaking the original one. Usually we cannot do this in parallel programs, since a new method sometimes requires the modification of the existing ones. (An example can be a linked list with an insert method. If we implement the erase function usually we must redesign our insert algorithm as well.)

The above mentioned ideas show that the speed up of an application by adding new processors to the system is much more difficult than simply by increasing the clock frequency of the CPU. But what is the maximal speed up that we can reach with parallel algorithms? Amdahl described an estimation about the relative power of parallel algorithms in 1967 [5, 6].

1.1 Amdahl’s law

Suppose that we upgrade our uniprocessor machine with an \( n \)-way multiprocessor chip. We like to see an \( n \)-fold increase in computational power. In practice, however, this never happens. The fundamental reason is that the real-world problems cannot be effectively parallelized. Amdahl’s law shows that the speed-up of a complex job is limited by the parts of the job which have to be executed sequentially.

Let \( S \) be the speed-up of a job, which is defined as the ratio between the running time of the sequential and parallel algorithms. Let \( p \) be the fraction of the job that we can compute parallel on all \( n \) processors. Assume that the sequential running time is 1. In this case the parallel running time is the following:

\[
1 - p + \frac{p}{n}.
\]

Amdahl’s law says that the speed-up between the two kinds of algorithms is

\[
S = \frac{1}{1 - p + \frac{p}{n}}.
\]

In this case we can easily calculate the theoretic maximal speed-up of a problem.
Let us see an example. Suppose that \( \frac{5}{6} \) parts of the algorithm can be parallelized and we have 5 processors.

\[
S = \frac{1}{1 - \frac{p}{n}} = \frac{1}{\frac{1}{6} + \frac{1}{6}} = 3.
\]

None the less we have 5 processors the maximum of the speed-up is 3. Naturally, in the real-world there are many other factors that make the speed-up smaller. We note that Amdahl’s law describes the speed-up of algorithms that work on general purpose processors. For special processors (such as GPUs, IBM Cell, etc.) the speed-up can be computed by the generalization of Amdahl’s law \([7]\).

### 1.2 Communication overhead

Amdahl’s law describes the theoretic speed-up without any communication overhead. Unfortunately, this cost of the parallel algorithms is usually significant. The modern processors have multilevel cache memories that contain the copy of the main memory for reaching the desired data faster. If the processor executes an instruction that depends on data from the memory it looks first for it in the level 1 (L1) cache (primary cache). This memory is very small and the CPU can reach it very fast. If the required data is not in the L1 cache than it tries to load it from the level 2 (L2) cache (secondary cache). And finally, if the data is not in the L2 cache, the processor loads it from the main memory. Naturally, the speed of an algorithm is highly depending on how the compiler and the hardware’s pipeline optimize the data moving between the caches and the main memory.

A normal processor of nowadays’ computers performs an add instruction during 1 clock cycle but moving the data from the main memory to the processor’s registers takes nearly 10 clock cycles. (Naturally, the processor’s pipeline starts to fetch the data when it recognizes an add instruction.) Another (more abstract) example for this overhead is the hard drive handling. The effectiveness of a database algorithm can be measured with the number of written and read hard drive blocks. These examples show that the communication cost of an algorithm is significant both in parallel and in sequential cases. External memory algorithms contains several parameters which are releated to the sizes of the levels in the memory hierarchy. In \([8]\) it was examined how these values should chooes in order to obtain an optimal running time.
1.3 Parallel architectures

In this section three architectures will be described that constitute the basic building blocks for analyzing distributive and parallel problem solving strategies, methods, data structures and algorithms.

1.3.1 Symmetric multiprocessing (SMP) architecture

This architecture consists of general purpose processors (CPUs) and a shared memory. The multi core chips are typical SMP architectures (see 1.1). The speed of the parallel algorithms of this architecture mainly depends on two factors.

![SMP system with 3 processors connected to the same memory module through a system bus or a crossbar.]

The first is the data moving like in the sequential case. In high level programming languages (such as C/C++) we cannot code cache handling instructions (just with assembly blocks that usually break the portability of the code), and we have just very limited opportunity to modify this aspect of the program. For example the following structure and array

```c
struct coordinate{
    float x, y, z;
};

coordinate coordinates[100];
```

usually produce faster code than
float x_coordinates[100];
float y_coordinates[100];
float z_coordinates[100];

Since the structure is stored probably in consecutive words, if we modify the x, y
and z coordinates together then the processor finds the y and z coordinates in the L1
cache after modifying the x coordinate. In the second case the modification of the
three coordinates requires three fetchings from the memory. (Sometimes these kinds of
optimizations cause significant speed-up.)

The second factor that determines the speed of a parallel program on SMP platform
is the cache coherency. If a processor modifies a piece of data and another processor
has a copy of this data in one of his caches then synchronization must be done between
the two processors caches. This synchronization is performed automatically on several
modern processors and it is generally slow. In contrast to the data fetching problem, in
high level languages the cache consistency problem can be avoided. The problem can
be resolved by hardware dependent or hardware independent ways. Suppose that we
have two threads and we want to add 200 to each coordinate in the coordinates array.

A possible algorithmic design can be that the first thread performs the modification
on the odd elements and the second does it on the even elements of the array. This
algorithm will cause several consistency errors on the major part of processors because
each coordinate element of the array will be fetched together with the following el-
ement. A hardware dependent solution for this problem is by adding some padding
bytes to the coordinate structure. In this case the cache loads some unused data from
memory, but it does not load more than one coordinate. A much better hardware inde-
pendent solution can be the following: the first thread performs the modifications from
the beginning of the array to the middle, and the second one from the middle to the
end. Both threads have the same amount of job but the modifications don’t interfere
with each other.

For the sake of completeness we mention that there are other architectures either
non-uniform memory access architectures in which different memory bandwidth dedi-
cates to different processors, or in which not all memory is available to all processors.
1.3.2 GRID architecture

The GRID architecture is not a single computer but a networked system of computers. Usually it is a set of PCs that are connected via Ethernet or other kinds of communication interface. It has no topological requirements, consequently the GRID’s nodes usually know nothing about the other nodes. Since a simple node may have more than one processor, the GRID node can be regarded as an SMP architecture. But in case of a GRID there is a much slower connection between the processors: the network. Since GRID nodes are connected through a network interface the data moving is programmable. Naturally, the speed of a GRID algorithm depends on both the speed of the algorithm that runs on the nodes and the communication overhead among the nodes.

The most common GRID algorithm schema is the master-slave model. In this model there is a master node that does not perform any calculation but organizes the jobs of the slave nodes. The slaves receive their input and after processing it they send the result to the master. If the solution of a problem requires only large calculations then the speed of the GRID algorithm depends on the speed of the calculation algorithm, otherwise the communication cost is significant.

GRID algorithms have to be fault tolerant because any node may crash or may be switched off. GRID algorithms have to manage these situations. The master-slave scheme is very powerful since it can handle the previously mentioned problem relative easily.

Grid computing involves computation in a distributed way which may also involves the aggregation of large-scale clusters.

1.3.3 SMP with special devices

This architecture is based on the SMP architecture but there are some programmable devices that are connected to the CPUs. The IBM Cell Broadband architecture is an example for this architecture, as the CPUs have Synergistic Processing Unit (SPU) cores. Another typical example is a simple PC with graphics card. The common attributes of these devices are the following:

- Perform arithmetic operations much faster than the CPU (usually they are vector processors),
they are programmable,

• there is a communication overhead between the main memory and the devices.

The speed of this architecture — similarly to the GRID — depends on the computation time and the data moving time. Efficient algorithms usually perform heavy calculations between data movings.

1.4 The problem of concurrent start-up in Erlang systems

In this section we summarize the basics of the Erlang systems and the problem statement of the system start up will be described.

Several aspects of concurrent systems written in message passing languages have been already studied including garbage collection [10], heap architectures [11], or memory management [12]. We present the problem of start-up concurrency.

Figure 1.2: The CELL processor architecture [9]

In the first thesis group a solution for parallel start up of applications written in Erlang/OTP will be presented. This can be run on SMP platforms (it was tested on an SMP machine), but Erlang can run in distributed environment as well. The second thesis group describes a software cache running on an SMP machine. The third thesis group — parallelizations of a number theoretic problems — uses all platforms mentioned above.

1.4 The problem of concurrent start-up in Erlang systems

In this section we summarize the basics of the Erlang systems and the problem statement of the system start up will be described.
1.4 The problem of concurrent start-up in Erlang systems

Now we turn our attention to the distributed programming language Erlang. Erlang was designed by the telecommunication company Ericsson to support fault-tolerant systems running in soft real-time mode. Programs in Erlang consist of functions stored in modules. Functions can execute concurrently in lightweight processes, and communicate with each other through asynchronous message passing. Forms of messages include function invocation, signals, and data packets. Computation based models on message passing include the actor model and process algebras [13]. The creation and deletion of processes require little memory and computation time. Erlang is an open source development system having a distributed kernel [13]. Erlang is implemented on “GRID-like” systems: each Erlang node is a separated program, but we can start several nodes both on an SMP architecture computer or on a GRID.

The Erlang system has a set of libraries that provide building primitives for larger systems. They include routines for I/O, file management, and list handling. In practice Erlang is most often used together with the library called the Open Telecom Platform (OTP). OTP consists of a development system platform for building, and a control system platform for running telecommunication applications. It has a set of design principles (behaviours), which together with middleware applications yield building blocks for scalable robust real time systems. Supervision, restart, and configuration mechanisms are provided. Various mechanisms, like an ORB, facilitate the development of CORBA based management systems. Interfaces towards other languages include a Java interface, an interface allowing Erlang programs to call C modules, etc. These interfaces are complemented with the possibility of defining IDL interfaces, through which code can be generated. The number of Erlang/OTP applications and libraries is continuously expanding. There are for example SNMP agents, a fault tolerant HTTP server, a distributed relational database called Mnesia, etc. One of the largest industrial applications developed in Erlang/OTP is the AXD 301 carrier-class multi-service (ATM, IP, Frame-relay, etc.) switching system of Ericsson. It is a robust and flexible system that can be used in several places of networks. It has been developed for more than 10 years (for an early announcement of the product, see [15]), resulting in a long product line. It contains several thousand Erlang modules and more than a million lines of code.

The first question is why the investigation of the start-up phase was important?

- Critical distributed systems often have the maintainability requirement of 99.999 availability, also known as the "five nines". In order to comply with the "five
nines" requirement over the course of a year, the total boot time may not take more than 5.25 minutes. In practice, due to the maintenance process, every system has a planned down time. In this case a fast and reliable start-up is a must.

- During system and performance testing, when the system is frequently started and stopped, fast start-up might be beneficial.
- The start-up time is not the only reason to study the start-up phase. In most product lines the requirements (and therefore the code) alter continuously. The changes may influence the code structure, which may affect the execution order of the parts. Although the code can often be reloaded without stopping the system, the changes may influence the start-up. The challenge is to give a generic solution which supports reliable, robust and fast start-up even when some software and/or hardware parts of the system had been changed.

How is the start-up of an Erlang application performed? The traditional startup provided by the Erlang/OTP library is sequential. It was not designed to start as quickly as possible, no special attention was paid to the possibility of parallelizing the different operations performed during start-up. The only order imposed is due to the explicit dependencies described in the application configuration files. Technically, the reason of the sequential start-up is that each process performing an OTP behaviour sends an ACK (acknowledge) signal to its parent only after the whole initialization process is finished. It means that each process has implicit preconditions. In the concurrent case, maintaining these preconditions is a fundamental problem. The proposed solution enables the concurrent start-up and provides an Erlang/OTP extension for describing and realizing preconditions between behaviour processes. Hence the start-up will not only be fast but remains reliable as well. The use of conditions to construct dependency graphs to manage the order of start-up bears a resemblance to the mechanism used by Apple’s MacOSX StartupItems. Each StartupItem includes a property list of items that provide/require/use other items, which are used by the SystemStarter to build a soft dependency graph controlling the order of starting items [16]. There does not exist any such mechanism in Erlang/OTP.

Of course, the start-up times do not only depend upon the dependencies among the applications and the degree to which these start-up activities can be parallelized. The start-up times are affected by several other factors, probably the most significant
1.5 The problem of cache cleaning

Computer systems often use caches for reaching slowly accessible data much faster. One of the most typical examples is the Domain Name Service. On the Internet, the data routing is handled by IP addresses. But users usually know just the name of a server (such as google\.com). When we try to connect to a server defined by its name, the application (browser or else) asks its domain name server. In case of google\.com, we try to get its IP address from the .com dns server. Naturally, our query walks through several other dns servers. Usually, the response to our query arrives from a server closer to us instead of the original name server of the target domain, as the data is stored in the cache of the server closer to us.

Since these data are just a copy of the original, the validity has to be rechecked periodically. Usually each data saved into a cache has a special attribute called Time To Live (TTL). This is a time value and means the validity duration of the data. When an element has expired it has to be dropped out from the cache or fetched again. Often this functionality is implemented with a special thread that starts periodically and checks the elements in the cache.

A cache can be defined as an abstract data type which can be implemented in several ways. In most cases trees and hash tables are the bases of this data type. Usually the caches have three operations: inserting an element into the cache, finding an element and erasing expired elements from the cache. The last operation will be called cleaning or regeneration. An additional operation can be the final erasing of an element from the cache (independently from its TTL value). Usually, the cleaning thread has to read the whole data structure.

An early lock based parallel hash table implementation was described by C. Sch. Ellis [17][18]. An analogous solution was used in the thesis. But several other implemen-
1.6 The problem of classifications through generalized number systems

The problem of classifications through generalized number systems exist. We say an object is lock-free if infinitely often some method call finished in fine number of steps. Lock-free hash-tables are free from deadlock and often have robust performance. The parallel lock-free hash-tables have several challenges. First of all the lock-free linked list. A complicated list was described by J. D. Valois [19]. Later T. Harris published an other list implementation [20] that is the base of numerous hash table implementations. These publications use reference counting or garbage collector mechanisms when deleting a node that has poor performance in parallel programs. M. M. Michel published an other memory reclamation scheme [21] that does not use reference counting and a lock-free hash table [22] that bases its memory reclamation and Harris’ linked list. The lack of Michael’s hash table is that the backbone cannot change its size. Consequently the list can be very long, which reduces the finding speed. O. Shalev and N. Shavit showed a resizable, lock-free hash table [23] that also bases on Harris’ list implementation and Michael memory reclamation scheme. Some other hash table implementations were also studied [24, 25] but they have performance or functional issues.

In this thesis group we focus on the design and implementation of parallel data structures in which the cleaning thread is able to reach the expired elements without reading the whole data structure. Once this decreases the running time of cleaning thread and probably increases the speed of search and insert functionality during the cleaning.

