matcos-13

Proceedings of the 2013 Mini-Conference on Applied Theoretical Computer Science

University of Primorska Press

MATCOS-13
Proceedings of the 2013 Mini-Conference on Applied Theoretical Computer Science

Edited by
Miklos Kresz and Andrej Brodnik

Reviewers and Programme Committee
Andrej Brodnik ▪ Ljubljana, Koper, Slovenia, co-chair
Gabor Galambos ▪ Szeged, Hungary
Gabriel Istrate ▪ Timisoara, Romania
Hans Kellerer ▪ Graz, Austria
Miklos Kresz ▪ Szeged, Hungary
Silvano Martello ▪ Bologna, Italy
Andras Recski ▪ Budapest, Hungary
Gerhard Reinelt ▪ Heildeberg, Germany
Jiri Sgall ▪ Prague, Czech Republic
Borut Žalik ▪ Maribor, Slovenia

Steering Committee
Andrej Brodnik ▪ Ljubljana, Koper, Slovenia
Gabor Galambos ▪ Szeged, Hungary
Miklos Kresz ▪ Szeged, Hungary
Gerhard Reinelt ▪ Heildeberg, Germany

Organizing Committee
Janez Žibert, chair
Tine Šukljan
Marko Grgurovič

Published by
University of Primorska Press
Titov trg 4, SI-6000 Koper

Editor-in-Chief
Jonatan Vinkler

Managing Editor
Alen Ježovnik

Koper, 2016

ISBN 978-961-6984-20-1 (pdf)
www.hippocampus.si/ISBN/978-961-6984-20-1.pdf
ISBN 978-961-6984-21-8 (html)
www.hippocampus.si/ISBN/978-961-6984-21-8/index.html

CIP - Kataložni zapis o publikaciji
Narodna in univerzitetna knjižnica, Ljubljana

004(082)(0.034.2)

MINI-Conference on Applied Theoretical Computer Science (2013 ; Koper)
MATCOS-13 [Elektronski vir] : proceedings of the 2013 Mini-Conference on Applied Theoretical Computer Science / edited by Miklos Kresz and Andrej Brodnik. - El. knjiga. - Koper : University of Primorska Press, 2016

Način dostopa (URL): http://www.hippocampus.si/isbn/978-961-6984-20-1.pdf
Način dostopa (URL): http://www.hippocampus.si/isbn/978-961-6984-21-8/index.html

ISBN 978-961-6984-20-1 (pdf)
ISBN 978-961-6984-21-8 (html)
1. Gl. stv. nasl. 2. Krész, Miklós
283625728

Preface

Dear reader,

In the cooperation of the University of Primorska and the University of Szeged we have decided to organize a conference in 2010, which reflects both the theoretical and applied aspects of computer science. The Mini-Conference on Applied Theoretical Computer Science (MATCOS - 10) was a two-day event in the frame of which a student session was organized. This occasion gave the opportunity for distinguished Master students and early stage PhD students to present their preliminary results and ongoing research. It turned out that our initiative was very successful; the full papers of 9 student talks were honoured to contribute to the post-proceedings of this event.

Continuing the tradition, the Middle-European Conference on Applied Theoretical Computer Science (MATCOS-13) also secured the place for high-level student research. This post-proceeding is devoted to publish the carefully reviewed full papers presented at MATCOS-13 held on October 10th and 11th, 2013 in Koper, Slovenia. The topics of the contributions cover a wide range of theory and application, reflecting the scope of the conference: algorithmic solutions for life science problems, artificial intelligence methods for games and computer vision, compiler construction, advanced data structures, application-oriented scheduling and numerical simulation. All papers presented promising results, proving the excellent research attitude of the contributors with showing the potential for a scientific career.

The high standard of the presentations in the regular session convinced us to collect selected papers for a special issue in the international journal Informatica, which was published as number 3 in volume 39, 2015. The success both in organizing the meeting and in publishing the best results of the conference motivate us to make this event a regular meeting. Our plan is to organize the next conference in October 2016 in Koper with involving again regular and student sessions.

MATCOS-13 was smoothly organized, we are very grateful to Janez Žibert, chair of Organizing Committee and to “his team”. Special thanks to Professor Silvano Martello for accepting our invitation as a keynote speaker and giving his nice talk. The success of the conference would have not been possible without the aid of programme committee, while the professional work provided by the University Primorska Press in the publishing process was also indispensable. Thank you for all their support.

Miklós Krész, chair of student conference

Contents

Efficient implementation of algorithms for solving subgraph isomorphism problem in cheminformatics

Mónika Vigula
Eötvös Loránd University
Budapest, Hungary
vigula.monika@gmail.com

ABSTRACT

In a typical task in cheminformatics [1], we have to retrieve all target molecules from a given database containing a query molecule as substructure. Representing the molecular structures as labeled graphs, this task can be formulated as the subgraph isomorphism problem, which is a well-known NP-complete problem.

In our work, we have studied three algorithms solving the subgraph isomorphism problem. We have implemented the Ullmann algorithm [5] and the VF2 algorithm [2] along with the atom re-ordering step of QuickSI [4] and additional heuristics. The time and memory requirements of these algorithms were also evaluated. We have determined the combination of heuristics resulted in the best performance on real data sets as well.

Keywords

cheminformatics, subgraph isomorphism problem, substructure search

Supervisors

Krisztián Tichler[1] and Péter Kovács[2]

1. INTRODUCTION

Cheminformatics is a rapidly growing field which implies interesting challenges for information technology beyond chemical background. It helps us to select pharmaceutical research directions and reduce the numbers of costly research tests. We try to predict the possible outcome of an experiment using our mathematical and information technological knowledge.

In a typical task, a database containing a lot of molecules (mostly hundreds of thousands or millions) and a query molecule are given. We have to find all the target molecules from the database containing the query molecule as substructure (see Figure 1). In the first step, we represent the molecules as undirected, labeled graphs. Then, we check for each target molecule whether it contains the query molecule or not. As this is a well-known NP-complete problem, screening methods are used as preliminary step. Our purpose is to identify quickly as many of those molecules from the database that cannot contain the query as substructure. Only the remaining molecules are to be examined by the actual substructure search algorithm.

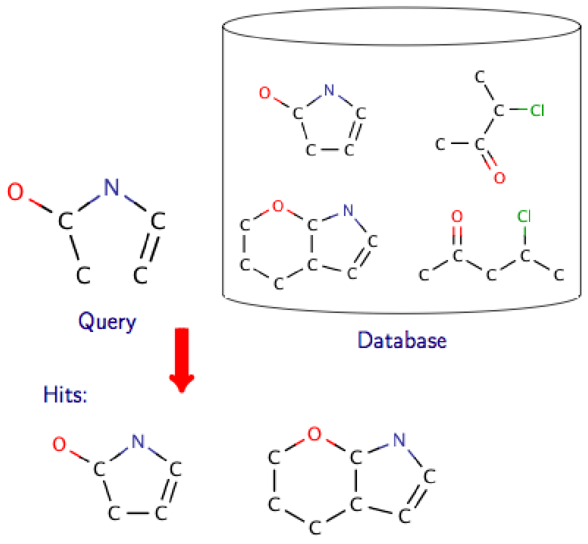

Figure 1: Our motivation

This problem has been studied extensively for decades. One of the most common solution methods is the algorithm proposed by J. R. Ullmann [5] in 1976, the VF2 algorithm [2] published by Cordella et. al. in 2001 is also widely used, while the QuickSI algorithm [4] is a quite new algorithm, it was published in 2008.

Our aim was to overview algorithms related to the subgraph isomorphism problem and implement some of these methods efficiently using heuristics that exploit characteristics of molecular graphs to improve performance (e.g., bounded degree, vertex and edge labels). In some applications, it is also necessary to compute all possible mappings between

[1]Eötvös Loránd University, Hungary, Institute of Informatics, ktichler@inf.elte.hu
[2]Eötvös Loránd University, Hungary, Institute of Informatics, kpeter@inf.elte.hu

the query and the target structures, thus we also considered this problem. We tested the correctness and efficiency of our implementations on real data sets.

2. THE SUBGRAPH ISOMORPHISM PROBLEM

Definition 1. A labeled graph is defined as a 6-tuple $G = (V, E, \Sigma_V, \Sigma_E, l_V, l_E)$ where V is the set of vertices, E is the set of undirected edges, Σ_V and Σ_E are sets of vertex and edge labels, and $l_V : V \to \Sigma_V$, $l_E : E \to \Sigma_E$ denote label functions mapping a vertex or an edge to a label, respectively.