1.6 The problem of classifications through generalized number systems

The number systems are coeval with mankind. In the everyday life we use several number systems for counting the time or our money. The universe of the “original” number systems is usually $\mathbb{N}$. In this case we say that a base $(q > 1 \in \mathbb{N})$ and a set of integer numbers $D \subset \mathbb{N}$ form a number system if each $n \in \mathbb{N}$ has a unique finite representation in the form of $n = \sum_{i=0}^{k} d_i q^i$ where $d_i \in D$. If the digit set has the values $\{0, 1, 2, ..., q-1\}$ then it is called canonical digit set.

Naturally, with a $q > 1$ base and a canonical digit set pair, we cannot describe negative numbers but we can find number systems on $\mathbb{Z}$ that have either a negative base or negative digits (or both).
D. E. Knuth produced an extension of the number system idea. He described the number system on Gaussian integers \(G = \{a + bi \mid a, b \in \mathbb{Z}\}\). He used the complex number \(-1 + i\) for base with the canonical digit set \(D = \{0, 1\}\) and he found that this is a number system. He also tried to use \(2i\) as base with canonical digit set but this pair does not form a number system. Later Kátai and Szabó solved the problem completely. They found that numbers \(-A \pm i\) with the canonical digit set form number systems and there are no more canonical number systems on \(G\).

There is an extension of the previous idea. Get \(n\) linear independent vectors from \(\mathbb{R}^n\). Let \(\Lambda\) be the lattice that is generated by the previously mentioned vectors. Let \(M\) be a \(\Lambda \rightarrow \Lambda\) function and let \(D \subseteq \Lambda\) be a complete residue system modulo \(M\) \((0 \in D\)\). We say that the \((\Lambda, M, D)\) is a number system iff each \(x \in \Lambda\) has a unique representation of the form \(x = \sum_{i=0}^{k} M^id_{ji}\) with \(d_{ji} \in D, d_{jk} \neq 0\) for some \(k \in \mathbb{N}\).

The first question is how to decide that a given \((\Lambda, M, D)\) is a number system. There are known algorithms for solving the problem but in some cases the decision algorithm produces an extremely large set of lattice points. We design and implement algorithms using parallel architectures in order to overcome the mentioned phenomenon.

Second, in case of \((\Lambda, M, D)\) is not a number system, determine the “witnesses”, by which a classification of \(\Lambda\) can be made.

In this thesis group we focus on the decision and classification problems of \((\Lambda, M, D)\). The existing solutions are all sequential. We presented some solutions that run on GIRD or parallel SMP platforms.
Chapter 2

Concurrent start-up of Erlang systems

2.1 Erlang/OTP

In this section a short description of Erlang/OTP concepts is given. The overview begins with a few Erlang language features, then OTP design principles and the startup mechanism are discussed. For a full description of Erlang with many examples the author refers to the books \[31, 32\] and to the on-line documentation \[14\].

2.1.1 Code structure and execution

The code written in Erlang is structured as follows:

- Functions are grouped together in source files called modules. Functions that are used by other modules are exported, modules that use them must import them (or alternatively, have to use the \texttt{apply(Mod, Fun, Arg)} built-in function, or the \texttt{module_name:function_name (args)} form).

- Modules that implement some specific functionality together form an application. Applications can be started and stopped separately, and can be reused as parts of other systems. Applications do not only provide program or process structure but usually a directory structure as well. There is a descriptor file for each application containing the module names, starting parameters and many other data belonging to the application.
2.1 Erlang/OTP

- A *release*, which is the highest layer, may contain several applications. It is a complete system which contains a subset of Erlang/OTP applications and a set of user-specific applications. A release is described by a specific file, called release resource file. The release resource file can be used for generating *boot_scripts* for the system, and creating a *package* from it. After creating a *boot_script*, the system is able to start. First, the Erlang kernel is loaded. Then, a specific *gen_server* module (*application_controller*) is started. This module reads the application descriptor files sequentially, and creates a process called *application_master* for each application. The *application_master* starts the corresponding application, and sends an ACK signal back when the start-up is finished. Thus, as it was mentioned earlier, the Erlang/OTP startup is sequential.

The central concept of the execution is the process. As Erlang is message-oriented, executing Erlang code means creating strongly isolated processes that can only interact through message passing. Process creation, which is a lightweight operation, can be performed using the *spawn* family of functions. These functions create a parallel process and return immediately with the process ID (Pid). When a process is created this way, we say that it is spawned. Erlang messages are sent in the form Pid!Msg and are received using *receive*.

2.1.2 Design principles

One of the most useful features in OTP is to have a pre-defined set of design patterns, called *behaviours*. These patterns were designed to provide an easy-to-use application interface for typical telecommunication applications such as client-server connections or finite state machines. In order to realize highly available and fault-tolerant systems, OTP offers a possibility to structure the processes into *supervision trees*.

2.1.2.1 Supervision trees

A principal OTP concept is to organize program execution into trees of processes, called supervision trees. Supervision trees have nodes that are either *workers* (leaves of the tree) or *supervisors* (internal nodes). The workers are Erlang processes which perform the functionality of the system, while supervisors start, stop, and monitor their child processes. Supervisor nodes can make decisions on what to do if an error occurs.
Supervision tasks have a generic and a specific part. The generic part is responsible e.g. for the contact with the children, while the specific part defines (among other things) the restarting strategy. It is desirable that workers have a uniform interface, therefore OTP defines several behaviours with the same communication interface.

2.1.2.2 Behaviours

Behaviours, like every design pattern, provide a repeatable solution to commonly occurring problems. For example, a large number of simple server applications share common parts. Behaviours implement these common parts. A server code is then divided into a generic and a specific part. The generic part might contain the main loop of the server that is waiting for messages, and the specific part of the code contains what the server should do if a particular message arrives. In practice, only a callback module has to be implemented. OTP expects the existence of some functions (e.g. handle_call) in this module. As an example, several callback functions can be implemented for the complete functionality of an application, but the most important ones are: start/2, stop/1.

Let us summarize the most significant OTP behaviours: gen_server, gen_fsm, gen_event and supervisor. Each of them implements a basic pattern. The gen_server is the generic part of a server process, the gen_event is the generic part of event handling, the gen_fsm is the generic part of finite state machines. The supervisor behaviour is the generic part of the supervisor nodes of the supervision tree. Its callback module only contains the function init(Arg), in which the children and the working strategy of the node can be specified. The gen_server behaviour also defines higher level functions for messaging, such as the synchronous (call) or asynchronous (cast) messages.

2.2 Basic idea of the solution

Let us suppose that we have an Erlang system and we plan to make the startup concurrent. If we use the spawn function instead of the built-in methods of supervisor child-starting then the spawned processes run parallel, but the Erlang/OTP supervisor monitoring mechanism – one of the strongest Erlang/OTP features – is lost. Omitting the ACK mechanism from the built-in child-starting process would mean a deep
redesign and reimplementation of the OTP (the ACK mechanism corresponds to sequential child-starting). Also, sequential start determines an order between processes, which would vanish using a too naive way of parallelization. Therefore an alternative parallel ordering is required to avoid dead-locks and startup crashes.

In the light of the previously mentioned properties the following guidelines were defined:

- The supervision tree structure, as well as other functionalities, must be preserved.
- The startup must be reliable and fast (faster than sequential).
- Only "small" modifications are permitted in the Erlang/OTP stdlib.

### 2.2.1 Dependency graph

In this subsection we consider dependence relations between modules and introduce the notion of dynamic dependency graphs.

In order to preserve the supervisor tree structure, we define *conditions*. Conditions represent the startup state of modules. A condition related to a module is false while the module’s startup is being processed (or has yet to begin) and set to true when the corresponding startup has been finished. At the beginning of the startup all conditions are false. Conditions that the startup of another module depends on are called the preconditions of that module. A process can only start if all its preconditions are true. We can represent these relations in a dependency graph. Modules (or corresponding conditions) are the vertices, dependence between modules (or preconditions) are the directed edges in this graph.

When a behaviour module starts the first user defined instruction (which is also the first that can imply preconditions) it is the first instruction of the module’s *init* function. Therefore the verification of the preconditions and setting up the completed conditions to true have to insert immediately before and after executing the *init* function.

Dependency graphs are widely used in computer science. For example, dependency graphs are applied in the startup of the Mac OSX operating system or in compiler

---

1Here we remark that during the sequential startup there exist implicit preconditions which are described in the hierarchy of the supervisor trees.
2.2 Basic idea of the solution

optimization [33]. Moreover, a dependency graph is created when the Erlang boot script is generated from a given application.

However, there is a significant difference between the graphs above and our graph. The same Erlang software start-up runs in different ways in different environments, therefore module parameters, execution and dependencies can vary and should be handled dynamically for full performance. In order to keep Erlang’s robustness, one more guideline is added to the above:

- The dependency graph should be dynamic.

2.2.2 Concurrent startup of a supervisor’s children

We also propose an Erlang trick that enables starting processes in a concurrent way. Here we can define which nodes should start concurrently. When a supervisor process \( s \) starts a child process \( w_1 \), the system starts a dummy (or wrapper) node \( s_1 \) instead. Then, the dummy process \( s_1 \) starts a simple function \( s_{1,f} \) (which just calls a spawn function) and sends an ACK message back immediately to its parent \( s \). Consequently, the next child \( w_2 \) of the supervisor node \( s \) can start. So far function \( s_{1,f} \) has spawned the function \( f \). The spawned function \( f \) starts process \( w_1 \) and attaches it into the dummy process \( s_1 \) using the supervisor::start_child function\(^1\). The already started dummy process \( s_1 \) runs independently (parallel) from the other parts of the system. The advantage of this method is that if process \( w_1 \) has a blocking precondition then only \( w_1 \) is waiting instead of the whole system. Figure 2.1 shows the described supervision hierarchy after start. The dummy supervisor node’s restart strategy can be set in such a way that a crashing child results in the termination of the dummy supervisor. Thus the connection between \( s \) and \( w_1 \) is preserved.

\(^1\)The supervisor::start_child function is going to wait until the ACK message has been sent back.
2.2 Basic idea of the solution

Figure 2.1: Inserting a dummy supervisor node for (1) preserving the supervisor’s restarting behaviour, and (2) enabling fast parallel start-up. Supervisors are denoted by squares, permanent processes by continuous border, and temporary processes by dashed border. The living processes \((s_2, w_2)\) can be seen in the middle of the tree after the termination of the dummy functions.

The following code fragment shows the concurrent start-up of a supervisor’s children.

```erlang
-module(dummy_sup_tree).

dummy_child({Tree_id, Child_spec}) ->
    spawn(dummy_sup_tree, child_starter, [{Tree_id, Child_spec}],
    {ok, self()}).

child_starter({Tree_id, Child_spec}) ->
    supervisor:start_child(Tree_id, Child_spec),
    ok.

start_link({Child_spec}) ->
    supervisor:start_link(dummy_sup_tree, [{Child_spec}]).

init([{Child_spec}]) ->
```
Sup_flags = {one_for_one, 0, 1},
{ok,
{Sup_flags,
[
 {dummy_child_id, {dummy_sup_tree, dummy_child,
    [{self()}, Child_spec]}}, temporary, brutal_kill,
    worker, [dummy_sup_tree, generic_server]}
]
}
).

The init function is the call back function of the dummy supervisor \((s_1, s_2)\). The dummy_child function (started by init) implements \((s_1, f)\), child_starter realizes \(f\). Finally the original \(w_1\) child will be started in child_starter's supervisor:start_child line. The child specification of \(w_1\) is passed to dummy supervisor through init's parameter by supervisor \(s\).

### 2.3 The prototype of the solution

In this section the details of the solution is given by describing the skeleton of a prototype. We discuss the implementation of the dependency graph and how the Erlang boot script, the supervisors’ init function, stdlib modules, etc. should be modified.

#### 2.3.1 Realization of the dependency graph

The dependency graph is implemented as a module called release_graph. This module implements and exports the following functions: get_conditions, get_preconditions and get_condition_groups.

The get_conditions function returns a list of pairs. Each pair consists of a module name (with parameters) and a condition name.

\[
\{ \{Mod, Args\} , \text{condition_name} \}. \]

We note that the function tag of the MFA (Module-Function-Arguments triplet) may be omitted, since it is always the init function of the module. The function get_conditions corresponds to the vertices of the dependency graph. Observe that a condition corresponds to a module together with parameters rather than to a module. This is
2.3 The prototype of the solution

in accordance with our dynamic dependency graph guideline. In general, the Args parameter can be an actual parameter value or undefined. In the latter case the condition describes the module’s startup with arbitrary parameters.

The get_preconditions function also returns a list. The elements of the list have the following structure:

\[
\{ \{ \text{Mod}, \text{Args} \} , [ \text{condition_names} ] \}.
\]

The function corresponds to the edges of the dependency graph. When a module’s init function is called then the validity of the conditions in the list must be tested. Once again, the Args parameter can be undefined meaning that the startup of this module with any parameters has to wait until all conditions in the list become true.

The third function facilitates the management of dependence relations. Huge systems are likely to have many conditions and these conditions can be organized into groups. The get_condition_groups function returns a list of pairs of the form

\[
\{ \text{condition_group_name} , [ \text{conditions} ] \}.
\]

One can use the condition_group_name instead of the conditions defined in the list.

We remark that the dependency graph is not necessarily connected. Some modules are not preconditions of any other modules. In this case the definitions of the corresponding conditions are superfluous. Other modules do not have any preconditions, consequently they can be omitted from the return value of the get_precondition function.

Let’s see an example. Let two applications app1 and app2 be given. The first has 3 server nodes that are controlled by a supervisor node. There is another server in the second application which has to wait for the complete startup of the first application. A possible implementation of the above functions might be:

...  
get_conditions() ->

[  
  { {app1_rootsup , undefined} , cond_app1_rootsup },  
  { {generic_server , [{app1_server1}]} , cond_app1_server1 },  
  { {generic_server , [{app1_server2}]} , cond_app1_server2 },  
  { {generic_server , [{app1_server3}]} , cond_app1_server3 }  
].
2.3 The prototype of the solution

get_condition_groups() ->
    [{group_app1_app, [cond_app1_server1, cond_app1_server2, cond_app1_server3, cond_app1_rootsup]}].

get_preconditions() ->
    [{generic_server, [{app2_server1}]} , [group_app1_app]}].

2.3.2 Condition server

The startup is controlled by a special server, called condition_server, which is started during the Erlang main system start. It stores and handles the dependency graph of the user programs. It also finds and loads the release_graph module and checks the validity of the data in it (checks for mistypes, not existing condition names, etc.). Clearly, any error in the Args fields remains undiscovered. If the Args tags are all undefined then the dependency graph is independent from the dynamic data. In this case, an acyclic dependency graph assures a dead-lock free structure if each node having preconditions is started in a concurrent way.

The condition_server performs the following two tasks based on the dependency graph. (1) First, it sets the conditions belonging to the \{M,A\}s to true. This is implemented in the set_condition({M,A}) function. (2) Second, it blocks the caller process until all its preconditions are satisfied. This is implemented in the wait_for_conditions({M,A}) function. These functions have to be called by the generic parts of the behaviours (independently of the users’ programs). Consequently, the condition_server must be implemented without the gen_server behaviour.

We remark that for those modules which don’t have any preconditions or don’t belong to any other module’s precondition, the corresponding function call has no effect.

The condition_server module has to be a part of the Erlang kernel modules, since during the Erlang system’s startup several event handler and server modules are started, and they require access to the condition storage system.
2.3 The prototype of the solution

The next code snippet is the main code of `condition_server`:

```erlang
-module(condition_server).

...

set_condition(Mfa) ->
    condition_server ! {set_condition, Mfa, self()},
    receive
        ok -> ok
        end.

wait_for_conditions(Mfa) ->
    condition_server ! {is_conditions_ready, Mfa, self()},
    receive
        ok -> ok;
        wait ->
            timer:sleep(100),
            wait_for_conditions(Mfa)
        end.

init(Parent) ->
    process_flag(trap_exit, true),
    %% Checking condition graph!
    ...

    case digraph_utils:is_acyclic(Graph) of
        false -> Parent ! {error, system_graph_is_cyclic};
        true ->
            State = #cs_state{conditions = Conditions, 
                              condition_groups = ConditionGroups, 
                              preconditions = Preconditions, 
                              ready_conditions = gb_sets:empty() 
```
{app2_server1, {generic_server, start_link, [app2_server1]},
  permanent, 10, worker, [generic_server], concurrent}

If the generic part of a supervisor interprets a concurrent child specification it starts a
dummy supervisor node with the proper parameters instead of the original child.
2.4 Implementation and measurements

2.3.4 Further modifications of the Erlang system

It is also necessary to modify each Erlang behaviour before the callback `init` function is called, and after it returns successfully. These modifications are put into the `gen_server`, `gen_event`, `gen_fsm`, `supervisor_bridge` and `supervisor` behaviour.