Definition 2. A graph $G_1 = (V_1, E_1, \Sigma_{V_1}, \Sigma_{E_1}, l_{V_1}, l_{E_1})$ is subgraph isomorphic to $G_2 = (V_2, E_2, \Sigma_{V_2}, \Sigma_{E_2}, l_{V_2}, l_{E_2})$ (denoted by $G_1 \subseteq G_2$) if there exists an injective function $f : V_1 \to V_2$ satisfying

1. $\forall u \in V_1 \ (l_{V_1}(u) = l_{V_2}(f(u)))$
2. $\forall u, v \in V_1 \ ((u, v) \in E_1 \Rightarrow (f(u), f(v)) \in E_2)$
3. $\forall u, v \in V_1 \ ((u, v) \in E_1 \Rightarrow l_{E_1}((u, v)) = l_{E_2}((f(u), f(v))))$

Notice that G_2 does not necessarily contain G_1 as induced subgraph. A simple example is shown in Figure 2. A possible mapping is

$f(1) = 14$	$f(5) = 5$	$f(9) = 12$
$f(2) = 2$	$f(6) = 8$	$f(10) = 3$
$f(3) = 1$	$f(7) = 4$	
$f(4) = 6$	$f(8) = 11$	

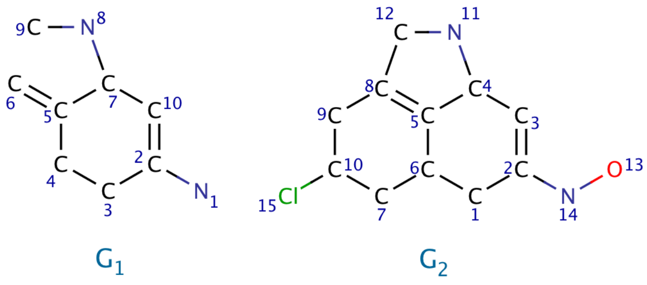

Figure 2: An example where G_2 contains G_1 as substructure

Note furthermore, that hydrogen is usually not represented in molecules as its presence can be deduced from our chemical knowledge.

In cheminformatics, query molecule, target molecule, atoms, bonds and substructure are usually used instead of G_1, G_2, vertices, edges and subgraph, respectively, therefore these notions are used in the rest of the paper.

3. SCREENING

A preprocessing step, called screening, is used to exclude as many molecules as possible from the target database that cannot contain the query molecule as substructure. Our aim was not to study and improve the existing screening methods, therefore only some simple conditions were checked before running the implemented substructure search algorithms. If *query* $\subseteq$ *target* then the atom/bond count of the query is less than or equal to the atom/bond count of the target. Similar inequalities hold for various types of atoms (e.g., *C*, *N*, *O*) and bonds (e.g., *simple*, *double*, *triple*) as well.

In another screening method, two matrices are calculated for each molecule. Columns and rows correspond to frequent atom types[3] (e.g., *C*, *N*, *O*) and bond types (e.g., *simple*, *double*), respectively. Initially, all matrix elements are zero. The bonds of the molecules are examined sequentially. If the current bond connects two atoms of frequent type, then the two elements of the matrix are increased by one. It can easily be proved that each element of the query's matrix is less than or equal to the corresponding element in the target's matrix if *query* $\subseteq$ *target*.

Simple example molecules are shown in Figure 3. The corresponding matrices are shown in Table 1. The elements of the matrices of the query and the target are separated by a colon. As the query's matrix contains non-zero elements and all elements are zero in the target's matrix, *query* $\nsubseteq$ *target* holds.

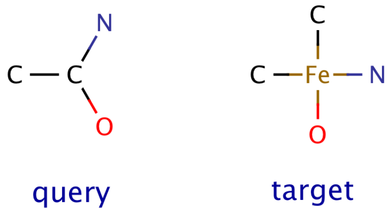

Figure 3: Example molecules to screening

$$\begin{array}{r|ccccccc} & C & N & O & F & P & S & \dots \\ \hline single & 4:0 & 1:0 & 1:0 & 0:0 & 0:0 & 0:0 & \dots \\ double & 0:0 & 0:0 & 0:0 & 0:0 & 0:0 & 0:0 & \dots \\ triple & 0:0 & 0:0 & 0:0 & 0:0 & 0:0 & 0:0 & \dots \end{array}$$

Table 1: Matrix

4. ALGORITHMS

We have studied three well-known substructure search algorithms, the Ullmann algorithm [5], the VF2 [2], and the atom-reordering step of the QuickSI [4]. Although they are all backtracking algorithms, they build the backtracking trees in different ways. Furthermore, QuickSI rearranges the atoms of the query molecule in the first step to achieve better performance. Each node of the backtracking tree represents a partial isomorphism from the query to the target molecule. As this tree can be exponential in size, the purpose of the algorithms is to filter out nodes that can not be extended to achieve a subgraph isomorphism function. Therefore, different *feasibility functions* are applied by the methods.

The Ullmann algorithm maintains a boolean matrix (denoted by M) representing the possible branches at the different depths. $M_{i,j}$ is true if the current partial mapping can be extended by adding $f(i) = j$ to achieve a subgraph

[3]The frequent atom types were determined based on a public molecule set of National Cancer Institute [3].

isomorphism between the query and the target. This matrix is refined after each step: the algorithm checks for each pair of i and j where $M_{i,j}$ is true whether the non-mapped neighbors of the i-th atom in the query can be mapped to a neighbor of the j-th atom in the target. If this condition does not hold, then $M_{i,j}$ is set to false and a new refine phase is performed.

The VF2 algorithm maintains a *neighbor set* for each molecule (Nb_q, Nb_t) which contains those atoms that are not involved in the current partial mapping but have a neighbor that is already involved. The algorithm distinguishes the following two cases.

1. $Nb_q \neq \emptyset$. In this case, the next atom to be mapped is the element of this set with the minimum index. This atom can be mapped only to an atom from Nb_t. Before the algorithm extends the current partial mapping by adding $f(i) = j$, it checks if the atom and the bond types match and $|Nb_q| \leq |Nb_t|$ holds for the new neighbor sets.
2. $Nb_q = \emptyset$. In this case, the non-mapped query atom with the minimum index is considered, which can be mapped to any non-used atom of the target. Before the extension of the current partial mapping, only the atom types are checked.

5. HEURISTICS

In this section, we briefly introduce the heuristics we have developed to improve upon the considered algorithms.

5.1 Parent heuristic

Definition 3. Suppose the atoms of the query are indexed by positive integers and a partial mapping is given. Let q_i $(i \in \{1, \ldots, |V_1|\})$ be an arbitrary non-mapped atom of the query molecule. Define the parent atom of q_i (denoted by $parent(q_i)$) as its mapped neighbor atom with the minimum index (if it has any).

Notice that at least one atom in each component does not have a parent atom (i.e., the atom mapped first in its component). Furthermore, note that the parent atoms are not modified by extensions of the current partial mapping. The key observation that we exploited is that a query atom q_i can only be mapped to those atoms of target that are adjacent to the image of $parent(q_i)$. Search algorithms can effectively take advantage of this property in case of molecular graphs, because they are very sparse.

A simple example is shown in Figure 4 to demonstrate how this heuristic can be applied. Suppose that the current mapping is $f(1) = 4$, $f(2) = 5$, and $f(3)$ is under consideration. The parent atoms are indicated by arrows pointing from an atom to its parent atom. As $f(parent(3)) = 4$, which has two neighbors that are not mapped yet, namely 2 and 6, the only possible assignments are $f(3) = 2$ and $f(3) = 6$.

5.2 Atom order in the query molecule

H. Shang et. al. introduced a new method in [4] to obtain better performance of search algorithm. Their idea is to rearrange the atoms of the query molecule according to the

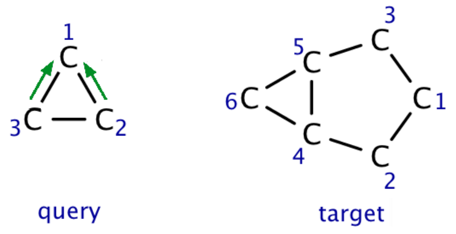

Figure 4: The parent heuristic

frequency of the different atom and bond types in the given database. They suggested mapping the rare atoms before the frequent ones while keeping connectivity. This new atom order needs to be calculated only once before any search against the database.