The built-in utilities create boot scripts which do not start the `condition_server` automatically. In order to start the server, a new line has to be inserted into the boot script. The second line of the following code segment shows this:

```erlang
...{kernelProcess,heart,{heart,start,[[]]}},
     {kernelProcess,condition_server,{condition_server,start,[[]]}},
     {kernelProcess,error_logger,{error_logger,start_link,[[]]}},
...```

2.4 Implementation and measurements

We fully implemented the prototype described in the previous sections. The implementation can be used as an OTP extension. This extension is based on Erlang/OTP R11B version and the modifications affected the `stdlib`'s (v. 1.14.1) behaviour modules (namely: `gen_server`, `gen_fsm`, `gen_event`, `supervisor_bridge`, `supervisor`). You can download the prototype from the following url:

`http://compalg.inf.elte.hu/projects/startup`

Up to now, we described a parallel and reliable solution of the concurrent startup. Our solution gives a well-defined interface for handling the dependency problems among the concurrent starting modules. Therefore it preserves the reliability. It means that reliability of the concurrent startup is based on the dependency graph description of the users’ programs. In the following we focus on the running time of the start-up.

We lack access to large industrial applications, therefore we created programs for measuring the start-up time in several cases. For simplicity, no dependence conditions were defined, but concurrent supervisor child starting was performed. The measured programs use the modified Erlang/OTP libraries for making fast startup. The tested systems have some `gen_server` and some `supervisor` nodes. The `gen_server` nodes perform time-consuming, resource-intensive computations in their `init` functions. Each measured system has been started both sequentially and concurrently, the time is given
in seconds. Each measurement has been performed five times and the figures show the average measured values. The measurements were executed on an SMP 4 machine with 2 AMD Dual Core Opteron, 2GHz, 16 GB RAM, Linux, Erlang 5.5, OTP R11B.

Three different system topologies were measured, a system with (1) deep process tree, (2) wide process tree, and (3) random process tree. The deep process tree was a 3-regular tree of depth 6, the wide process tree was a 10-regular tree of depth 2, and the random process tree was generated using uniform distribution from the range [1, 5] for the number of children of a node, then truncating the tree at level 5.

We measured the time that is needed for the system start-up as all servers and supervisors were started. The timer started when the erl shell was called and stopped when the last server or supervisor started. For this purpose we created a special application which starts just after all other servers or supervisors, and then immediately performs an illegal statement. Since this node crashes at once, consequently erl terminates. In other words, we measured the time between the starting and crashing of the Erlang shell.

There are several ways to make a system’s process tree concurrent. We tagged the modules which have to start parallel. The speed of the startup depends on the number of the concurrent processes. The deeper the position of the fork point in the tree, the more parallel threads are created (more dummy supervisors). Therefore we show the running times as a function of the number of concurrent threads and as a function of the depth of fork points.

Figure 2.2 shows that the concurrent versions (not surprisingly) are always faster than the sequential ones. In some cases however, the concurrent start-up was two times faster than the sequential one.

Figure 2.3 shows the startup speed as a function of the number of fork points. Since there were 4 processors (2 dual core) in the testbed, it is not surprising that 4-fold parallelism yields the best results. When only 3 parallel processes were started, one processor did not work, and 3 processors performed the whole startup. In case of more than 4 active processes, the processors had to switch between the active processes resulting in a serious overhead. Note however, that the most significant overhead in the measurements comes from the time consuming part of the servers’ init functions.

Figure 2.4 shows how the results depend on the depth of the fork points. We measured a fall back in performance when all nodes in a given level were started parallel.
2.4 Implementation and measurements

![Bar chart showing startup speed of sequential and concurrent versions (in seconds).]

**Figure 2.2:** Startup speed of the sequential and concurrent versions (in seconds). In concurrent case we can put different number of fork points into different places in the process tree. The author created several concurrent cases for each kind of trees. The worst and best case values represent the slowest and the fastest concurrent start-up time in the proper kind of trees.

In this case the system had more concurrent processes in the deeper levels. One can also observe that the version of 4 active process forking is the most resistant to the depth. In this case the only overhead comes from the number of dummy supervisor trees. The measurement suggests that the system should be forked as close to the root as possible.
Figure 2.3: Start-up speed (in seconds) depending on the number of running processes, which can be set by the fork points. The levels show the depth of the fork points.

Figure 2.4: Start-up speed (in seconds) depending on the depth of the fork points.
2.5 Conclusions

In this chapter a solution for the parallel start-up of Erlang systems was presented. We gave a general description of the solution and we measured the start-up time in several cases. The measurements show that the parallel start-up can be much faster than the sequential. On the other hand the solution provides a well-defined mechanism for controlling the dependency relations among processes resulting reliable systems. The main advantages of the solution are:

- precise and concise dependency handling,
- preserving the supervision tree structures,
- the dependency graph is an Erlang module,
- the dependency graph is dynamic
- less than 150 lines modification in the stdlib.

A disadvantage of the solution is that bad dependency graph could result in dead-lock or system crash. We conclude that the solution is highly capable for the parallelization of Erlang systems’ startup in case of legacy systems and new developments as well.
Chapter 3

Cache cleaning

3.1 A traditional solution

In this section we give a concurrent cache implementation and mention some other potential solution. Furthermore special thread that maintains expired elements is described.

Each cache has a database which is stored either on the disk or in the memory. When the data are stored on disk, usually a database system is used for searching. Since the speed of this solution greatly depends on the implementation of the database system, these caches are not examined in the present dissertation. In certain cases, the database is stored in the memory. In this case, some data can be written on the disk, but it seems to be practical to store the key values in the memory for faster search. If the requested record is on the disk, it can be read, and this solution does not make the search slower, but the getting of the proper element. The last solution is the one, where the whole database is in the memory. Subsequently, these two solutions are merged into one where the whole database is in the memory. If the database is in the memory more data structures can be used: search trees, chained or open addressing hash tables. In the thesis the chained hash table is used [34, 35] for data structure and suppose that the whole database is in memory.

A simple way of implementing a concurrent cache is based on threads and read/write lock [6, 36, 37, 38, 39, 40, 41]. Threads can read or write the database and read/write lock is responsible for mutual exclusion.

Read/write lock is a gate for the data structure, threads can access the structure
3.1 A traditional solution

Figure 3.1: A schematic picture about a cache implementation. Threads try to process insert and find instructions on a hash table guarded by a lock. The clearer thread tries to clear the old elements from the same hash table.
by closing this lock in read or in write mode. If a thread closes the lock for reading no other thread can secure this lock for writing but several threads can get read access for the structure in parallel. If a thread tries to close the lock for writing it has to wait the reader threads leaving the critical (mutual excluded) area (open the lock), and after that it can use the structure and during its action no other thread can read or write the cache. The thread implementations use queues for performing this functionality.

It is a special thread that maintains the data as it cleans up or updates expired records. This thread will be called: cleaner thread. This thread periodically reads the database and removes or refreshes the records. During action only this thread can reach the data so all read and write accesses have to wait.

3.2 A lock based concurrent hash table description

An obvious improvement for the linked hash tables is the bucket level locking. (Buckets are the lists of nodes.) This solution was described for database systems by C. Sch. Ellis [17, 18]. In this case each node has a private lock or mutex, furthermore there is a lock (called resize lock) for the “backbone” of the hash table.

In this solution when a thread tries to access the hash table for reading, it calculates the hash value, closes the resize lock and the corresponding bucket’s lock for reading. If the thread found the searched record or it scanned all records in the bucket, it frees the locks in reverse order. In the case of writing this algorithm becomes more complicated. When a writing operation is in progress the thread secures the resize lock for reading and closes the corresponding list’s lock for writing. Consequently the other threads can read from or write in other buckets of the hash table. The unlocking mechanism works as above, in reverse order. If an inserting occurs in a bucket that has too many elements, (or there is any other reason to resize the hash table), an additional step will be executed as follows: after the writing frees its locks, it blocks the whole hash table by closing the resize lock for writing. In this case, no other thread can reach the data structure and the resizer thread can make its actions undisturbed. After that, it unlocks the whole structure.

This solution needs more memory than the simple one, but enables concurrent reading and writing actions or parallel writings in the cache.
3.3 Improvements for concurrent caches

The term primary structure will be used when it refers to the data structure of the cache and secondary structure for the other structures, which are responsible for efficient data updating. In the implementation of the solutions the hashed value of the keys were stored in the records. The corresponding bucket was calculated by a modulation with the size of the backbone of the hash table. This is useful in case of resizing, because the new index can be recalculated with just a modulation during each reinsertion. We used hash tables with a backbone size of a prime number.

3.3.1 Cleaning thread improvement

The main approach of cleaning improvements is the arrangement of data by two values: the key and the expiration time of the data. The idea of multiple indexes is widespread in informatics [35, 42]. In this case, we have simpler data structure, because we do not lookup records simultaneously by both keys. Each thread except the cleaner thread only uses the original key for searching, and the cleaner thread looks up elements in the cache by the expiration time.

We redesigned the hash table and added a new data structure to it. This secondary structure works like an index structure to the records. It simply stores the expiration times and pointers to the original records in the hash table. The first obvious observation is that an arbitrary priority queue implementation could be the base of the secondary data structure. Items should be stored with the expiration time as priority. At each cleaning, each item with lower priority value than the current time should be erased. The new data structures’ find algorithm works the same way as a normal hash table. The secondary structure affects only the insertion and cleaning. This secondary structure can be implemented in many different ways. These will be described in the following sections.

Binary heap

We studied the well-known binary heap as one of secondary structure implementations. If we store $n$ elements in the cache this means $n$ elements in the binary heap. An insertion time complexity has two parts. First the insertion into the hash table. This means $\Theta(1)$ computational time similar to an ordinary hash table. Second the insertion
3.3 Improvements for concurrent caches

of the (Expiry Time, Pointer) pair into the heap. This has $O(\log(n))$ computational time. Consequently the insertion has $\Theta(1) + O(\log(n))$ time complexity. If we use perfect hash function, the computational time will be $O(\log(n))$.

When a cleaning has started the thread examines the heap. If the top element’s expire time is greater than the actual time, it stops because no elements have expired. In this case the computational time is $\Theta(1)$. If expired elements exist, the clearer thread erases the top of the heap and the proper element from the hash table ($\Theta(1)+O(\log(n))$). This instruction will be repeated until no expired elements exist in the cache. This means $ee O(\log(n))$ computational time, where $ee$ is the number of expired elements in the cache, if we have a perfect hash function.

Figure 3.2: A cache with binary heap as secondary structure
3.3 Improvements for concurrent caches

Unbalanced binary tree

This data structure is an unbalanced binary search tree with a pointer to the smallest and the largest element. The proper place to insert a new node is always searched from the largest item in the tree. The search is performed as a walk-down to the first smaller node on the path from the largest to the root, and afterwards a regular binary tree search is done from the node that has been found. It should be noted that if the current item is larger than the previous largest value in the tree, then the search is performed in constant time, while on the other hand the expected value of the insertion time for a random sequence is $O(\log(n))$. The cleaning algorithm works as follows: Find the first tree node containing smaller or equal value than the expire time. The node and its left subtree can be deleted. The right child will be relink to its grandparent node. After that the previous two steps have to be repeated from the right child till the search terminates. This algorithm can be optimized for this special tree if the first search starts to the smallest element and stops if a node’s parent is greater than the expiry time or the node is the root.

In this case, the insertion into the secondary structure has $O(\log(n))$ time complexity, where $n$ is the number of elements presented in the cache. The cleaning time complexity is much more complicated. The sequencing of the expired elements means $O(ee)$ time if they are placed in a subtree ($ee$ is the number of expired elements). For the worst case scenario an upper bound can be given: $O(n)$ for the search and $O(ee)$ for the summarized sequencing of the expired elements. Consequently $O(ee)+O(n) = O(n)$ is a worst case estimation.

Bucket list

If the TTL value is a constant then the binary search tree is a list. In this case each insertion has $O(1)$ computational time, and the erasing of the expired elements has $O(ee)$ time complexity (where $ee$ is the number of expired elements). This observation gives the idea of bucket lists. These lists contain time buckets depending on the cleaning period of the cleaner thread. The time is divided into periods, and each bucket represents a period. When a new element arrives, it will be inserted into the proper bucket containing the elements expiring between $ct_k$ and $ct_{k+1}$, where $ct_i$ is the time of the $i^{th}$ clearing. The expiration time can be calculated as follows: $et = now + TTL$. 

34
3.3 Improvements for concurrent caches

During an insertion, usually the new element has to be inserted into the last bucket, if the TTL values are mostly constant. In this case, the elements that will expire in the next cleaning, are all in one bucket at the beginning of the list. As a result the cleaning thread just easily reads the first bucket's border time, and if it is less than the current time, it updates or removes the referenced elements. Note that we do not need to store the empty buckets.

![Figure 3.3: A cache with bucket list as secondary structure](image)

The running time of this structure depends on the assumption. As you can see in Figure 3.3, the buckets are represented by a simple one directional linked list. In this case, insertion into a bucket is a $\Theta(1)$ time algorithm. If the TTL values are mostly constant, the insertion to it is $\Theta(1)$. In the worst case this action can take $\Theta(tb)$ asymptotic running time, where $tb$ is the number of non-empty time buckets. The cleaning always has $\Theta(ee)$ running cost (with perfect hash function), where $ee$ is the number of expired elements, because the cleaner thread can find the expired elements by walking through the last bucket. This data structure enables the cleaner thread to
3.4 Implementation and results

A lock based hash table and the secondary structures mentioned above were implemented. The secondary structures have an additional lock. The insertion algorithm has to be extended with an insertion into the secondary structure. This additional algorithm locks and unlocks the secondary lock. The secondary lock is just a mutex, consequently one thread can insert into the cache at one time. Since the resizing and the finding do not affect the secondary structure during insertion finding may be performed. Additionally we implemented a traditional solution (called simple on the figures) and a multi lock solution without secondary structure (called multi on the figures). The measurements were performed on several platforms and machines. For testing the performance 3 different architectures were used:

1. Pongrac is an Intel Pentium 4 machine with 1 Gb memory. A FreeBSD works on it.
2. iMac and Macbook have Intel Core Duo 2GHz processor with 1Gb memory. Their operating system is Darwin.
3. Lime has 2 pieces AMD Dual Core Opteron 2.2GHz with 16 Gb memory. It is a Linux machine.

The measurement programs perform numerous find and insert instructions in different distributions. Figure 3.4 shows the overall performance. Pongrac’s results show that on a single processor machine the simple solution is the fastest. The reason is that the kernel hibernates the threads that are waiting for a lock. Consequently the thread switching has a smaller time overhead. In the other three cases the simple solution was the slowest. Unfortunately the speed of the measures strongly depends on the operating system thread and lock implementation as you can see the difference between Linux (Lime) and Darwin (MacBook, iMac).

Figure 3.5 shows the number of transactions depending on the cache size. The simple solution was much slower than the multi lock and multi lock solutions were not sensitive to the number of elements. However there was no significant speed up among the different kind of secondary structures.
Finally figure 3.6 shows the performances depending on the number of threads. The multi lock solutions were insensible for the thread number in contrast of the simple solution. But the different kind of secondary structures do not have significant performance difference.
3.4 Implementation and results

**Figure 3.5:** Transactions vs. Size of Cache (iMac)

**Figure 3.6:** Transactions vs. Number of Threads (iMac)
3.5 Conclusion

During this research three different secondary structures were created for making the cache-cleaning faster and these additional structures did not affect the finding time. These data structures were implemented and measured. The implementation based on lock based data structures.
Chapter 4

Classifications of $\mathbb{Z}^n$ through generalized number systems

4.1 Introduction

Let $M$ be an $n \times n$ regular integer matrix (called the base) and let $D$ be a set of $|\det(M)|$ integer vectors (called digits) of dimension $n$, $0 \in D$. The pair $(M, D)$ is called a generalized number system (GNS) if every integer vector $x$ has a unique finite representation of the form $x = \sum_{i=0}^{k} M^i d_i$ with $d_i \in D$, $d_{j_k} \neq 0$, $k \in \mathbb{N}$. Two important necessary conditions for $(M, D)$ to form a GNS are the expansivity of the matrix $M$ and that the members of $D$ constitute a complete residue system modulo $M$. If these two conditions hold, the pair $(M, D)$ is called a radix system. A large number of sufficient conditions are known, we refer the reader e.g. to [29, 41, 44, 45]. Below we only consider radix systems.

A reformulation of the problem is also possible using the function $\phi : \mathbb{Z}^n \rightarrow \mathbb{Z}^n$, $x \mapsto M^{-1}(x - d)$ for the unique $d \in D$ satisfying $x \equiv d $ (mod $M$). The congruence here means that $x$ and $d$ are in the same coset of the factor group $\mathbb{Z}^n/M\mathbb{Z}^n$. Whenever $(M, D)$ is a radix system, $M^{-1}$ is contractive and $D$ is finite, there exists a norm on $\mathbb{Z}^n$ and a constant $C$ such that the orbit of every $x \in \mathbb{Z}^n$ eventually enters the finite set $S = \{v \in \mathbb{Z}^n \mid \|v\| < C\}$ for the repeated application of $\phi$. Hence, the sequence $x, \phi(x), \phi^2(x), \ldots$ is eventually periodic for all $x \in \mathbb{Z}^n$.

It is then easy to show that the unique representation property holds iff for every $x \in \mathbb{Z}^n$ the orbit of $x$ eventually reaches 0. In such a case, the residue classes of the
iterates of $x$ by $\phi$ correspond to the digits in the unique representation of $x$. A point $x$ is called periodic if $\phi^k(x) = x$ for some $k > 0$. The orbit of a periodic point is called a cycle. Hence, $(M, D)$ is a GNS iff the only periodic point is 0, or equivalently, the only cycle is $0 \rightarrow 0$. In this chapter we investigate the following two problems. The GNS decision problem for $(M, D)$ asks if the pair forms a GNS or not. The more general GNS classification problem means finding all cycles. There are two types of points in the space. A point $x$ is called zero-orbit-point if the orbit of $x$ ends in 0, non-zero-orbit-point otherwise. In the latter case there must be a non-zero periodic point in the orbit of $x$. The orbit points of all non-zero orbit points constitute the witness set.

A similar dynamic behaviour can be observed in case of a well-known problem called Collatz conjecture[46]. The Collatz conjecture is an unsolved conjecture named after Lothar Collatz, who proposed it in 1937. “Mathematics is not yet ready for such problems.” told Pál Erdős about the Collatz conjecture. In 2006 it was proved that a generalization of the Collatz conjecture is undecidable, but this result cannot be applied to the original Collatz problem.

Let $f : \mathbb{N} \rightarrow \mathbb{N}$ be an integer function:

$$f(n) = \begin{cases} \frac{n}{2} & \text{if } n \text{ is even} \\ 3n + 1 & \text{if } n \text{ is odd} \end{cases}$$

The Collatz conjecture is the following:

$$\forall n \in \mathbb{N}\setminus\{0\} \quad \exists k : f^k(n) = 1$$

where $f^k$ means:

$$f^1(n) = f(n)$$
$$f^k(n) = f(f^{k-1}(n)).$$

It is known that the generalization of function $f$ to the complex numbers often leads to fractal structures.

The finiteness of the set $S$ above implies that the GNS decision and GNS classification problems are decidable. However, no general efficient algorithms are known. Most algorithms in the topic are specific to the decision and to the canonical case. Canonical means that $D = \{(j, 0, \ldots, 0) \mid j = 0, \ldots, |\det(M)| - 1\}$. A prominent example is
Brunotte’s algorithm [29], which was generalized to the direction of shift radix systems [47, 48].

In [43] two possible approaches are presented for the general case of the classification problem. One of them (called method $\alpha$ in the article) follows a top-down philosophy: it finds a rectangular region where all cycles are located and then an exhaustive search through this region solves the GNS classification. Method $\alpha$ is an improvement of the algorithm based on a cover of the set of fractions, which appeared in [30]. Method $\beta$ is bottom-up: starting with the digits it grows a set of witnesses that eventually contains all cycles. This method is the adaptation of Brunotte’s algorithm to the GNS setting, also including classification. The experiments show moderate correlation between the running times of the two methods, but neither one outperformed the other in all cases. Unfortunately, even in fairly simple cases (with only two digits, and $n$ being only 12), they reached their limits on a single computer. Method $\alpha$ requires too much computation, while method $\beta$ encountered memory problems.

In section 4.2 we present methods how the computational needs of method $\alpha$ can be (somewhat) relaxed using grid architecture. Grid methods have already been successfully applied to other computation-intensive problems in the field, notably for the search of expansive polynomials [49, 50, 51]. In section 4.3 we will introduce parallel solution for the method $\beta$ both on a general multiprocessor architecture and on a Cell Broadband Engine.

## 4.2 Classification problem on a GRID

### 4.2.1 Algorithms

In this section, two different algorithms for the classification problem are shown. Huge parts of these algorithms are common. First the common parts, than the special parts will be discussed.

#### 4.2.1.1 Common parts of the algorithms

For the classification problem distributed algorithms were developed on a grid architecture. Our algorithms are based on the master-slave computing model. The master divides the enclosed $n$-dimensional rectangle into parts and schedules them as jobs
among the slaves. The slaves get a job and perform the arithmetic computations by searching the periodic points.

Algorithm 1 and Algorithm 2 show a schematic view of the master and slave algorithms, resp.

The master and the slaves communicate with each other via different types of messages which usually contain some data. For example JOB messages contain the description of the parts or WITNESS_FOUND messages include the found witness point. It is important to note that all kinds of messages carry just a few bytes of data.

**Algorithm 1 Schematic Master Algorithm**

1: Create the space division and store parts
2: working_nodes := 0
3: is_number_system := true
4: while exists unchecked parts or working_nodes ≠ 0 do
5:  while exists NODE_READY message in the message queue and
6:  exists unchecked part do
7:    Send back a JOB message to the slave
8:    working_nodes := working_nodes + 1
9:  end while
10: while exists JOB_FINISHED message in the message queue do
11:  Send an ACK message back
12:  working_nodes := working_nodes − 1
13: end while
14: while exists WITNESS_FOUND message in the message queue do
15:  is_number_system := false
16:  Search, check and store the new periodic points
17: end while
18: end while
19: Send TERMINATE message to each slave
20: return is_number_system

The master algorithm divides the space and shares the jobs among the slaves. The working_nodes variable holds the number of slaves working actually on parts. This variable helps to avoid the early finishing of the main loop. When the last parts were sent, the working_nodes variable will be different from zero, consequently the master will process the JOB_FINISHED and WITNESS_FOUND messages of the running slaves.
Implementing the master the space division is actually performed dynamically in successive phases (not just at the beginning of the master algorithm) which means that if the generated parts are run out then the master generates new parts until it is possible. Furthermore, the implemented master algorithm not only decides the number system property but it is able to enumerate the periods from the stored data as well. Finally the schematic algorithm does not specify the treatments of the crashing nodes. These are just implementation details and they do not contribute to the essence of the algorithms.

**Algorithm 2** Schematic Slave Algorithm

1: Find master
2: set_of_witness_points := Ø
3: while true do
4:    Send NODE_READY message to master
5:    Wait for JOB or TERMINATE message
6:    if TERMINATE message arrived then
7:      break
8:    end if
9:  set_of_checked_points := Ø
10: while not all points checked do
11:    Check a point’s orbit
12:    if witness point was found then
13:      Send a WITNESS_FOUND message to master
14:    end if
15: end while
16: Send JOB_FINISHED message to master
17: Wait for ACK message
18: end while

The slave algorithm first finds the master and starts a loop until all given parts are checked (the loop condition is between lines 6–8). When all parts are checked the master sends a TERMINATE message to the slave nodes (**Algorithm 1** line 18). In each iteration every single slave receives a JOB message (a set of points), and checks the orbits of these points. If a witness point was found, a WITNESS_FOUND message is sent to the master with that point.
Algorithm 3 The common orbit checking algorithm

Require: point_to_check : a point required to check
1: set_of_orbits_points = ∅
2: x := point_to_check
3: while x ≠ 0 and
4: x ∉ set_of_orbit_points and
5: x ∉ set_of_previously_checked_points and
6: x ∉ set_of_witness_points and
7: x ∉ set_of_checked_points do
8: set_of_orbit_points = set_of_orbit_points ∪ {x}
9: x = phi(x)
10: end while
11: if x ∈ set_of_orbit_points or x ∈ set_of_witness_points then
12: set_of_witness_points = set_of_witness_points ∪ set_of_orbit_points
13: Save x for sending
14: return WITNESS_FOUND
15: end if
16: set_of_checked_points = set_of_checked_points ∪ set_of_orbit_points
17: return NO_WITNESS_FOUND
In the algorithm the common parts of the orbit checking algorithm is shown. The sets contain the informations as follows:

\[\text{set of orbit points}\]: This set contains the points of an orbit.

\[\text{set of witness points}\]: Stores the witness points. The conditional statements from line 6 to line 10 detect whether a witness point was found or not and merge the new witness points into this set.

\[\text{set of checked points}\]: Contains the points that were reached by this node during points’ checking of the actual job. In practice this set can be very large which explains the necessity of clearing (Algorithm line 9).

\[\text{set of previously checked points}\]: This set represents the points that already have been investigated (or the investigation is in progress) by the whole system. Since the master generates the points in a well-defined order therefore if the checking algorithm (Algorithm 3, Algorithm 11) reaches a point which was enumerated earlier in this order the algorithm stops and starts working with the next point. This set, instead of the previous ones, is not stored in the memory. The membership in this set is checked with (simple) functions. This calculation is usually faster than the searching in an ordinary set. A great advantage of this set realization is that it can store an arbitrary number of points.

The general algorithms do not describe the details of the point enumeration strategy of the master and the usage of the appropriate searching function belonging to the \[\text{set of previously checked points}\]. The next two subsections describe two solutions for the classification problem depending on the chosen enumeration strategy.

### 4.2.1.2 Deadlock-freedom

**Theorem 4.2.1** The communication between the master and slave nodes is deadlock-free.

**Proof** Since the master does not wait for messages just checks if a message has arrived or not the master can’t be blocked. (The master can’t spin in the loop between lines 5 and 8 because there are finite nodes and the the slaves wait for an ACK message from the master after sending a JOB_FINISHED message.)
The slave nodes wait for message in two places at line 5 and 16. Suppose indirectly that at line 16 a slave is blocked. In this case no ACK message has arrived from the master. The slave had previously sent a JOB_FINISHED message to the master, consequently the master had to enter the loop where the JOB_FINISHED messages are processed. In this case the master reaches the sending ACK message line at line 10. This is a contradiction.

If unchecked parts exist the slaves can’t be blocked at line 5 and the proof is similar to the previous one. If no more unchecked parts remain then the master will process the JOB_FINISHED messages and jump out of the main loop. After the main loop it sends TERMINATE messages to each node. The nodes receive the TERMINATE messages at line 6 and jump out of the loop in the next line.

□

We notice that the general description of the solution focuses to the mathematical side of the implementation. The details of the message passing and the communication protocols will not be specified.

4.2.1.3 Brute Force Solution

In the first solution the master uses an algorithm similar to the backtrack search algorithm for enqueuing the points. The created sequence is cut into slices. A slave processes one slice at a time. The slaves use the same algorithm for enqueuing the points of a slice, consequently a start point and a length is enough for describing a slice. ALGORITHMS 4 and 5 show the master’s slice creating mechanism. Points endpoint1 and endpoint2 are the two endpoints of the n-dimensional rectangle which bounds a set containing all the periodic elements (30). We assume that endpoint1 ⪯ endpoint2 and EXTR is an extremal element which represents the end of the sequence and is generated by nextPoint function. The next function is similar to the iterative backtrack algorithm. The function steps back when result[tmp] = endpoint2[tmp] in line 3.

Since slaves apply the same enqueuing algorithms for generating the points of a slice the p ∈ set_of_previously_checked_points is equivalent with p ≺ point_to_check. The decision algorithm of this relation is shown by ALGORITHM 6.

Advantage of this solution is that the enqueuing algorithm is very simple. If the moduli of the eigenvalues are big ( ≫ 1 ) then the iterates of φ converge quickly to the
4.2 Classification problem on a GRID

Algorithm 4 Creating slices

Require: \textit{endpoint1}, \textit{endpoint2} : n-dimensional arrays

Require: \textit{slice\_size} : default size of a slice

1: \texttt{p} := \textit{endpoint1}
2: \textbf{while} \texttt{p} \neq \textit{EXTR} \textbf{do}
3: \texttt{slice\_begin} := \texttt{p}
4: \texttt{slice\_length} := 0
5: \textbf{while} \texttt{slice\_length} \neq \texttt{slice\_size} \textbf{and} \texttt{p} \neq \textit{EXTR} \textbf{do}
6: \texttt{p} := \texttt{nextPoint}(\texttt{p}, \textit{endpoint1}, \textit{endpoint2})
7: \texttt{slice\_length} := \texttt{slice\_length} + 1
8: \textbf{end while}
9: \textbf{end while}
10: \textbf{end while}
11: \textbf{end while}
12: \textbf{end while}

Algorithm 5 \texttt{nextPoint}

Require: \texttt{p}, \textit{endpoint1}, \textit{endpoint2} : n-dimensional arrays

1: \texttt{result} := \texttt{p}
2: \texttt{tmp} := 1
3: \textbf{while} \texttt{tmp} \leq \textit{DIMENSION} \textbf{and} \texttt{result[tmp]} = \textit{endpoint2[tmp]} \textbf{do}
4: \texttt{result[tmp]} := \textit{endpoint1[tmp]}
5: \texttt{tmp} := \texttt{tmp} + 1
6: \textbf{end while}
7: \textbf{if} \texttt{tmp} \leq \textit{DIMENSION} \textbf{then}
8: \texttt{result[tmp]} := \texttt{result[tmp]} + 1
9: \textbf{else}
10: \texttt{result} := \textit{EXTR}
11: \textbf{end if}
12: \textbf{return} \texttt{result}
4.2 Classification problem on a GRID

Algorithm 6 Deciding the $x \prec y$ relation

Require: $x, y : n$-dimensional arrays

1: $tmp := \text{DIMENSION}$
2: while $tmp \geq 1$ and $x[tmp] = y[tmp]$ do
3: $tmp := tmp - 1$
4: end while
5: if $tmp = 0$ or $x[tmp] > y[tmp]$ then
6: return false
7: end if
8: return true;

periodic points, consequently, the main disadvantage of the Brute Force solution is that it enqueues the points near to 0 “randomly”.

4.2.1.4 Smart Force Solution

In dimension 3, the previous solution works similar to a mechanism filling a box with water. The water fills the box in bottom-up direction. The set of previously checked points set is the water on the bottom of the box. Since $\phi$ tends to the periodic points the set of previously checked points usually is far from the orbits of the points being actually checked.

The Smart Force solution is based on small $n$-dimensional rectangles called bricks. Bricks play the role of slices in this solution. The master generates bricks and the points of bricks are checked by slaves. We introduce the notion of prism as follows. A prism is an $n$-dimensional rectangle which is built from bricks. The bricks which have already been checked and are under checking form a prism.

At the beginning, the master creates the first brick which encloses 0. This is the first prism as well. The brick and prism generator algorithm works as follows:

1. Gets the latest prism and chooses a direction (a dimension and a sign ($\pm 1$)).
2. Extends the prism towards to corresponding direction, namely generates bricks which border on the prism in that direction. The extended prism forms a new prism.
3. The algorithm continues with step (1) until the extended prism is equal to the $n$-dimensional rectangle given by endpoint1 and endpoint2.
The sizes of bricks are the same except in the borders. The job messages contain the corresponding brick and the neighbour prism. Both are represented with the two \( n \)-dimensional end-points. Figure 4.1 shows the order of prisms and bricks in the second dimension. Figures 4.2 to 4.8 show the first round of the extension in the third dimension. The gray bricks create a prism. If we check the iteration of a point from a white brick then we can stop if we arrive at the gray area.