Although this method has been shown to reduce search time, it has some limitations. In its original form, it is applicable only for connected query graphs. However, as in many real-world applications, the query structure may consist of multiple components, we have extended this method to be applicable for such queries. For example, an isolated atom can be mapped to any atom of the target having the same atom type; therefore it is beneficial to consider isolated atoms last. Furthermore, if we apply this atom ordering technique, then the calculation of parent atoms should also be adjusted.

5.3 Ullmann Algorithm

We have successfully applied the aforementioned two heuristics in the Ullmann algorithm. They both substantially decrease its running time.

In addition, we could also decrease the memory requirement of the algorithm. Its straightforward implementation requires $O(n^3)$ space (where $n = max\{|V_1|, |V_2|\}$), which can be reduced to $O(n^2)$ by saving only the modified matrix elements at each depth instead of saving the entire boolean compatibility matrix. Because every element can be set from *true* to *false* only once under a node in a backtracking tree, at most $O(n^2)$ positions need to be saved in total.

Another improvement exploits that the compatibility matrix contains exactly one *true* value in the first d rows, where d denotes the current search depth. Therefore, the matrix refinement step can be started at the $(d + 1)$-th row.

5.4 VF2

We have also devised a few improvements for the VF2 algorithm. First, the candidate sets are not pre-calculated and not stored in memory. If the next candidate is needed, it can be calculated on the fly. Additionally, if only the first mapping is required, then there is no need to calculate those candidates that might not be used in a later step.

Although comparing the cardinality of neighbor sets improves the performance of the original algorithm, we found that it becomes superfluous when the parent heuristic is also applied. In fact, this new heuristic developed by us seems to be clearly superior to the original technique.

Apart from that, the atom reordering heuristic is also applied to further reduce the running time of the algorithm. Moreover, an additional improvement is based on the observation that the order of the atoms in the query can be fixed in the first step, thereby avoiding to find the atom with the minimum index at each step.

6. TEST CASES, SPEED AND MEMORY RESULTS

We have compared the implemented algorithms combined with all the improvements mentioned above in different aspects. We have studied the time requirement for finding either the first or all substructure mappings for molecules having different size. Our test suite consists of three instance sets: (1) small queries and small targets[4]; (2) small queries and large targets, and (3) large queries and large targets. The test molecules were selected from a public molecule database of National Cancer Institute [3]. The different test cases are summarized in Table 2. The first and second row of each cell corresponds to the queries and targets, respectively. Some of the 20 small, frequent test molecules are shown in Figure 5.

The heuristics mentioned above have decreased the search time of the Ullmann algorithm and VF2 by 35-40%. The running time can be further reduced by 10% if the algorithms are implemented in iterative way instead of recursive way. Finding all mappings required between 1.5 and 3 times as much running time as finding only the first mapping.

Our results show that the VF2 algorithm is typically much faster than the Ullmann algorithm. We found that the refinement of the initial boolean matrix used by the Ullmann algorithm takes about 70% of the total search time.

The reduced search time for 1000 query-target pairs are shown in Table 3.

		Query molecules	
		small	large
Target molecules	small	20 small molecules molecules having 11-15 atoms	–
	large	20 small molecules molecules at least 76 atoms	large, similar molecules

Table 2: Test cases

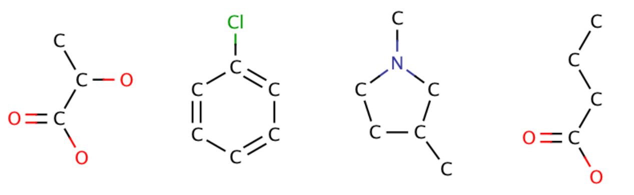

Figure 5: Some small query molecules

[4]In cheminformatics, molecules having at most 15 non-hydrogen atoms are considered as small molecules.

	Ullmann alg.		VF2	
	first	all	first	all
small queries and targets	0.004 s	0.005 s	0.001 s	0.002 s
small queries and large targets	0.067 s	0.136 s	0.009 s	0.013 s
large queries and large targets	0.27 s	1.084 s	0.01 s	0.027 s

Table 3: The search time for 1000 query-target pairs at finding the first/all mapping

7. CONCLUSION

This paper presents our results for solving the subgraph isomorphism problem. We have implemented the Ullmann algorithm, the VF2 algorithm, and the atom-reordering step of QuickSI along with various heuristics that we devised to improve upon them. We have compared the memory and time requirements of the implementations on real-world molecular graphs having different size. The search time has been reduced by 35-40% by applying the developed heuristics.

8. ACKNOWLEDGMENTS

I would like to express my gratitude to Krisztián Tichler and Péter Kovács, my supervisors for their patient guidance, enthusiastic encouragement and useful critiques of this research work. I am grateful to István Fekete for introducing me to cheminformatics and for his continuous support. I am obliged to staff members of ChemAxon Ltd. for the valuable information provided by them in their respective fields.

The research project presented in this paper is supported by the European Union and co-financed by the European Social Fund (grant agreement no. TÁMOP 4.2.2./B-10/1-2010-0030).

Marvin was used for drawing, displaying and characterizing chemical structures, substructures, and reactions, Marvin 6.0.5, 2013, ChemAxon (`http://www.chemaxon.com`).

9. REFERENCES

[1] N. Brown. Chemoinformatics – an introduction for computer scientists. *ACM Computing Surveys*, pages 8:1–8:38, 2009.

[2] L. P. Cordella, P. Foggia, C. Sansone, and M. Vento. An Improved Algorithm for Matching Large Graphs. *In: 3rd IAPR-TC15 Workshop on Graph-based Representations in Pattern Recognition, Cuen*, pages 149–159, 2001.

[3] National Cancer Institute. NCI Open Database Compounds, 2000. `http://cactus.nci.nih.gov/download/nci/`.

[4] H. Shang, Y. Zhang, X. Lin, and J. X. Yu. Taming Verification Hardness: An Efficient Algorithm for Testing Subgraph Isomorphism. *Proceedings of the VLDB Endowment*, pages 364–375, 2008.

[5] J. R. Ullmann. An algorithm for subgraph isomorphism. *Journal of the ACM*, 23:31–42, 1976.

Adaptive Algorithms For Dynamic Programming With Applications to Bioinformatics

[Extended Abstract]

Marko Grgurovič
University of Primorska
Titov Trg 4
Koper, Slovenia
marko.grgurovic@student.upr.si

ABSTRACT

In this paper we study a simple computation that lies at the core of many dynamic programming algorithms, perhaps most notably those for computing the edit distance between two strings. Past solutions have assumed properties of the input such as convexity and concavity and would not work on general inputs. We obtain an algorithm that combines the previous algorithms in a way that makes no assumptions on the input, yet its running time depends on a measure of "sortedness" present in the input. This measure turns out to be very natural, and if the input comes from a function, it corresponds to the number of inflection points of the input function. An immediate consequence of the result is that if the input comes from a polynomial of degree d, then the running time of the algorithm can be upper bounded by a function of d. The new algorithm extends the previous results to a wider family of functions and is never worse than either special cases.

Categories and Subject Descriptors

F.2.2 [**Theory of Computation**]: Nonnumerical Algorithms and Problems

General Terms

Algorithms, Theory

Keywords

Dynamic programming, edit distance

1. INTRODUCTION

We consider the following simple computation: given a sequence of values $X_1, ..., X_n$ and a function $g(k)$ for $1 \leq k \leq n$, compute for all $1 \leq i \leq n$:

$$Y_i = min_{1 \leq k < i} X_k + g(i-k). \tag{1}$$

It can be shown that certain dynamic programming algorithms, including algorithms for the edit distance problem and algorithms used in geology [4] and in speech recognition [3], can be reduced to this simple computation. The straightforward algorithm for computing Eq. 1 takes $O(n)$ time per i, which amounts to a total of $O(n^2)$ time over all i.

Previous algorithms for this problem focused on specific types of the function $g(\cdot)$ (usually referred to as the *gap function*), e.g. those that satisfy the quadrangle inequality [1, 5, 2] or the inverse quadrangle inequality [1]. None of these algorithms produce the correct answer when $g(\cdot)$ does not satisfy their assumptions. In this paper, we develop a combination of these approaches that works for all inputs, but has a running time that depends on the function $g(\cdot)$ in a natural way.

All logarithms in this paper are in base two.

2. THE CONVEX AND CONCAVE CASE

In this section we describe the algorithm of [1] for the convex case. We do not explicitly consider the concave case, since it is essentially analogous.

Consider the case when $g(\cdot)$ is a convex function, that is a function which increases at an increasing rate. In other words, given $a < b$ the following residues are implicitly sorted:

$$g(b+c) - g(a+c) \leq g(b+c') - g(a+c') \quad for \quad 0 \leq c \leq c'.$$

Now consider the line $1, 2, ..., n$. We will assign to each point i on this line the value $Y[i]$, i.e. the combination that achieves the minimum for a given i in Eq. 1. Observe that we can represent $Y[i]$ with X_k, since given some X_k and i the choice of gap is implicit (i.e. $g(i-k)$). For each i in the list we would like to find the X_k which achieves the minimum for that i.

LEMMA 2.1. *If X_k achieves the minimum for some i, but does not achieve the minimum for $i+1$, then it does not achieve the minimum for any $i' > i$.*

PROOF. Let X_j be the element that achieves the minimum for $i+1$. Then:

$$X_k + g(i+1-k) \geq X_j + g(i+1-j)$$
$$X_k - X_j \geq g(i+c-j) - g(i+c-k).$$

Which holds for all $c \geq 1$, since the residues on the right-hand side at most decrease. $\square$

What follows from Lemma 2.1 is that each element from X occupies at most one interval on our line $i = 1...n$.

We would also like to know, given two elements X_k and X_j and $j < k$, for which gap values will X_k be smaller than X_j. This leads to the inequality:

$$X_k + g(a) \leq X_j + g(b) \quad \forall b, a : b - a = k - j.$$

Rearranging this gives us

$$X_k - X_j \leq g(k+c) - g(j+c).$$

We can now find the maximal value of c for which the inequality still holds in $O(\lg n)$ by employing binary search.

Now we are ready to describe the algorithm. First we assign X_1 as the interval that covers the entire line. Then, the algorithm works by traversing the list of candidates $X_2, ..., X_n$ and comparing the current candidate X_k with the element X_j, which corresponds to the rightmost interval on the line. One can now determine for which values of i the new candidate is better, and the interval can be updated. In the case that the new candidate is better than the previous one for the entire interval, we simply remove the old candidate, and repeat the process on the rightmost remaining interval.

In order to analyze the time required by the algorithm, we note that each new candidate from X will perform at least one binary search, which takes $O(\lg n)$ time. Multiple binary searches may be performed once intervals are removed from the rightmost end of the line, but observe that each element X is only added once, so after being removed it is never reinserted. Therefore, the algorithm takes $O(n \lg n)$ time.

3. ALGORITHM

Observe that the function g only takes on integral inputs. Therefore, we can consider a more general computation: given a sequence of values $X_1, ..., X_n$ and a sequence of values $G_1, ..., G_n$ for $1 \leq k \leq n$, compute for all $1 \leq i \leq n$:

$$Y_i = min_{1 \leq k < i} X_k + G_{i-k}. \tag{2}$$

This allows us to do away with the strict convexity requirements. Consider the sequence of values:

$$G_2 - G_1, G_3 - G_2, ..., G_n - G_{n-1}.$$

In a preprocessing step, we can traverse this sequence to discover blocks which contain elements in ascending (resp. descending) order. These are precisely the regions of the "function" which are convex (resp. concave). Build the list L which, for each block, contains an element (*start*, *end*, *asc*) that stores information about the block boundary (*start*, *end*) and whether the block is in ascending ($asc = 1$) or descending ($asc = 0$) order. Let us denote the number of such blocks $b_1, b_2, ..., b_B$ by B, and let $|b_i|$ denote the number of elements in block b_i. Observe that $1 \leq B \leq n/2$, since two elements form either an ascending or descending sequence, and in the case $B = 1$ the "function" is either convex or concave. This step takes $O(n)$ time, and allows us to partition G into disjoint intervals where the property of sorted residues holds.

The algorithm works by moving through the list L and for each block (*start*, *end*, *asc*), it finds the optimal combinations of $Y_i = X_k + G_{i-k}$ whenever $i - k$ is inside the interval (*start*, *end*). Once these combinations have been found, they are stored as Y_i if they are lower than the current value and the algorithm moves onto the next block, where the same process is repeated. Observe that these subproblems can be solved by a slight modification of the algorithms from [1], where we use the convex algorithm if $asc = 1$ and the concave algorithm otherwise. Once we solve the subproblem in a block, we will not revisit it and so the space can be reused. The algorithm requires $O(n)$ extra space, which matches the space requirements of [1].

The time required to solve the subproblem on block b_i comes from the algorithm described in Section 2. In our case, the binary search is performed on b_i, so the time is $O(n \lg |b_i|)$. Over all subproblems the time becomes:

$$O(n \sum_{i=1}^{B} \lg |b_i|).$$

Turning the sum of logarithms into the logarithm of the product we get:

$$O(n \lg(\prod_{i=1}^{B} |b_i|)).$$

Assume B is fixed. Since the logarithm is a monotonically increasing function, the time is maximized when $\prod_{i=1}^{B} |b_i|$ is maximized. Recall that the geometric mean is less than or equal to the arithmetic mean. Then we have:

$$(\prod_{i=1}^{B} |b_i|)^{1/B} \leq n/B$$
$$(\prod_{i=1}^{B} |b_i|) \leq (n/B)^B.$$

Thus, we can upper bound the time by $O(Bn \lg(\frac{n}{B}))$. Regardless of G, the time is never worse than the straightforward $O(n^2)$ algorithm, and achieves the $O(n \lg n)$ bound when $B = O(1)$. If the elements of G come from a function $g(\cdot)$, which is defined in the domain $[1, n]$, then B can be upper bounded by the number of inflection points[1] of $g(\cdot)$ in that region. Recall for example, that a polynomial of degree d has at most $d - 2$ inflection points. Therefore, if the values in G come from a polynomial, we can upper bound the running time by its degree.

4. FUTURE WORK

Since we are no longer working with functions explicitly, but rather sequences, we can also obtain a speedup by working on X in an analogous way. For example, we could traverse X

[1]These are points where convexity changes into concavity and vice-versa.

and find blocks, then decide to use either X or G, whichever has fewer blocks. However, in most dynamic programming applications, the minimum computation that we study is used as a subroutine. In these cases, X changes between calls to the subroutine, whereas G remains static. Hence, it makes sense to analyze G, especially since it tends to be a particular mathematical function. However, working on X would have its merits if one could show that, for a given application, the number of blocks B in X between subroutine calls can be bounded by some value.

Similar techniques could perhaps be applied to other dynamic programming algorithms that operate on the $(\min, +)$ semi-ring. One notable example is $(\min, +)$ matrix multiplication.

5. REFERENCES

[1] Z. Galil and R. Giancarlo. Speeding up dynamic programming with applications to molecular biology. *Theoretical Computer Science*, 64(1):107 – 118, 1989.

[2] D. S. Hirschberg and L. L. Larmore. The least weight subsequence problem. In *Proceedings of the 26th Annual Symposium on Foundations of Computer Science*, SFCS '85, pages 137–143, Washington, DC, USA, 1985. IEEE Computer Society.

[3] D. Sankoff and J. B. Kruskal. *Time warps, string edits, and macromolecules.* Cambridge University Press, Cambridge, England, 2000.

[4] M. S. W. T. F. Smith. New stratigraphic correlation techniques. *Journal of Geology*, (88):451 – 457, 1980.

[5] F. F. Yao. Speed-up in dynamic programming. *SIAM J. on Alg. Discr. Meth.*, (3):532 – 540, 1982.

An Algortihm for Recognition of Position Repetition in Chess

Nándor Németh
Eötvös Loránd University
Department of Software Technology and Methodology
nem.nandor@gmail.com

ABSTRACT

A chess game ends in draw if the same position appears for the third time. In a chess program, we need to implement this rule to avoid losing some winning positions, and not to lose positions, where we have the possibility to make the match a draw. Recognizing position repetitions is not trivial problem. The algorithm must run in most of the positions we reach in the game tree, so it must run quickly. We can read about a lot of algorithms which recognize the repetitions. There are two groups of these algorithms: correct ones, and statistically correct algorithms having faster solutions. This paper demonstrates a correct solution which is simpler and faster than any other correct algorithm, and it can work with any board representation. We can check some statistically correct algorithms quickly with a correct method. Therefore we can get a more faster correct algorithm for the problem.