**Figure 4.1:** On the picture a rectangle, a line or a point represent a brick in the second dimension. a) the numbers mean the order of bricks. b) the numbers show the new bricks of a prism. For example the bricks numbered 1, 2 and 3 create a prism, and during calculating of bricks with number 3, no more iterations are needed, if the \( \phi \) function arrived in a prism created in bricks numbered 1 and 2.

**ALGORITHMS from 7 to 10** show the details of this solution. **ALGORITHM 7** generates and stores the bricks with the corresponding prisms. Between lines 1 and 7 the first brick is generated. At the beginning no checked points exist consequently the 0\( ^{th} \) prism
4.2 Classification problem on a GRID

Figure 4.3

Figure 4.4

Figure 4.5

Figure 4.6

Figure 4.7
is an empty prism (lines 1, 2). The first brick bounds 0 (lines 3 – 6). The \textit{BrickSize} variable is an \textit{n}-dimensional vector which holds the default size of bricks. Here we assume that it contains odd numbers. In case of even side length the generation of the first brick a bit more complicated but works similar. Furthermore in the algorithms the variable \textit{DIMENSION} is equal to \textit{n} and represents the dimension of the space. The variable \textit{bricks} is a queue which contains the jobs in form of pairs (prism, brick) (line 7). The first (non empty) prism will be the first brick (line 8). Prism and brick variables have the same type. It is a record with two fields \((ep1, ep2)\) which represents the two endpoints of the rectangles.

From lines 9 through 19 the algorithm generates the bricks for the whole space. We mentioned earlier that direction is given by a pair; during the algorithms it is represented by the variables \textit{dimension} and \textit{sign}. The cycle in lines 11 – 19 chooses a direction and extends the prism in the above described way. The function call in line 12 extends the corresponding prism (and changes the value of \textit{P} and \textit{bricks}). The \texttt{if} statement in lines 13 – 18 changes the direction of the next extension.

The \texttt{generateSide} function (see algorithm \ref{alg2}) performs an extension. The algorithm has two separate parts. The first initializes the variables (lines 1 – 20) and the second generates and stores the bricks furthermore it saves the new prism into \textit{P} (line 30).
Algorithm 7 Generating and saving the bricks

Require: $endpoint1, endpoint2$ : endpoints of the bounding rectangle

Require: $\forall i : endpoint1[i] \leq endpoint2[i]$

1: $P.ep1 := 0$
2: $P.ep2 := 0$
3: for $tmp = 1$ to $DIMENSION$ do
4:   $B.ep1[tmp] := \max(-\text{BrickSize}[tmp]/2, endpoint1[tmp])$
5:   $B.ep2[tmp] := \min(\text{BrickSize}[tmp]/2, endpoint2[tmp])$
6: end for
7: bricks.push($\langle P, B \rangle$)
8: $P := B$
9: $dimension := 1$
10: $sign := 1$
11: while $P.ep1 \neq endpoint1$ and $P.ep2 \neq endpoint2$ do
12:   $\text{generateSide}(P, dimension, sign, bricks)$
13:   if $dimension = DIMENSION$ then
14:     $dimension := 1$
15:     $sign := -sign$
16: else
17:   $dimension := dimension + 1$
18: end if
19: end while
During the algorithm additional variables are introduced. The following enumeration describes them:

**BrickStep**: BrickStep is defined for technical reasons, by
\[ \text{BrickStep}[i] = \text{BrickSize}[i] - 1 \]

**newP**: Represents the new prism. This prism is calculated from the original one \((P)\) and contains the newest extended bricks as well.

**t,s**: These two variables define the \(n-1\)-dimensional face of the prism shifted by \(\pm 1\).

**\(\delta, \text{step}\)**: For calculating the bricks’ endpoints the algorithm walks through the \(n-1\)-dimensional shifted face defined by \(s\) and \(t\) with BrickSize steps. If \(p\) is a point which is the endpoint of a brick, then the other endpoint will be: \(p + \text{BrickStep}\) (if \(\text{sign} = 1\) and \(p + \text{BrickStep}\) is in the space). The next brick’s first endpoint will be determined by Algorithm 10.

The initialization of BrickStep vector is performed between line 1 and 3. Lines 4 through 14 initialize the variables newP, \(t, s\). Finally, the next 4 lines set the value of vector \(\delta\). The \(\delta[\text{dimension}] = 1\) instruction helps the traversal algorithm, to skip the steps towards the extending direction (see Algorithm 10).

The generation starts with the first brick. The creation and storage of this brick happens between lines 20 and 22. The cycle from line 23 to 26 steps to the next point, generates and stores bricks. The algorithm sets the new value of \(P\) at the end (line 27).

The generateBrick algorithm (Algorithm 9) checks if the brick fits in the space and cuts the points outside the space. If the direction is \(-1\), it rechecks the previous property and swaps the coordinates of the endpoints in the extension’s direction in order to achieve \(B.ep1 \preceq B.ep2\).

The nextStep function (Algorithm 10) performs the traversal from \(t\) to \(s\). The algorithm returns true if the next point (stored in \(t'\)) is a valid point and returns false otherwise. Since \(\delta[\text{dimension}] = 1\) this function will not walk toward the extension’s direction because when \(\text{tmp} == \text{dimension}\) the loop condition will be true \((\text{tmp} \neq \text{DIMENSION} + 1 \text{ and } 0 \leq 1 - 1)\) consequently the loop jumps over this coordinate.

The \(x \in \text{set_of_previously_checked_points}\) function can be performed as algorithm 11 shows. Simply verify that the corresponding point is in the proper prism or not.
Algorithm 8 \textit{generateSide}

\textbf{Require:} $P$ : The extended prism

\textbf{Require:} $\text{dimension, sign}$ : The direction

\textbf{Require:} $\text{bricks}$ : The queue that stores bricks

1. \textbf{for} $\text{tmp} = 0$ to $\text{tmp} \leq \text{DIMENSION}$ \textbf{do}
2. \hspace{1em} $\text{BrickStep}[\text{tmp}] = \text{BrickSize}[\text{tmp}] - 1$
3. \textbf{end for}

4. \textbf{for} $\text{tmp} = 1$ to $\text{tmp} \leq \text{DIMENSION}$ \textbf{do}
5. \hspace{1em} $\text{t}[\text{tmp}] = \text{newP.ep1}[\text{tmp}] = \min(\text{P.ep1}[\text{tmp}], \text{P.ep2}[\text{tmp}])$
6. \hspace{1em} $\text{s}[\text{tmp}] = \text{newP.ep2}[\text{tmp}] = \max(\text{P.ep1}[\text{tmp}], \text{P.ep2}[\text{tmp}])$
7. \textbf{end for}

8. \textbf{if} $\text{sign} = 1$ \textbf{then}
9. \hspace{1em} $\text{newP.ep2[\text{dimension}]} = \text{newP.ep2[\text{dimension}]} + \text{BrickSize[\text{dimension}]}$
10. \textbf{else}
11. \hspace{1em} $\text{newP.ep1[\text{dimension}]} = \text{newP.ep1[\text{dimension}]} - \text{BrickSize[\text{dimension}]}$
12. \textbf{end if}

13. \textbf{for} $\text{tmp} = 1$ to $\text{DIMENSION}$ \textbf{do}
14. \hspace{1em} $\delta[\text{tmp}] = \lceil(\text{s[\text{tmp}] - t[\text{tmp}] + 1}/\text{BrickSize[\text{tmp}]}) \rceil$
15. \textbf{end for}
16. $\delta[\text{dimension}] = 1$

17. \textbf{step} = 0
18. $t_1 = t$
19. $B = \text{generateBrick}(t_1, \text{BrickStep, dimension, sign})$
20. \textbf{push}(\text{bricks, (P, B)})
21. \textbf{while} nextStep($t_1, \text{step, } \delta, t$) \textbf{do}
22. \hspace{1em} $B = \text{generateBrick}(t, \text{BrickStep, dimension, sign})$
23. \hspace{1em} \textbf{push}(\text{bricks, (P, B)})$
24. \textbf{end while}
25. $P = \text{newP}$
4.2 Classification problem on a GRID

Algorithm 9 generateBrick

Require: \( t_1 \) : Current brick starting point
Require: \( BrickStep \) : Vector of step sizes
Require: \( dimension, sign \) : Direction

1: \( B.ep1 = t_1 \)
2: for \( tmp = 1 \) to \( DIMENSION \) do
3: \( B.ep2[tmp] = \min(t_1[tmp] + BrickStep[tmp], endpoint2[tmp]) \)
4: end for
5: if \( sign = -1 \) then
6: \( B.ep2[dimension] = \max(t_1[dimension] - BrickStep[dimension], \)
7: \( endpoint1[dimension]) \)
8: swap\((B.ep1[dimension], B.ep2[dimension])\)
9: end if
10: return \( B \)

Algorithm 10 nextStep

Require: \( t_1 \) : Current brick starting point
Require: \( step, delta, t \) : Helper variables for face traversal

1: \( tmp = 1 \)
2: while \( tmp \neq DIMENSION + 1 \) and \( step[tmp] \leq \delta[tmp] - 1 \) do
3: \( step[tmp] = 0 \)
4: \( tmp = tmp + 1 \)
5: end while
6: if \( tmp = DIMENSION + 1 \) then
7: return false
8: end if
9: \( step[tmp] = step[tmp] + 1 \)
10: \( t' = t + step \cdot BrickSize \quad //Pointwise product \)
11: return true
Algorithm 11 Checking $x \in set \_ of \_ previously \_ checked \_ points$

Require: $prism$ : A prism
Require: $x$ : An $n$-dimensional point

1: for $tmp = 1$ to $DIMENSION$ do
2: if $prism.endpoint1[tmp] > x[tmp]$ or $x[tmp] > prism.endpoint2[tmp]$ then
3: return false
4: end if
5: end for
6: return true

4.2.2 Measured results

After the description of the two algorithms we give a review of the implementations measurements. Both algorithms were written in C++ for 64 bit AMD processors. Hash tables were used for implementing the set functionality. Both algorithms used the same data structure implementation. The atomic object of the algorithms is the $N$-dimensional point. This data structure was implemented in two different ways. There is a reference counted and a static implementation. The reference counted implementation requires a memory allocation in case of point creation but in case of copying usually no data is transferred. In static case the compiler can calculate the size of the objects in compile time but it has to duplicate the data in case of copying. The static implementation works better (roughly 10% speed-up) consequently our results based on this implementation.

The programs consist of two different parts such as the algorithms described above. The master generates slices and bricks, and collects the calculating time sent back by slaves. The results are measured in seconds and are summarized at the end of the running.

The authors generated some input problems. The problems’ spaces are cut for an easy calculable size (10 computers in grid compute them in about 2 hours). Figure 4.9 shows the speed relation of the two algorithms. Y axis shows the speed-up in case of samples (X axis). As you can see in small dimension ($2 – 3$) the brute force algorithm works faster. In higher dimensions the SmartForce algorithm starts to be faster.
4.2.3 Conclusion

At the ELTE Faculty we were able to use about 70 – 80 computers. With this computational capacity the missing problems could be solved in the thirteenth dimension, and some problems were solved in the fourteenth dimension, consequently a higher grid is required in case of the dimension is bigger or equals to 14.\footnote{The SZTAKI Grid has much more nodes.}

4.3 The Parallel Brunotte algorithm

4.3.1 Introduction

Throughout this chapter the aim is to solve the following decision problem\footnote{\cite{52}}: on input $(M, D)$, decide if the system is a GNS or not. Brunotte’s (sequential) algorithm\footnote{\cite{29}}
applies a function \( \phi \) to locate a nonzero periodic point or to conclude that the system is a GNS. The generalization in [28] locates also all periodic points. Algorithm [12] shows an implementation oriented version of the original algorithm [28].

Consequently the focus of this section is an effective parallel implementations of the Algorithm [12]. As you can see, the original algorithm contains many \( n \)-dimensional vector operations and set operations like insert, find, clear, etc. The parallel algorithms meet two different kinds of concurrent problems. First, they need computationally intensive arithmetic vector operations, second, the set implementations require huge amount of memory and general purpose processors. The algorithms described in this section are basically designed for two platforms. The first platform is a generic symmetric multiprocessing (SMP) architecture without any vector processor extension, the second is the Cell Broadband Engine. The SMP platforms have several general purpose processors in contrast of the Cell Broadband Engine where the processors have Synergistic vector processors.

### 4.3.1.1 Description of the Brunotte algorithm

Algorithm [12] works on sets in the following way: starting from \( S_0 = D \), in each iteration \( i \), the set \( S_i \) is enlarged with some new elements,

\[
S_i \subseteq \bigcup_{s \in S_i, d \in D} \phi^k(s + d) = S_{i+1}.
\]

It is known that the iteration eventually stabilizes, i.e. [29]

\[
S_0 \subseteq S_1 \subseteq \cdots \subseteq S_n = S_{n+1} = \cdots
\]

for some \( n \in \mathbb{N} \). In our programs these sets are implemented as follows: in the \( i^{th} \) iteration let \( E = S_i \), let \( old \Delta E = S_i \setminus S_{i-1} \) and let \( \Delta E = S_{i+1} \setminus S_i \), (where \( S_{-1} = \emptyset \)).

The following table shows the parts of this algorithm.

<table>
<thead>
<tr>
<th>Part name</th>
<th>Functionality</th>
</tr>
</thead>
<tbody>
<tr>
<td>Init</td>
<td>Check unit vectors</td>
</tr>
<tr>
<td>Phi</td>
<td>Iterate ( \phi )</td>
</tr>
<tr>
<td>Clear</td>
<td>Check ( periodicFound )</td>
</tr>
<tr>
<td>Swap</td>
<td>( swap(\text{old}\Delta E, \Delta E) )</td>
</tr>
</tbody>
</table>
4.3 The Parallel Brunotte algorithm

Algorithm 12 Sequential Brunotte Algorithm($M, D$)

Require: $M$: The base; $D$: Set of digits

1: for all $b \in B$ do
2:  if $b$ has no finite expansion then
3:    return false
4:  end if
5: end for
6: $E \leftarrow D$, $old\Delta E \leftarrow D$, $\Delta E \leftarrow \emptyset$
7: $periodicFound \leftarrow$ false
8: while $old\Delta E \neq \emptyset$ do
9:  for all $e \in old\Delta E$, $d \in D$ do
10:   Orbit $\leftarrow \emptyset$, $p \leftarrow \phi(e + d)$
11:   while $p \notin Orbit$ and $p \notin E$ do
12:     Orbit $\leftarrow Orbit \cup \{p\}$
13:     $\Delta E \leftarrow \Delta E \cup \{p\}$
14:     $E \leftarrow E \cup \{p\}$
15:     $p \leftarrow \phi(p)$
16:   end while
17:   if $p \in Orbit$ then
18:     $periodicFound \leftarrow$ true
19:   end if
20: end for
21: if $periodicFound$ is true then
22:  return false
23: end if
24: $old\Delta E \leftarrow \emptyset$
25: $swap(old\Delta E, \Delta E)$
26: end while
27: return true
4.3 The Parallel Brunotte algorithm

In the Init part, the algorithm checks the finiteness of the expansions of $B = \{0, \ldots, 0, \pm 1, 0, \ldots, 0\}$ (all basis unit vectors and their opposites) and initializes the variables (lines 1–7).

The last three parts accomplish the main loop of the algorithm (lines 8 – 26). The Phi part is responsible for the iterations of $\phi$ for all elements of $old\Delta E$ and it enlarges the sets $E$ and $\Delta E$. The algorithm in the Clear part checks whether any periodic points were found and clears $old\Delta E$. Finally, in part Swap the algorithm swaps the two sets $\Delta E$ and $old\Delta E$ with simple pointer operations without any copy.

4.3.2 Universal parallel Brunotte algorithm

The universal parallel algorithm, described in this section, runs on generic SMP platforms. Additionally, it serves as the basis of the second algorithm running on the IBM Cell Broadband Engine architecture. Both algorithms use sequential and parallel hash tables for implementing sets. Insertion into a set is denoted by ← in the sequential case, and ⇐ in parallel case. Similarly, the “member” operation has two forms, $\in$ represents the sequential and $\in = $ the parallel operation.

The parallel Brunotte algorithm uses two types of threads. The first type is responsible for the synchronization, scheduling and initialization of global data. It does not perform function $\phi$ calculations or parallel hash operations. The unique thread of the first type is the manager thread. The other threads are called workers, they do the computations and the set operations.

The threads share the following global variables:

**periodicFound**: This is a simple boolean variable. It is true if a nonzero periodic point has been found.

**isEnd**: It is again a boolean variable and shows whether the size of $\Delta E$ is zero or not.

In the parallel case the size of a data structure is usually stored in a variable and this variable is incremented in case of each insertion and decremented when an erasing occurs. For data consistency atomic instructions may have to be used for this operation, which are relatively slow. In our case the $\Delta E = \emptyset$ equality will be checked in a point of execution where only one thread (manager) can reach the set, consequently, it can check this property with iterators in linear time (of the
Algorithm 13 Manager Thread Algorithm

1: Initialization of sets: \( E, \text{old} \Delta E, \Delta E \)
2: \( \text{periodicFound} \leftarrow \text{false} \)
3: \( \text{isEnd} \leftarrow \text{false} \)
4: \( E \leftarrow D \)
5: \( \text{old} \Delta E \leftarrow D \)
6: \( \Delta E \leftarrow \emptyset \)
7: \( \text{syncEndInit.setCounter}(\text{numberOfWorkers} + 1) \)
8: \( \text{syncStart.setCounter}(\text{numberOfWorkers} + 1) \)
9: Starts the worker threads
10: \( \text{syncEndInit.sync()} \)

11: while not \( \text{periodicFound} \) and not \( \text{isEnd} \) do
12: \( \text{syncClear.setCounter}(\text{numberOfWorkers} + 1) \)
13: \( \text{syncEndClear.setCounter}(\text{numberOfWorkers} + 1) \)
14: \( \text{syncStart.sync()} \)
15: \( \text{syncEndSwap.setCounter}(\text{numberOfWorkers} + 1) \)
16: \( \text{syncClear.sync()} \)
17: if \( \text{periodicFound} \) then
18: \( \text{break} \)
19: end if
20: \( \text{syncEndClear.sync()} \)
21: if \( \Delta E = \emptyset \) then
22: \( \text{isEnd} \leftarrow \text{true} \)
23: end if
24: \( \text{swap}((\text{old} \Delta E, \Delta E)) \)
25: \( \text{syncStart.setCounter}(\text{numberOfWorkers} + 1) \)
26: \( \text{syncEndSwap.sync()} \)
27: end while
28: Wait for worker threads
29: return not \( \text{periodicFound} \)
Algorithm 14 Worker Thread Algorithm

**Require:** \( id \) : The id of the thread. \((0 \leq id < \text{numberOfWorkers})\)

1: for all \( d \in B \) do
2: if \( b \) has no finite expansion then
3: \( \text{periodicFound} \leftarrow \text{true} \)
4: end if
5: end for
6: syncEndInit.\text{sync}()

7: while not periodicFound and not isEnd do
8: syncStart.\text{sync}()
9: for all \( e \) such that is \( id^{th} \) partition of \( old\Delta E \) do
10: for all \( d \in Digits \) do
11: \( Orbit \leftarrow \emptyset \)
12: \( p \leftarrow \phi(e + d) \)
13: while \( p \neq 0 \) and
14: \( p \notin Orbit \) and
15: \( p \notin \Delta E \) and
16: \( p \notin E \) do
17: \( Orbit \leftarrow Orbit \cup \{p\} \)
18: \( E \leftarrow E \cup \{p\} \)
19: \( \Delta E \leftarrow \Delta E \cup \{p\} \)
20: \( p \leftarrow \phi(p) \)
21: end while
22: if \( p \in Orbit \) then
23: \( \text{periodicFound} \leftarrow \text{true} \)
24: end if
25: end for
26: end for

27: syncClear.\text{sync}()
28: if periodicFound then
29: break
30: end if
31: \( old\Delta E \leftarrow \emptyset \)
32: syncEndClear.\text{sync}()
33: syncEndSwap.\text{sync}()
34: end while
hash table’s backbone size). This operation only happens once in each iteration (so it’s a rare event).

\( E, \Delta E, old\Delta E: \) The same sets as in sequential case.

**syncEndInit, syncStart, syncClear, syncEndClear, syncEndSwap:**

These are barrier type synchronization primitives [6, 53]. Each of them is an (encapsulated) integer variable with two functions. The first function is `setCounter` that simply sets the integer variable. The second function is `sync`. This function decreases the variable’s value by one and blocks the caller thread if the new value is not zero. If the new value is zero then all blocked threads will be woken up.

The algorithm consists of five parts. The following table shows the parts and the jobs that are performed by the threads.