Categories and Subject Descriptors

I.2.1 [**Artificial Intelligence**]: Applications and Expert Systems—*games*

General Terms

Algorithms

Keywords

chess programming, position repetition, move sequences

1. INTRODUCTION

Among the laws of chess, we can find rules which guarantee that games must end in finite moves. One of them is the rule of the position repetition. When the same repetition appears at the third time on the board, any of the players can ask for a draw. Although the fifty-move rule is easily implementable in chess programs, the algorithms for recognizing position repetitions are very complex and slow. Therefore, some chess programs omit this rule. This can cause some enormous mistakes, roughly in every second game. Nowadays we can find several solutions for the problem in sophisticated chess programs[2]. Some of the programs use correct algorithms for the problem. Others use statistically correct methods which run faster, and rarely make mistakes. In this paper, the author demonstrates a new correct method for the problem.

2. DEFINITIONS

To understand this paper more easily, let us describe some definitions:

Two positions are identical if each square of the board contains the same piece, the same player has to move, and the move sentences, the payers can make, are identical, too.

The normal search tree is a graph. The root vertex is the position we can see on the chessboard. The other vertices are the positions the program reaches during the search. The edges represent the moves. Let us extend this tree with all the positions which have appeared on the chessboard during the chess game. So, the root vertex of the new graph is the initial position. In this paper, let us name this graph *extended search tree.*

There are moves, such as pawn moves or captures, when the position before the move can not be created after the move. Let us say them *boundary moves* in the extended search tree. Let us name the position after a boundary move *boundary position.* To use the definitions more easily, name the initial position boundary, too. Therefore, all positions are boundary or have a boundary position created previously in the extended search tree.

The *examined position* is the position which is just examined by the chess program.

3. ALGORITHMS REALIZED IN WELL-KNOWN PROGRAMS

There are some solutions for recognizing position repetitions[2]. The programs usually recognize the first repetition. It is enough for the correct control. If one of the players can make a better move than making a repetition, then the repetition will not appear on the board. If one player can make a repetition independently from the opponent, then he can produce easily the position at the third time, too. Therefore, it is enough to find the first repetition, and it is not neccessary to wait for the second one. In such a way, the problem was simplified greatly. The neccessary search depth for recognizing repetitions is reduced to half of the original depth.

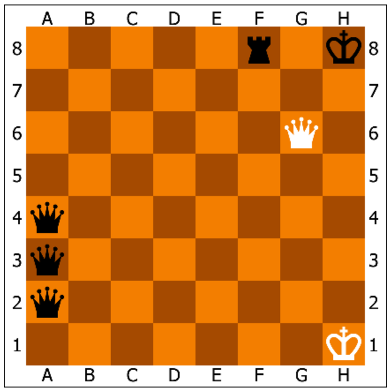

Figure 1: In the original case there are 8 plies neccessary, but 4-ply search is sufficient for recognizing the first repetition

There are two groups of the algorithms we can find in chess programs[2]. The correct methods recognize the repetitions exactly. The other methods are only statistically correct ones. Although they make mistakes, they need shorter running time.

3.1 Correct algorithms

All of the correct algorithms are based on two control methods, the position comparing control or the move control. Both of them run in linear time. Of course, we must examine all of the positions or moves after the last boundary position in the extended search tree. Therefore, the algorithm's running time depends on the number of the moves after the boundary position, linearly. This type of the algorithms can result significant slowdowns in some move sequences, when there are no pawn moves, or captures.

3.1.1 Board-comparing methods

The easiest way for the problem is the comparsion of the examined position to the previous ones. The more bits define the chess board, the slower the algorithm is. There are some methods to optimize the algorithm. Of course, we need to investigate only every second position. We can first check for example the last move's target square, and next the whole table, if there is a same type of piece on this square.

3.1.2 Computing repetitions from move sequences

In the Axon chess program there is a very interesting algorithm [4] which uses only the moves for recognizing position repetitions. Let us see the move sequence from the last boundary to the examined position. We can easily reverse this sequence. The program investigates this reverse move sequence.

When the program scans the reverse moves sequentially, it updates a data structure that stores some information about the moved pieces. In one reverse move a piece can start a new journey, continue its journey or reach its original position. If a piece starts a new journey, the program stores the piece's original and actual square. If a piece continues its journey, then the program modifies its actual square. When a piece reaches its original position, then the program deletes its record from the data structure. When all of the pieces moved reached their own original position, there isn't any record in the data structure. So, the program recognizes a position repetition. Of course, the algorithm can manage the swap of two identical pieces.

To understand this method, let us see a simple example: an examined position, and a reverse move sequence. In Table 1 we can see, what is stored in the data structure during the running of the algorithm.

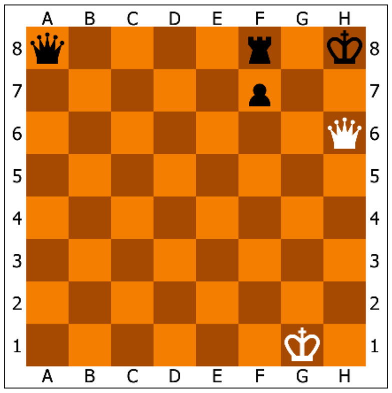

Figure 2: The examined position

The reverse move sequence is:

Q H6-G5, K H8-G7, Q G5-H6, K G7-H8, (...)

Table 1: The data structure during the reverse moves

Reverse move	The records	Numbers
Q H6-G5	Q H6-G5	1
K H8-G7	Q H6-G5, K H8-G7	2
Q G5-H6	Q H6-H6, K H8-G7	2
-	K H8-G7	1
K G7-H8	K H8-H8	1
-	-	0

3.2 Statistically correct algorithms

There are some algorithms which are faster than the correct methods, but they make mistakes occassionally. Most of them use hash keys, and hash tables[5, 3]. If the algorithm uses hash tables, it can run in constant time[3]. This is a great advantage against the correct methods in long capture-, and pawn-move-free move sequences. The program doesn't make really big mistakes frequently, because mistakes occur rarely, and only a few mistakes cause bad move on the board, most of them are disappear because of the minimax algorithm.

4. THE NEW ALGORITHM

The paper shows an algorithm based on the method of the Axon chess program[4]. The algorithm uses the reverse move sequence, and the examined position, too, to recognize the repetitions. It investigates every reverse move. If all moved pieces during the reverse moves reach a square which contains the same type of piece in the examined position, then the position is repeated. Let us name the pieces which are standing on a square which doesn't contain the same type of piece in the examined position as *moved pieces*. In the examined position the number of the moved pieces is zero. When it will be zero again, the algorithm finds a repetition. Sometimes it can happen that two identical pieces swap their position. In this case there can be repetition. Therefore, if a piece reaches a square which contains a same-type piece in the examined position, the algorithm considers it as the end of its journey.

This number is similar to the number of the records in the Axon. The great observation is that we need no data structures, only this number, to recognize repetitions.

4.1 The basis of the algorithm

The algorithm modifies the number of the moved pieces in every reverse move. When the algorithm starts to make a move, it investigates the from-square of the move. If the square in the examined position contains a piece which has the same type as the moving piece, then a piece starts a new journey, so the number of moved pieces increases by one. In other cases, its value doesn't change.

The algorithm then investigates the to-square. If it contains a same-type piece in the examined position, a piece ends its journey. Therefore, the number of moved pieces decreases by one. In other cases, its value doesn't change.

So, we have $2 \cdot 2 = 4$ cases. In the first case, the moving piece starts a new journey, and it doesn't finish the journey at the end of the move. Then, the algorithm increases the number of the moving pieces by one.

In the second case, when a piece ends its journey. Of course, the algorithm decreases the number by one.

In the third case, when a piece continues its journey. So, it has some moves, but it may make a big tour before it finishes its journey. In this case the number of the moving pieces doesn't change.

The fourth case can happen very rarely. In this situation, a piece starts its journey, and ends it at the end of the move. In this case, the number of the moving pieces increases and decreases at the same time, so its value doesn't change.

This algorithm is enough for managing the number of the moving pieces. Its great advantage is that it needs no extra data structure. It can manage the swapping of the pieces similarly with a piece which reaches its starting square during the reverse moves. In the Axon chess program the programmers had to implement two different algorithms for the problem. The running time of the algorithm is stable. In the Axon, the running time depends on the number of the moved pieces.

The running of the algorithm is usually similar to the running of the algorithm of the Axon. In Figure 2, the two algorithms are similar, but the new one uses only one number. Let us see, how the algorithm works, when two identical pieces swap their positions. In the example, only the white moves are shown. The coloumn "Cases" shows which of the four cases is applied.

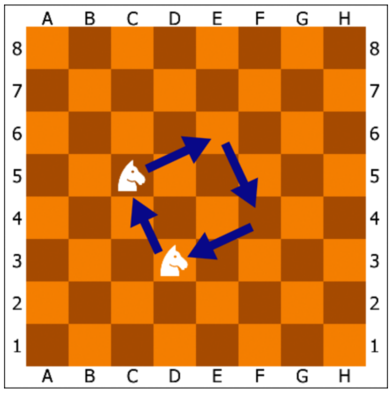

Figure 3: Swap of two knights

The reverse move sequence is:

N C5-E6, N E6-F4, N D3-C5, K F4-C5, (...)

Table 2: The modifying of the number of the moving pieces

Reverse move	Moving pieces	Case
-	0	-
N C5-E6	$0 + (1 + 0) = 1$	1
N E6-F4	$1 + (0 + 0) = 1$	3
N D3-C5	$1 + (1 - 1) = 1$	4
N F4-C5	$1 + (0 - 1) = 0$	2

4.2 Optimizing the algorithm in array-based board representation

The algorithm is good, but we can do some optimizations. The program must stop the reverse moves at the point, where the algorithm recognized the repetition, or where there can't be any repetitions. When there are less than four plies distance from the last boundary position to the examined one, then players can't make repetitions. Sometimes a lot of pieces move. If there isn't the neccessary number of moves to end all of the moved pieces' journey, then the algorithm stops the search for repetitions. Considering the fact that only every second position can be identical with the examined position, it is better if the algorithm investigates two plies during one running of the body of the loop.

4.3 The method with bitboard representations

We can define a *condition(piece,square)* boolean function which gives true if the *square* contains *piece*-type piece in the examined position, and gives false in other cases. With this function, we can generalize the algorithm for any type of board representation[1].

If a program uses bitboard representation, the function can be seen as follows: Let us consider the bitboard which stores the position of the *piece*-type pieces. Let us make an other bitboard which has only one non-zero bit. This bit is the bit of the *square*. If the algorithm uses the *xor* binary operator in this bitboards, the result is nonzero if and only if there is a *piece*-type piece in the *square* at the examined position.

5. ANOTHER USE OF THE NUMBER OF THE MOVED PIECES

In some situations the programs can evaluate the positions better, if it calculates with the measure of the change of the position. This defines a distance between two positions. The number of the moved pieces is one approximation of its value.

In the opening moves, the players try to move a lot of pieces to make an open position. In this part of the game, there are some pawn moves, too. So, we must extend the algorithm.

In the endgame, when there are long move sequences without pawn move or captures, the nearly constant position implies the draw. If a few pieces move within several plies, it seems none of the players can make a really good attack. In this case the program does well to use modified distance in some cases. Its reason is that if the original distance is zero, but not the same player has to move, then at least 5 plies must be between the two positions.

Such a use of the number of the moved pieces needs further research.

6. THE RESULTS

6.1 The test program

A test program has been written to compare some algorithms for recognizing the repetitions. The compare made in several search depth, with alpha-beta and simple minimax methods[3]. The program used all positions which appeared in 2010 at the Linares International Chess Tournament. It runs the minimax algorithm at all the positions, and measures the running times.

6.2 Comparsions with other algorithms in average case

In an average case, the new algorithm needed almost half the time comparing to the Axon's method[4]. The other correct algorithms had a little bit more longer running time. The statistically correct methods were a little bit faster. Their great advantage is that the hash keys are needed in other algorithms[3], so the slowdown of the program is smaller than that of the correct methods.

The type of the pruning also modifies the running time. The minimax method needs relatively the most extra time if the program uses repetition recognizing algorithm. The alpha-beta method needs less time. Its reason is the following: when the program uses pruning, the number of investigated long move sequences without captures and pawn moves is less.

The author investigated the running time difference between a chess program that uses the new repetition recognizing algorithm, and another one that doesn't use any repetition recognizing algorithm. The program runs a little bit slower with the algorithm. The difference reduces by the growing of the search depth. The reason may be that the repetition recognizing algorithm prunes branches from the search tree. If the program uses greater search depth, the algorithm can prune longer branches.

6.3 Running time with long move sequences

The algorithm's running time can move up to 15 times longer than in average cases. The Axon's algorithm needs time 3 times longer than the new algorithm. Therefore the algorithm used in Axon is not considered as a linear running-time algorithm. This result was expected.

6.4 Using the algorithm to control statistically correct methods

Some statistically correct methods make mistakes only if the examined position isn't repeated. This type of algorithm can be controlled with correct ones. With the new algorithm, the slowdown of the control becomes negligible. Because of the rare repetitions and the very few number of mistakes, the mixed algorithm must use only 0.4% more time than a simple statistically correct method.

7. CONCLUSIONS

The author made a new position repetition recognizing algorithm for chess. The algorithm is demonstrably correct. Its running time is linear, as the other correct algorithms. But it is faster than other correct methods. Programs can use the algorithm to control some statistically correct algorithms. This mixed algorithm is correct of course, and has a running time as fast as simple statistically correct algorithms. So the author could combine the advantages of the two type of repetition recognizing algorithms.

8. ACKNOWLEDGMENTS

The author is grateful to his supervisor Tibor Gregorics for the motivation and the continuous guidance. The author is also thankful to István Fekete, because of his valuable suggestions; and to Gábor Gévay, who gave a lot of great ideas about the algorithm.

9. REFERENCES

[1] Chess programming - board representation. http://chessprogramming.wikispaces.com/ Board+Representation, 2012. [Online; accessed 07-Sept-2013].

[2] Chess programming - repetitions. http://chessprogramming.wikispaces.com/Repetitions, 2013. [Online; accessed 07-Sept-2013].

[3] T. Marsland. Computer chess and search. Technical report, Computing Science Department, University of Alberta, 1992.

[4] V. Vuckovic and D. Vidanovic. An algorithm for the detection of move repetition without the use of hash-keys. *Yugoslav Journal of Operations Research*, 17(2):257–274, September 2007.

[5] A. L. Zobrist. A new hashing method with application for game playing. Technical report, Computer Sciences Department, University of Wisconsin, Madison, Wisconsin, 1969.

Comparison between a cache-oblivious range query data structure and a quadtree

Tine Šukljan
University of Primorska
Institute Andrej Marušič
Muzejski trg 2
Koper, Slovenia
tine.sukljan@upr.si

ABSTRACT

After the introduction of the cache-oblivious model by Frigo et al. in 1999, a lot of research has been done about data structures that work well in multi-level memory hierarchies. Range reporting, as one of the fundamental problems in computational geometry, received a lot of attention. As there was a lot of research done in this area, there are very little implementation and comparisons between cache-oblivious and "non"-cache-oblivious data structures.

In this article we implemented one of the cache-oblivious data structures for range queries and compared it with the quadtree. The results show, that the cache-oblivious data structure answers the queries much faster, but at the expense of bigger space consumption.

Keywords

cache-oblivious, range queries, quadtree

1. INTRODUCTION

The memory systems in modern computers normally consist of several levels in a hierarchy, typically of cache, main memory and disk. The access time of different levels vary by orders of magnitude. To improve the running time of accessing the data far from the processor, the data are often moved from the farthest levels to the close ones in big blocks. It is important to design the algorithms and data structures to support high locality in the memory-access patterns.

When working in RAM model, one assumes a flat memory system with uniform access time. Because of that most of the algorithms and data structures exhibit low memory-access locality and are not efficient in a hierarchical memory system. A lot of work has been done in a two-level memory model (I/O model) introduced by Agarwal and Vitter in 1988[3] to model the high difference in access time between memory and disks. Much less work has been done in multi-level hierarchy, mostly because many parameters are used in such models to describe the memory levels. Until the introduction of the cache-oblivious model by Frigo et al. in 1999[6] that allows obtaining algorithms that are efficient in multi-level memory hierarchy without the use of complicated models.