<table>
<thead>
<tr>
<th>Part</th>
<th>Manager thread</th>
<th>Worker threads</th>
</tr>
</thead>
<tbody>
<tr>
<td>Init</td>
<td>Initialize the variables</td>
<td>Check digits</td>
</tr>
<tr>
<td></td>
<td>Set <code>syncStart, syncEndInit</code></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Start the workers</td>
<td></td>
</tr>
<tr>
<td>Start</td>
<td>Set <code>syncClear, syncClearEnd</code></td>
<td>Iterate ( \phi )</td>
</tr>
<tr>
<td>Phi</td>
<td>Set <code>syncEndSwap</code></td>
<td>Check <code>periodicFound</code></td>
</tr>
<tr>
<td></td>
<td></td>
<td>Check <code>periodicFound</code> steep</td>
</tr>
<tr>
<td></td>
<td></td>
<td>( old\Delta E \leftarrow \emptyset )</td>
</tr>
<tr>
<td>Clear</td>
<td>( isEnd \leftarrow (\Delta E = \emptyset) )</td>
<td></td>
</tr>
<tr>
<td></td>
<td>( swap(\Delta E, old\Delta E) )</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Set <code>syncStart</code></td>
<td></td>
</tr>
<tr>
<td>Swap</td>
<td>( isEnd \leftarrow (\Delta E = \emptyset) )</td>
<td></td>
</tr>
<tr>
<td></td>
<td>( swap(\Delta E, old\Delta E) )</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Set <code>syncStart</code></td>
<td></td>
</tr>
</tbody>
</table>

Now two threads will be described in the light of the parts. In case of the manager thread the Init part is between line 1 and 10. In the first line the sets are created and the global variables are initialized (lines 2–6). Before the workers start (line 9), the manager sets `syncEndInit` and `syncStart` variables to `numberOfWorkers + 1`. It means that each synchronization blocks all workers and the manager. These variables are set before the starting of workers because all workers check the unit vectors and synchronize with `syncEndInit`.

The Start part is responsible for the initializations that have to be executed in each iteration until either a periodic point is found or \( \Delta E \) becomes empty. This part lies between the synchronization with `syncEndInit` (the first time) or `syncEndSwap` (at the
4.3 The Parallel Brunotte algorithm

end of the loop) and \textit{syncStart}. In this part only the manager thread works and it just sets some synchronization primitives like \textit{syncClear} and \textit{syncEndClear}.

Part Start is followed by part Phi. This part is between lines 15 and 16. Manager here just sets \textit{syncEndSwap}.

The next part is called Clear. Here, the manager thread checks if a periodic point has been found and in that case it jumps out of the loop\(^1\) (lines 17–19), otherwise it is suspended until \(\text{old}\Delta E\) becomes empty (line 20).

After clearing the manager must check the size of set \(\Delta E\) (between lines 21 and 23) and swaps the two sets \(\Delta E\) and \(\text{old}\Delta E\) (at line 24), finally, it sets \textit{syncStart}. This part is called Swap and it takes place between synchronization with \textit{syncEndClear} and \textit{syncEndSwap}.

If the loop is terminating then the manager waits for its workers and returns \textbf{true} as output if the input describes a number system otherwise it returns \textbf{false}.

The worker threads perform the computational parts of the algorithm. In the case of workers, the first part (Init) begins when the manager thread starts them. Until that, the \textit{periodicFound}, \textit{isEnd}, \textit{syncEndInit} variables are set. The workers check the unit vectors (lines 1–5) and wait for all the others (line 6). Note that the \textit{syncEndInit} synchronization step is obligatory since otherwise in case of reaching line 11 the manager thread may not know that the unit vector set contains a periodic element and this can produce a deadlock.

In the Start part the workers just check the loop conditions and wait for the manager thread (lines 7–8).

In the next part (Phi) the manager thread is sleeping but the workers perform the \(\phi\) iterations.

The Clear part starts at line 28 and ends at line 32. During this part two important steps occur. The workers (and the manager) jump out of the loop and terminate if any periodic element has been found (lines 28–30). Otherwise they clear parallel the \(\text{old}\Delta E\) set.

Finally, in part Swap, the workers do nothing.

\(^1\)The workers do the same.
4.3 The Parallel Brunotte algorithm

4.3.2.1 Correctness

In this section we prove that the sequential and generic SMP algorithms are equivalent.

Theorem 4.3.1 The sequential and the parallel algorithms are equivalent.

Proof The two algorithms are equivalent if the $E$ and $old\Delta E$ sets are equal at the beginning of each iteration (the beginning of the Phi part). We introduce the following notation: let $E^s$ and $E^p$ denote the set $E$ in the sequential and parallel cases, respectively. Similarly we will use the following notations: $old\Delta E^s$, $old\Delta E^p$, $\Delta E^s$, $\Delta E^p$.

Before the first execution of part Phi, equality trivially holds by the initializations. Suppose that the equality holds until the $(i-1)^{th}$ iteration. The value of sets $E$, $old\Delta E$ are calculated from the $\Delta E$ sets as follows: $E \leftarrow E \cup \Delta E$ and $old\Delta E \leftarrow \Delta E$. It is sufficient to show that the $\Delta E$ sets are equal at the end of the $(i-1)^{th}$ iteration.

Let $q \in E^s$ be an arbitrary point that was calculated in the $(i-1)^{th}$ iteration. This is equivalent to the formula below:

$$\exists prec_q \in old\Delta E^s \land k \in \mathbb{N} \land d \in D : \phi^k(prec_q + d) = q.$$ 

By induction, $old\Delta E^p = old\Delta E^s$, consequently it is also equivalent to

$$\exists prec_q \in old\Delta E^p \land k \in \mathbb{N} \land d \in D : \phi^k(prec_q + d) = q,$$

which is the same as $q \in \Delta E^p$.

4.3.2.2 Deadlock-freedom

The parallel set instructions are wait-free solutions, consequently a deadlock can occur only during the synchronization of the manager and workers. At the Init part both kinds of threads perform non-blocking instructions except for the synchronization with $syncEndInit$. One or more threads could block at this point if the barrier is not initialized or it is initialized with a bad value. The barriers are initialized at line 7 in the manager’s algorithm. This happens before the threads are created, consequently the barrier is initialized before of the corresponding call of its $sync$ call. Its initial value is $numberOfWorkers + 1$, consequently all threads can pass through this barrier.
The algorithm can run in deadlock only at the points of the algorithm where the threads synchronize with barriers. The system hangs at a barrier if one of the following assertion holds:

- A barrier’s `sync` function is called before its `setCounter` function.
- More threads try to synchronize at a barrier than its initial value (set by `setCounter`).
- Fewer threads try to synchronize at a barrier than its initial value (set by `setCounter`).

We prove the impossibility of the above assertions in the 3 lemmas below.

For the proofs we use the event ordering notation of Leslie Lamport [54]. Let $i$ be a thread and $A$, $B$ barriers. In this case $A^i.sync \rightarrow B^i.sync$ means that the $i^{th}$ thread called barrier $A$’s `sync` function before it called barrier $B$’s `sync` function.

**Lemma 4.3.2** We have $n$ pieces of threads indexed from 1 to $n$. Furthermore let $A$ and $B$ two barriers, and $k$ be a dedicated thread with manager behaviour. If

\[ \forall i \in [1..n] \land \forall j \in [1..n] : A^i.sync \rightarrow B^j.sync, \quad \text{and} \quad B^k.setCounter \rightarrow A^k.sync \]

then

\[ \forall l \in [1..n] : B^k.setCounter \rightarrow B^l.sync. \]

**Proof** Indirectly, suppose that $\exists l : B^l.sync \rightarrow B^k.setCounter$. But the conditions suggest $\forall i \in [1..n] \land \forall j \in [1..n] : A^i.sync \rightarrow B^j.sync$ consequently $A^k.setCounter \rightarrow B^l.sync \rightarrow B^k.setCounter \rightarrow A^k.sync$. That is a contradiction.

**Lemma 4.3.3** There is no barrier whose `sync` function is called by more threads than its initial value.

**Proof** This is trivial from the fact that each barrier is initialized to `numberOfWorkers + 1`. 

---

4.3 The Parallel Brunotte algorithm
Lemma 4.3.4 There is no barrier whose \texttt{sync} function will be called by fewer threads than its initial value.

\textbf{Proof} A thread can terminate at three places. When it tries to enter the main (outermost) loop at first time; when it reaches the Clear part and checks the \textit{periodicFound} variable; and finally when it tries to reenter the main loop.

When the threads try to enter the main loop for first time, the \textit{isEnd} variable is \texttt{false}, consequently the entering depends on the \textit{periodicFound} variable. Before the condition checking each thread waits for the others at the \texttt{syncEndInit.\texttt{sync}} call and the \textit{periodicFound} variable won’t change between the synchronization and condition checking. Hence, all threads enter the loop or jump to the end of the loop.

If a thread has found a periodic point, it sets \textit{periodicFound} to \texttt{true}. At that part Clear all threads checks if a periodic point was found then jumps out of the loop. Before this checking a barrier synchronizes the threads and \textit{periodicFound} variable does not change between the barrier’s \texttt{sync} and the checking. So like the previous case, all threads jump out from the loop or all threads continue it.

Finally, when the threads reenter the loop, the \textit{periodicFound} variable is irrelevant because it does not change after the part Phi. The \textit{isEnd} variable gets its value at the beginning of the part Swap. The threads will wait for the others at the \texttt{syncEndSwap.\texttt{sync}} line. Between this line and the loop condition checking the value of \textit{isEnd} does not change, consequently the threads are able to enter into the loop always together.

\textbf{Theorem 4.3.5} The algorithm is deadlock-free.

\textbf{Proof} This proof follows from the previous three lemmas. Check conditions of lemma \[4.3.2\] for all reasonable pairs \(A, B\).
4.3 The Parallel Brunotte algorithm

4.3.3 IBM Cell Broadband Engine optimized version

In this subsection a special version of the Brunotte algorithm is described that works with 2-element digit sets (though it could be generalized). This version is optimized to IBM Cell Broadband Engine (IBM Cell B. E.) and its realization based on the generic SMP implementation.

The Cell Broadband Engine \cite{cell} is a single-chip multiprocessor with nine processors operating on a shared, coherent memory. Furthermore all processors are connected to each other and to external devices by a high-bandwidth, memory-coherent bus. The platform consists of a PowerPC Processor Element and eight Synergistic Processor Elements interconnected with Element Interconnect Bus that is a high bandwidth communication connection.

The PowerPC Processor Element (PPE) is the main processor. It contains a 64-bit, dual-threaded PowerPC Architecture reduced instruction set computer (RISC) core with a traditional virtual-memory subsystem. It runs an operating system, manages system resources, and is intended primarily for control processing, including the allocation and management of SPE threads. It can run legacy PowerPC Architecture software and performs well executing system-control code.

Synergistic Processor Elements (SPEs) are SIMD processors optimized for data-rich operations allocated to them by the PPE. Each of these identical elements contains a RISC core, 256-KB, software-controlled local store for instructions and data, and a large (128-bit, 128-entry) unified register file. The SPEs support a special SIMD instruction set, and they rely on asynchronous DMA transfers to move data and instructions between main storage (the effective-address space that includes main memory) and their local stores. SPE DMA transfers access main storage using PowerPC effective addresses. As on the PPE, address translation is governed by PowerPC Architecture segment and page tables. The SPEs are not intended to run an operating system.

The PPE and SPEs communicate coherently with each other and with the main storage and I/O through the Element Interconnect Bus (EIB). The EIB is a 4-ring structure (two clockwise and two counterclockwise) for data, and a tree structure for commands. The EIB’s internal bandwidth is 96 bytes per cycle, and it can support more than 100 outstanding DMA memory requests between main storage and the SPEs.
4.3 The Parallel Brunotte algorithm

The Cell Brunotte algorithm is based on the generic parallel algorithm but the $\phi$ iterations are performed by SPEs. The main idea of this algorithm is that the generic worker threads only insert the points into the sets furthermore send and receive the points to the SPEs. Each worker thread has an SPE thread pair that calculates the $\phi$ function. Because the DMA request latency is quite long, consequently several points’ iteration are executed at the same time. Furthermore if the structure of the program changes, a number of new threads are needed to start and run the SPE’s processes.

Algorithm 15 describes the starting of an SPE thread. These threads run on the PPE and only load the SPE object codes into the SPE local memory and start the synergistic processors. When the PPE thread starts the SPE thread it will be blocked until the SPE thread terminates. There are three global variables: $initialData$, $inputData$, $outputData$. The $initialData$ is a compound variable (or struct) and holds the numerical data of the computation such as $M^{-1}$, digits and so on\(^1\). These data are loaded at the beginning of the algorithm and do not change after this point. Both $inputData$ and $outputData$ are arrays (of arrays). Each element of these two variables is an array and holds $SPU\_Data$ pieces of $N$-dimensional points. Each SPE thread has three parameters: the addresses of three global variables (the last two are indexed with $id$). The last line of the algorithm just clears the memory related to the SPE thread.

**Algorithm 15 SPE Starters**

**Require:** $id$ : Identification of the thread. ($0 \leq id < numberOfWorkers$)

**Require:** $SPEid$ : Identification of the SPE thread. ($0 \leq id < numberOfWorkers$)

1. Load SPE object code
2. Start SPE thread(“SPE thread”, $SPEid$, {$initialData$, $inputData[id]$, $outputData[id]$} )
3. Unload SPE object code

Algorithm 16 shows the SPE thread. At line 1 it loads the data needed for $\phi$ calculation from the main memory to its local memory. The $getMessage()$ function is waiting for a message from the PPE. When an SPE thread executes a $getMessage()$ function it will be blocked until a message arrives from PPE. The contents of the message is stored in the variable $message$. The loop between lines 3 and 11 is the main loop. In

\(^1\)Preserving the numerical stability the adjoint of $M$ and $det(M)$ are used instead of $M^{-1}$ during the calculations.
its body the thread performs \textit{SPU\_DATA} number of $\phi$ iterations with attached DMA actions. First the input is read from the main memory (line 4) then the $\phi$ iterations are performed (lines 5 through 7) and finally the results are sent back from the local memory to the main memory (line 8). At line 9, the thread sends a signal to the worker thread (PPE) that the SPE side of \textit{putDMA} command is finished and then at line 10 it is waiting for the next message. The value of the message is of no interest unless it is \textit{TERMINATE}. In this case the SPE thread terminates and puts back the execution to its SPE starter thread.

\begin{algorithm} 
\caption{SPE's Thread} 
\begin{algorithmic}[1]
\State \textit{getDMA}(&\textit{initialData}, &\textit{localInitialData})
\State message $\leftarrow$ \textit{getMessage}()
\While {message \neq \textit{TERMINATE}}
\State \textit{getDMA}(&\textit{inputData}, &\textit{localInputData})
\For {all $i = 0..\text{sizeOf} (\textit{localInputData})$}
\State $\textit{localOutputData}[i] \leftarrow \phi (\textit{localInputData}[i])$
\EndFor
\State \textit{putDMA}(&\textit{localOutputData}, &\textit{outputData})
\State \textit{sendMessage} (\textit{SPE\_READY})
\State message $\leftarrow$ \textit{getMessage}()
\EndWhile
\end{algorithmic}
\end{algorithm}

Algorithm 17 shows how the worker threads work. The major part of the algorithm does not change. First the algorithm checks the basis unit vectors (from line 1 to 5). After the checking the points of old$\Delta E$ (lines from 46) the algorithm makes the same functionality as in general case except in line 56 where a \textit{TERMINATE} message is sent to the SPE thread.

Now let us see the lines from 9 through 45. Because the $\phi$ function is performed for several points parallel, each point must have an orbit set. The orbit sets are stored in the \textit{orbits} array so \textit{orbits}[i] refers to a set with the same functionality as the set \textit{Orbit} does in the general algorithm. We call \textit{chain} the partial orbit of a point from the original point to 0 or to a periodic point or to an already checked point. The \textit{inputData}[id] holds \textit{SPU\_DATA} number of points. Each position represents a chain. The SPE thread does not know if these points are valid or not, so it performs the $\phi$ iteration for all points. But the worker thread must know which points are relevant.
Algorithm 17 Worker Threads’ Algorithm (Cell-optimized)

Require: \( id \): The id of the thread. ( \( 0 \leq id < \text{numberOfWorkers} \) )

1: for \( \forall b \in B \) do
2: if \( b \) has no finite expansion then
3: \( \text{periodicFound} \leftarrow \text{true} \)
4: end if
5: end for
6: syncEndInit.sync()
7: while not \( \text{periodicFound} \) and not isEnd do
8: syncStart.sync()
9: for all \( d \in D \) do
10: \( \text{numberOfPoints} \leftarrow 0 \)
11: \( e\text{Iterator} \leftarrow \text{old}\Delta E.\text{begin}() \)
12: \( e\text{Iterator}.\text{next}(id) \)
13: \( \text{endIterator} \leftarrow \text{old}\Delta E.\text{end}() \)
14: for all \( \text{tmp} = 0..\text{SPU}_\text{DATA} \) and \( e\text{Iterator} \neq \text{endIterator} \) do
15: \( \text{inputData}[id][\text{tmp}] \leftarrow (\ast e\text{Iterator} + d) \)
16: \( \text{numberOfPoints} \leftarrow \text{numberOfPoints} + 1 \)
17: \( \text{orbits}[\text{tmp}].\text{clear}() \)
18: \( e\text{Iterator}.\text{next}(	ext{numberOfWorkers}) \)
19: end for
20: endedChains.clear()
21: while endedChains.size() \neq \text{numberOfPoints} do
22: sendMessage(\( SPUid, PPU_\text{READY} \))
23: message \leftarrow \text{getMessage}()
24: for \( \text{tmp} = 0..\text{numberOfPoints} \) do
25: if \( \text{output}[id][\text{tmp}] \in \text{orbits}[\text{tmp}] \) then
26: \( \text{periodicFound} \leftarrow \text{true} \)
27: end if
28: if \( \text{outputData}[id][\text{tmp}] \neq 0 \) and
29: \( \text{outputData}[id][\text{tmp}] \notin \text{orbits}[\text{tmp}] \) and
30: \( \text{outputData}[id][\text{tmp}] \notin \Delta E \) and
31: \( \text{outputData}[id][\text{tmp}] \in E \) then
32: \( \text{orbits}[\text{tmp}] \leftarrow \text{orbits}[\text{tmp}] \cup \{\text{outputData}[id][\text{tmp}]\} \)
33: \( E \leftarrow E \cup \{\text{outputData}[id][\text{tmp}]\} \)
34: \( \Delta E \leftarrow \Delta E \cup \{\text{outputData}[id][\text{tmp}]\} \)
35: else

72
Algorithm 18 Continued...