Range reporting is one of the most studied problems in computational geometry. Given a set S in R^d, the problem is to preprocess S, so that for a given query range q, the points in $S \cap q$ are reported quickly. There exist many types of queries as axis-aligned boxes, circles, halfspaces. If R^2 there exists special cases like the two-sided and three-sided range reporting. In the case of the two-sided range reporting, it consists of an axis-aligned box with two adjacent boundaries fixed at ∞ (or $-\infty$ respectively). Similarly, a three-sided range reporting consists of a axis-aligned box with one boundary fixed at ∞ (or $-\infty$ respectively). A four-sided query is also called orthogonal.

In this paper we try to make a comparison of data structures for orthogonal range queries. Specifically between a classical data structure for range query which was developed for the RAM model and a data structure created in the cache-oblivious model.

1.1 Related work

Much of research has been done about cache-oblivious data structures for range reporting, range counting and dominance in various dimension after the introduction of the cache-oblivious model. Most notable were the cache-oblivious B-Trees structures[5] with $\mathcal{O}(\log_B N)$ search and update time. All the structures rely intensely on the so-called van-Emde-Boas layout for storing a balanced constant-degree tree of size $\mathcal{O}(N)$ in memory so that any root-to-leaf path can be traversed cache-obliviously in $\mathcal{O}(\log_B N)$ memory transfers[7].

Agarwal et al.[2] were the first to develop a cache-oblivious version of a two-dimensional range tree that answers planar range queries in the optimal $\mathcal{O}(\log_B N + T/B)$ memory transfers using $\mathcal{O}(\log_2^2 N)$ space. The downside was that B has to be of the form $B = 2^{2^c}$ for some non-negative integer constant c. Later, Arge et al.[4] developed a static cache-oblivious linear-space data structure for answering two-sided queries in $\mathcal{O}(\log_B N + T/B)$. Using a simple construction method, we can also obtain a static cache-oblivious data structure for four-sided queries, with the query running time $\mathcal{O}(\log_B N + T/B)$ and $\mathcal{O}(\log_2^2 N)$ space usage. This structure doesn't have any assumption about B and it's much simpler. In 2009, Afshani et al.[1] proved, that for optimal query bound, for three-sided range reporting, the structure has to use $\Omega(N(\log\log N)^\varepsilon)$ space, where $\varepsilon > 0$.

2. CACHE-OBLIVIOUS DATA STRUCTURE

For the cache-oblivious data structure we chose the data structure from [4], as it's the simplest and doesn't have assumptions about B, and has the same running time. We first describe the data structure for two-sided queries and how to construct it. After that we describe the construction of the data structure for four-sided queries.

2.1 Data structure for two-sided queries

The structure consists of two parts: a sequence $\mathcal{L}$ of length $\mathcal{O}(N)$, called the layout, which stores the points of S, with the possibility of duplicate entries; and a balanced binary search tree $\mathcal{Y}$ on a subset of the y-coordinates of the points in S, stored implicitly in the van-Emde-Boas layout. Each leaf in the tree stores a pointer to an element of $\mathcal{L}$. To answer a query $(-\infty, x] \times [y, \infty)$, a search for y in $\mathcal{Y}$ is done and then the pointer is followed into $\mathcal{L}$. Then a scan forward over $\mathcal{L}$ is performed until an x-coordinate greater than x is found.

The key point of the data structure is the way the points are stored in the layout $\mathcal{L}$. Assume that we are prepared to scan through αT points for $\alpha > 1$, when answering a query with output T. We will call it a dense if we scan at most αT points, and sparse otherwise. Consider the minimum y-coordinate y_1 such that there exists a sparse query $(-\infty, x] \times [y_1, \infty)$. Let S_0 be a sequence sorted by the x-coordinate, with y coosrdinate less than y_1. As there are no sparse queries with the y-coordinate less than y_1, we can answer those queries efficiently. As we repeat the step with the remaining points, we get a concatenation of sequences $S_0, S_1, ..., S_k$ of points, with which we can answer all the queries efficiently. The problem with this approach is that the space can be more than linear, since the worst case size is $\Theta(N)$.

To reduce the size of the layout we need to store the sequences in such a way, that every sequence S_i is identical with S_{i+1} having a suffix valuse as large as possible. We take a point with the minimum y-coordinate, such that there exists a sparse query $(-\infty, x_i] \times [y_i, \infty)$. Instead of storing all the points with y-coordinate less than y_i, we store only the points that has the x-coordinate less than x_i too. With this improvement it can be proved that the size of the layout $\mathcal{L}$ is linear:

Lemma 1. [4]Layout $\mathcal{L}$ uses at most $\frac{\alpha}{\alpha-1}N = \mathcal{O}(N)$ space.

To answer the two-sided queries we create, in addition to the leayout $\mathcal{L}$, a binary search tree $\mathcal{Y}$ over the y-coordinates we used to construct $\mathcal{L}$, and store the whole tree using the van-Emde-Boas layout. Each leaf stores a pointer to the start of the sequence it produced in $\mathcal{L}$.With both, the tree $\mathcal{Y}$ and the layout $\mathcal{L}$, we can answer two-sided range queries in $\mathcal{O}(\log_B NT/B)$.

Lemma 2. [4]The layout $\mathcal{L}$ and search tree $\mathcal{Y}$ can be used to answer any two-sided range query in $\mathcal{O}(\log_B NT/B)$ memory transfers, where T is the number of reported points.

2.1.1 *The construction*

Arge et al.[4] presented a contruction method for the layout. The basic idea is to store all the points, sorted by the x-coordinate, in the leafs of a balanced binary search tree, stored implicitly using the van-Emde-Boas layout. All the internal nodes carry additional information about the actual points in the subtree. Then using the sweep line approach; starting from $-\infty$, we sweep a horizontal line upwards across the plane. Each time the sweep line is at point p, we traverse the tree leaf-to-root and update all the nodes on the path. After this step, we have to check in the root of the tree, if there exists a sparse query. If it does, we have to traverse a root-to-leaf path (specified by the information in the internal nodes), to find the point, we use to create the tree $\mathcal{Y}$. The whole process was proved[4] to take $\mathcal{O}(N \log_B N)$ memory transfers.

2.2 Data structure for four-sided queries

We first explain how to construct a data structure for three-sided queries, as it is a step to construct the data structure for four-sided queries. Our three-sided structure consists of a balanced binary tree $\mathcal{T}$ with the points in S stored at the leaves, sorted by their x-coordinates. $\mathcal{T}$ is laid out in memory using the van-Emde-Boas layout, so that a root-to-leaf path can be traversed cache-obliviously in $\mathcal{O}(\log_B N)$ memory transfers. For each internal node v in $\mathcal{T}$, let S_v be the points of S stored in the subtree rooted at v. We store the points in S_v in two secondary structures, L_v and R_v, associated with v. L_v is a structure for answering two-sided queries of the form $(\infty, x] \times [y, \infty)$ (with the x-opening to the left); R_v is a structure for answering two-sided queries of the form $[x, \infty) \times [y, \infty)$ (with the x-opening to the right).

Each point is stored in two linear space structures on each of the $\mathcal{O}$ levels of the tree $\mathcal{T}$, the structure uses $\mathcal{O}(N \log_2 N)$ space. To answer the query $[x_l, x_r] \times [y_b, \infty)$, we take the first node v such that x_l is contained in the subtree rooted at he left child l, and x_r is contained in the subtree rooted at the right child r. Then we query R_l with the query range $[x_l, \infty) \times [y_b, \infty)$ and L_r with the query range $(infty, x_r] \times [y_b, \infty)$.

To construct the data structure for four-sided queries, we need to apply the same method again. We store all the points, this time sorted by the y-coordinate, in a balanced binary tree, stored using the van-Emde-Boas layout. For each internal node v, let S_v represent the points from S stored in the subtree rooted in v. We store the points in S_v in two secondary structures U_v and B_v, associated with v. U_v is a structure for answering a three-sided query of the form $[x_l, x_r] \times [y, \infty)$; B_v is a structure for answering a three-sided query of the form $[x_l, x_r] \times [y, -\infty)$. This construction adds another factor of $\mathcal{O}(log_2 N)$ to the space complexity, to get the final $\mathcal{O}(N \log_2^2 N)$ space complexity of the data structure for four-sided queries.

3. RESULTS

As we implemented the mentioned data structures, we noticed that the construction algorithm rely heavily on root-

Table 1: Comparison between the cache-oblivious data structure (C-O) and the quadtree (QT).