```plaintext
36: if eIterator ≠ endIterator then
37:     outputData[id][tmp] ← eIterator + d
38:     orbits[tmp] ← ∅
39:     eIterator.next(numberOfWorkers)
40:     else
41:         endedChains ← endedChains ∪ {tmp}
42:     end if
43: end if
44: end for
45: memcopy(inputData[id],outputData[id],SPU_DATA)
46: end while
47: end for
48: syncClear.sync
49: if periodicFound then
50:     break
51: end if
52: oldΔE ← ∅
53: syncEndClear.sync
54: syncEndSwap.sync
55: end while
56: sendMessage(TERMINATE)
```

For this purpose two additional variables are defined. The `numberOfPoints` variable is an integer number. If no `SPU_DATA` amount of points exist (because the algorithm draws to an end or there are too few points in the `oldΔE` set) this variable stores the amount of chains that are valid. The other variable is `endedChains` that is a set, which holds the array index of irrelevant chains. During the algorithm description iterators are used. An iterator enumerates the elements of a container (here set). The containers have two instructions for helping enumerating themselves. The `begin` instruction creates an iterator and it points to the first element of the container, the `end` instruction makes an iterator object that does not point to any element but an extra element which is logically placed after the last element of the container. Iterators of this article have two functions as follows: `*iteratorName` represents the N-dimensional point the iterator points to, and the `next(i)` instruction steps i times to the next element in the set.
4.3 The Parallel Brunotte algorithm

The algorithm gets all digits and all elements of the set $old\Delta E$ and makes the $\phi$ iteration on them. Between lines 10 and 20 the initial chains are set. From line 10 to 13 the iterators are set to the beginning and the end of the set and the variable $numberOfPoints$ is initialized to 0. The iterator steps as many as many threads there exist at a time. With this mechanism the set is partitioned among the threads and each point will be checked exactly once. At line 12 the iterator goes to the first element of the thread’s part of the set. The loop between lines 14 and 19 stores $point + digit$ values into $inputData$, clears the appropriate orbit sets and counts the maximum size of input and stores it in $numberOfPoints$.

The loop from line 21 through 46 runs until a relevant chain exists. First, the algorithm sends a message to the proper SPE and then it is waiting for the end of $\phi$ iterations (at line 23). When the new points arrived, the algorithm checks them in sequential order (lines between 24 and 44). First the periodicity is checked (lines 25–27). Between lines 28 and 34 the new points are handled, that is, inserted into the sets. If the current point is a periodic point or is already in the sets ($\Delta E$, $E$) and there is an unprocessed element in the set $old\Delta E$ then its chain will be dropped and a new chain is started (lines 36–39). If there are no more chains, the position will be stored in the $endedChains$ set (at line 41). Finally, the whole output of the iteration is copied to the input of the next iteration in line 45. This step is the reason for line 37.

The manager thread of the Cell-optimized version differs only slightly from the original. Before the worker threads start, the SPE starter threads have to be started, and at the end of the algorithm it has to join to SPE starter threads also.

4.3.3.1 Deadlock-freedom

Because the Cell-optimized version is based on the SMP solution, it is enough to prove that the communication between SPE threads and workers is deadlock-free.

Theorem 4.3.6 The communication between a worker and its SPE thread is deadlock-free.

Proof We argue by induction on the number of loop execution. At the beginning before entering the loop the SPE thread has a receive instruction, consequently it is blocked before its main loop. When the worker enters its loop (independently for the inner
4.3 The Parallel Brunotte Algorithm

loops) it first performs a message sending. Consequently the SPE thread can enter the loop and perform the $\phi$ iteration and send back a message. During these actions the worker is waiting for a message. Finally, the worker receives $SPU\_READY$ message and finalizes its loop and the SPE thread is blocked by the message receiving at the end of the loop. Therefore, the first execution of the loops is deadlock-free except the SPE last receiving. Suppose that this state is held in the $n - 1$ execution of the loops. In this case two different events may happen. First there is an $n^{th}$ loop execution. In this case the worker sends a message to the SPE that starts its $n^{th}$ loop iteration and sends back a message to the waiting worker and receives the next message. The worker is able to finalize the loop again. In the other case, when the worker does not start a new loop iteration, it terminates. The last instruction before terminating (at line 67) is sending a TERMINATE message to the SPE. The SPE is receiving this message and jumps out from the loop and ends.

4.3.4 Implementation

The parallel Brunotte algorithms have two kinds of concurrent objects: the sets and the barriers. The sets are much more frequently used than the barriers. The two most popular realizations of sets are trees and hash tables [35]. We implemented the sequential Brunotte algorithm with both. We found that the hash table based implementation runs faster (with a simple hash function $\sum_{i=1}^{DIM} a_i13^i$, where $DIM$ is the dimension of the problem, and $a$ is a $DIM$ dimensional point). The most natural concurrent realization of tree sets are skip lists [6, 56], but based on our sequential experiences, only the hash table based implementation was used.

4.3.4.1 Concurrent hash sets

There are several parallel hash table implementation [6, 14, 18, 23, 24, 25, 57], but we used the hash table implementation of M. Michael with its memory reclamation scheme [21, 22].

The parallel Brunotte algorithm requires some other functionalities that are not described in the original papers. These operations are as follows: clear (denoted by $\leftarrow \emptyset$), and a special iterator which can walk through every $n^{th}$ element of a set (called
4.3 The Parallel Brunotte algorithm

The clear function exploits the simple fact that during a clearing no other set operation occurs. The threads walk through separate lists of the hash table and logically or physically destroy the nodes, consequently this operation does not require any mutual exclusion. The iterator is also based on the previous property. When we iterate through the oldΔE set, none of the threads perform erase, insert or clear operations on that set, consequently the iterator works similarly to the sequential case.

4.3.4.2 Barriers

The other parallel objects in the algorithms are barriers \[6, 53\]. In most of the publications on this objects the authors try to reduce the memory contention of the barrier objects. In our case the functions of the barriers are called very rarely, consequently a simple CAS-based barrier implementation with an additional \textit{setCounter} function was sufficient.

4.3.4.3 Arithmetic vector operations

The arithmetic operations are performed by the general purpose processors in case of the SMP version of the algorithm without using the MMX, SSE, Vector/SIMD Multimedia Extension or other SIMD extension of processors. The Cell-optimized version is based on the synergistic processors and makes a huge part of the arithmetic operations on the SPUs. The Cell-optimized version SPU code is implemented in C++ and assembly. The communication is implemented in C++ and the \(\phi\) function is implemented in assembly (called by the C++ code). In the Cell-optimized version the vector size is extended to a higher dimension for alignment with the 128-bit wide vector registers.

4.3.5 Results

We used two platforms experimenting with the algorithms. The first is an IBM BladeCenter LS22 with two AMD Quad-Core Opteron Processors Model 2356 (2.3GHz 2MB L2) and 4GB of ram. The second is an IBM BladeCenter QS22 with two IBM PowerXCell 8i processors (3.2GHz 512KB cache for each) with 16GB memory. The input data of the measurements was a 12 dimensional problem, this was a natural choice because of the proper (not too short or not too long) running time. The algorithms were
measured with the standard unix \texttt{time} command, and the measurements were repeated three times. The following figures show the average values.

Figure 4.10 shows the results, measured on the Dual-Opteron LS22. I implemented several versions of the Brunotte algorithm. The sequential algorithm is based on the original algorithm described in section 2. In the generic SMP algorithm the insertion into $E$ happens together with the insertion into $\Delta E$. In the original algorithm these insertions take place in different locations. First the algorithm inserts the elements into $\Delta E$ and after the computation of $\Delta E$ it performs a set union ($E = E \cup \Delta E$). This union is called part Union on the figures.

The figure shows that the sequential algorithm is faster in case of a single processor. Several reasons can explain this. First, the parallel hash table can’t resize its backbone, consequently the list could be longer. Second, the parallel algorithms use \texttt{CAS} instructions that are much slower than a usual instruction [6, 58]. Third, the parallel algorithm makes more steps. In both parallel cases the algorithms are the fastest if 7
The most likely explanation is that when we reach the number of cores, the scheduling of the system threads and processes causes more overhead than what is gained with the new CPU core [38].

Figure 4.11 shows the execution time in case of all SMP cases. The PowerXCell results are much slower than the Opteron’s result. The PowerXCell processors have smaller cache, thus the set operations cause frequent cache misses.

Finally Figure 4.12 visualizes results of the Cell-optimized version. During the algorithm the points are sent to the SPEs with DMA. An SPE can transfer data up to 16Kb in DMA requests. The algorithm described above uses only one buffer. The PPE saves points into the buffer, the SPE makes the $\phi$ iterations, and the PPE waits for the end of the iterations. It is obvious that this can be performed with two buffers and while the SPE calculates one buffer, the PPE can insert points into the sets and into the other buffer. We implemented both type of algorithms. Since running times are only slightly influenced by the number of points in a DMA and by the number of
buffers, it is plausible that the Brunotte algorithm’s bottleneck are the set instructions rather than the arithmetic computations. Probably this is also the explanation for the better performance of Opteron CPUs.

![Graph showing execution time vs. number of points sent](image)

**Figure 4.12:** Measured running time depending on the number of points sent

### 4.3.6 Conclusion

We considered a mathematical problem and showed a sequential and several parallel algorithms for it. The algorithms were implemented for two platforms and the results were analyzed. The parallel algorithms meet two different kinds of concurrent problems. First, they need computationally intensive arithmetic vector operations, second, the set implementations require huge amount of memory and general purpose processors.

The measurements show that the Cell B. E. is very effective in case of arithmetic computation but it is not well suited for the memory-intensive set operations.
Summary

Nowadays multiprocessor computers become more and more conventional. Computers with hundreds or thousands of processors are common in the industry, and probably your desktop PC at home has more than one processor as well. On the other hand, there are several problems regarding the sequential solutions, which often do not satisfy the needed performance issues. One of the main challenges in the future’s computer science is developing efficient and safe parallel algorithms instead of the sequential ones. The methodologies, practices and algorithms that are never-falling in the sequential world usually waste the power of the parallel architectures. Well-known examples are reference counting and garbage collecting.

In this thesis (a) I present solutions for some problems that had only sequential solutions earlier and (2) I optimized some existing parallel solutions.

1. A concurrent start-up method that enables us to perform the start-up of an Erlang systems is created:

   - **fast**: with parallel starting of nodes
   - **safe**: with dependency graph description of nodes’ starting order, and
   - **easy to use**: it is based on the standard stdlib codes, furthermore, in case of legacy systems, minor changes are enough to switch from sequential start-up to parallel.

2. In case of linked hash table based caches the speed of the cleaning thread was increased by three additional datastructures. In case of constant TTL value the third data structure has only $\Theta(1)$ additional running time, but the cleaner thread checks only $\Theta(ee)$ number of nodes, where $ee$ is the number of the expired elements. Unfortunately our measurements highly depended on the underlying operating systems.
3. Let $M$ be an $n \times n$ regular integer matrix (called the base) and let $D$ be a set of $|\text{det}(M)|$ integer vectors (called digits) of dimension $n$, $0 \in D$. The pair $(M, D)$ is called a generalized number system (GNS) if every integer vector $x$ has a unique finite representation of the form $x = \sum_{i=0}^{k} M^i d_{ji}$ with $d_{ji} \in D$, $d_{jk} \neq 0$. The GNS decision problem for $(M, D)$ asks if they form a GNS or not. The more general GNS classification problem means finding all the periodic expansions.

(a) Two grid based solutions were created for both the decision and the classification problems. The two algorithms divide the $n$ dimensional space in different ways. We found that in higher dimensions the one outperforms the other. Furthermore, some unsolved 13 and 14-dimensional problems were computed on the ELTE grid.

(b) A parallel version for the sequential Brunotte algorithm was created. This parallel version was implemented in two different architectures. The Brunotte algorithm requires computational power and fast handling of sets. The two architectures differ in their ability in computational power and set handling properties. The SMP platform has less computational power but performs faster finding and inserting instructions on sets, contrary to the Cell processor, where the set operations are slow but it has huge computational power. The result shows that the parallel Brunotte algorithm works better on SMP platforms.
Összefoglalás

A többprocesszoros számítógépek és mobil eszközök korát éljük. Az iparban teljesen megszokott a több száz vagy ezer processzormagot tartalmazó hardver, de valószínűleg sokak otthoni számítógépében is legalább két processzor található. Szoftver oldalról közelítve azonban számos problémának csak szekvenciális algoritmikus megoldása ismert, amelyek alig, vagy egyáltalán nem tudják kiaknázni a párhuzamos környezet kinálta lehetőségeket. Talán a számítástudomány egyik legnagyobb kihívása, hogy biztonságos és jól skálázható párhuzamos algoritmusokat és adatszerkezeteket alkosson.

A disszertációm három olyan probléma elemzését és megoldását tartalmazza, amelynek előtte vagy csak szekvenciális megoldása volt ismert, vagy pedig az ismert párhuzamos megoldás valamilyen szempontból javítható volt.

1. Az Erlang rendszerhez olyan párhuzamos indítási metódust készítettünk, amely:

- **gyors:** mert párhuzamosan tudja indítani a supervisor fák csúcsait
- **biztonságos:** mert a csúcsok egymás közötti függőségeit egy gráffal le lehet írni, és
- **könnyen használható:** mert a megoldás visszafelé kompatibilis a régi szekvenciális indítással, illetve csak kevés változtatást igényelt a **stdlib** kódjában.

2. Lángolt hash-táblákkal implementált gyorsítótárak tisztátáblájának gyorsítását másodlagos adatszerkezetekkel oldottuk meg. Konstans TTL érték mellett a harmadik adatszerkezetnek csak $\Theta(1)$ plusz futásideje volt, viszont a lejárt elemeket a tisztítószával $\Theta(\varepsilon)$ időben tudja felsorolni, ahol $\varepsilon$ a lejárt elemek száma. Sajnos a méréseink nagymértékben függtek az operációs renszertől.

3. Legyen $M \in \mathbb{Z}^{n \times n}$ mátrix a bázis és $0 \in D \subseteq \mathbb{Z}^n$ egy véges halmaz, amit a jegyek halmazának hívunk. Az $(M,D)$ párt általánosított számrendszernek hívjuk, ha minden $x \in \mathbb{Z}^n$ pontra létezik egy $k \in \mathbb{N}$ egész szám, amire $x = \sum_{i=0}^{k} M^i d_i$,
4.3 The Parallel Brunotte algorithm

ahol $d_{ij} \in D$. Két egymástól egy kicsit eltérő problémát tudunk megfogalmazni az általánosított számrendszerekkel kapcsolatban. Eldöntési problémának híve azt, amikor el kell dönteni egy $(M, D)$ párról, hogy általánosított számrendszer-e, osztályozási problémának hívejük azt a feladatot, amikor egy adott $(M, D)$ rendszerben minden periodikus elemet meg kell keresni.


(b) Elkészítettük a Brunotte algoritmus párhuzamos változatát. A párhuzamos verzió két különböző platformra készült el, mivel az algoritmus egyrészről komoly számítási teljesítményt igényel, másrészről pedig igen intenzíven használ halmazokat. Az SMP platformra megírt változat gyorsabban tudta kezelni a halmazokat, míg az IBM Cell processzorra írt variáns jóval komolyabb számítási kapacitással rendelkezett. Az kapott eredményeket kiértékeltük és megállapítottuk, hogy a Brunotte algoritmus jóval érzékenyebb a halmazműveletek sebességére, mint a számítási teljesítményre.
Publications of the author


References


REFERENCES