	Construction time		Query time		Size	
Number of points	C-O	QT	C-O	QT	C-O	QT
100.000	30 s	0,06 s	5 ms	43 ms	~1 GB	2MB
200.000	69 s	0,12 s	28 ms	90 ms	~2,5 GB	4MB
1.000.000	7,5 min	0,7 s	132 ms	453 ms	~7 GB	24MB
2.000.000	17 min	1.5 s	413 ms	951 ms	~16 GB	50MB
10.000.000	~2 h	22 s	2817 ms	6259 ms	~80 GB	300MB
100.000.000	~8 h	115 s	32523 ms	341282 ms	~950 GB	3,5GB

to-leaf and leaf-to-root traversals. Each computation to find the index of the parent/child of a specific node takes $\mathcal{O}(\log_2 N)$ time, so we decided to precompute the indexes for all the topologies of the trees, which drastically reduced the construction time.

We decided to compare the cache-oblivious data structure with a quadtree, probably the most used data structure for range queries in the RAM model. We compared the query time, construction time and the size of the data structure. We tested on problems of different sizes, varying from 100.000 to 100.000.000 points. All the coordinates were random 32-bits decimal numbers. All the queries were of type $(-\infty, \infty) \times (-\infty, \infty)$, so all the points had to be returned. All the tests were run five times and the average results are shown in Table 1. All the tests were run on a Core 2 Duo @ 2.53 GHz and 4GB of RAM.

From the results of the tests, it can be seen that the cache-oblivious is much faster in answering the queries. The shaded cells in Table 1 show, that the data structure for that specific size was already bigger than the available main memory, so swapping occured. Note, that because of high locality for the cache-oblivious data structure, swapping didn't affect it as much as it affected the quadtree.

On the other side, the size of the whole cache-oblivious data structure is much bigger than the quadtree. The main reason is that even if the key ingredient of the cache-oblivious data structure (the two-sided data structure) has linear space complexity, we need a lot of them ($\mathcal{O}(\log_2^2 N)$ exactly). When N is getting bigger, this factor is not negligible.

The difference in the construction time is probably of the least importance, as the cache-oblivious data structure is a static data structure, so it can be precomputed. Despite that, it is interesting to notice, that 2/3 of the construction time is sorting the points, as the four-sided data structure needs the points sorted by their y-coordinate, the three-sided needs them sorted by their x-coordinate. So for every three-sided data structure we need to sort all the points. The same happens for every two-sided data structure, as it needs them sorted by their x coordinate.

4. CONCLUSIONS

This is the first imeplementation of this data structure to the author's knowledge. It can be seen from the results, that the cache-oblivious data structure, once constructed, is not affected by the swapping of blocks made by the operating system, as the theory suggested. On the other side, to achieve that, the implemented solution takes a lot more space. So there is actually a trade-off between the speed of the queries and the space.

5. REFERENCES

[1] P. Afshani, C. Hamilton, and N. Zeh. Cache-oblivious range reporting with optimal queries requires superlinear space. In *SCG '09: Proceedings of the 25th annual symposium on Computational geometry*. ACM Request Permissions, June 2009.

[2] P. K. Agarwal, L. Arge, A. Danner, and B. Holland-Minkley. Cache-oblivious data structures for orthogonal range searching. pages 237–245, 2003.

[3] A. Aggarwal and J. Vitter. The input/output complexity of sorting and related problems. *Communications of the ACM*, 31(9):1116–1127, 1988.

[4] L. Arge and N. Zeh. Simple and semi-dynamic structures for cache-oblivious planar orthogonal range searching. In *SCG '06: Proceedings of the twenty-second annual symposium on Computational geometry*. ACM Request Permissions, June 2006.

[5] M. A. Bender, E. D. Demaine, and M. Farach-Colton. Cache-oblivious B-trees. *SIAM J. Comput.*, 35(2):341–358, 2005.

[6] M. Frigo, C. E. Leiserson, H. Prokop, and S. Ramachandran. Cache-Oblivious Algorithms. In *FOCS '99: Proceedings of the 40th Annual Symposium on Foundations of Computer Science*. IEEE Computer Society, Oct. 1999.

[7] H. Prokop. Cache-oblivious algorithms. *Master thesis, Massachusetts Institute of Technology, Cambridge.*

A new method for transforming algorithm into VHDL by starting from a Haskell functional language description

Gergely Suba
Budapest University of Technology and Economics
2 Magyar tudósok körútja
Budapest, Hungary, 1117
sugergo@iit.bme.hu

Prof. Dr. Péter Arató[*]
Budapest University of Technology and Economics
2 Magyar tudósok körútja
Budapest, Hungary, 1117
arato@iit.bme.hu

ABSTRACT

In the field of computer engineering, there are a lot of problems that are too time-consuming like biological or physical calculations and that's why they have to be implemented using special hardware structures. Usually, these hardware structures are described in HDL (Hardware Description Language). Developing in HDL languages is not as efficient as it would be in case of software languages due to its low level structures.

The aim of this work is to implement and test a method (a compiler program), where the starting point is a code written in the functional language Haskell and the output is the same algorithm in VHDL (a kind of HDL) language. The main advantage of the novel method presented in this paper is that it generates a synthesizable VHDL description from Haskell code automatically for FPGA implementation.

The method introduced in this paper can solve two separate problems: 1) running algorithms effectively in FPGA, 2) development of digital hardware implementing a specified function. We demonstrate the efficiency of the method through practical examples like a part of the MP3 decoding algorithm.

Categories and Subject Descriptors

C.3 [**Special Purpose and Application-based Systems**]: Real-time and embedded systems; D.2.2 [**Software Engineering**]: Design Tools and Techniques

Keywords

HLS, FPGA, Haskell, VHDL, dataflow graph

[*]Prof. Dr. Péter Arató is the supervisor.
Department of Control Engineering and Information Technology, Faculty of Electrical Engineering and Informatics

1. INTRODUCTION

Although the speed of the conventional processors is growing continually, there are a lot of problems that are too time-consuming like biological or physical computing and that's why they have to be implemented in a more efficient hardware structure and sometimes in special hardware. Usually, these hardware structures are described in HDL (Hardware Description Language). These languages are not so well-known by the mathematicians and software engineers, they usually write the algorithms in general-purpose programming languages. There is a great difference between using HDL or software languages, and the chance to transform these two representations into each other is rather small. To overcome this gap, there is plenty of research that are very important parts of the so called System Level Synthesis.

Hardware synthesis starting with a functional software language has some advantages compared to a hardware description language. Besides, modifying the hardware when the algorithm changes is easier, because a code written in functional language can be adapted more directly than one written in a hardware description language. Transforming an algorithm to HDL can be easily automated by the novel compiler program introduced in this paper. The program code written in functional language can be run in PC, therefore it can be tested easily. In contrast to this, the hardware description languages require a complex simulation for testing, where the inputs and outputs are handled as digital signals, which is an additional complication.

Based on the above arguments, the aim of this work is to implement and test a method (a compiler program), where the starting point is a code written in a functional language and the output is the same algorithm in HDL language. Functional languages have benefits in case of digital signal processing and it fits the capabilities of the hardware world better than the imperative ones. I focus on the Haskell [9] functional language, which is being developed dynamically, and it supports the research work well.

The main advantage of the method presented in this paper is that it generates the synthesizable VHDL [6] description from Haskell code automatically for FPGA implementation. Even a pipeline optimization tool can be involved by the procedure such as the high level synthesis tool PIPE, developed at the Department of Control Engineering and Information Technology in BME.

We demonstrate the efficiency of the method through practical examples like a PID controller and a part of the MP3 decoding algorithm.

For the method, we defined the following requirements:

1. compile Haskell to VHDL code automatically
2. notify the user about the error details in case the code cannot be compiled
3. make the compiler program modular, so that its parts can be used separately
4. ensure that pipeline optimization methods can be included if necessary
5. the operation set should be dynamically extendable, without modifying the compiler program

Among some functional languages (Lava [3], μFP [12], Ruby [5], SASL [4]) that can use to generate HDL code, the most similar project is the CλasH (CAES Language for Synchronous Hardware) [2, 7], nevertheless it meets only the first two requirements from the above list. However, the purpose of CλasH is different from the method introduced in this paper. Although, CλasH is based on Haskell syntax, it is a hardware description language, in contrast to our method where the purpose is a language without hardware concepts. CλasH also has some disadvantages compared with the method in this paper. It has no pipeline optimization stage, and it can not include external optimization methods, as it doesn't use dataflow graphs as intermediate representation. The operation set is constant, thus new operations can not be added dynamically.

The method and the compiler program introduced in this paper meets Requirement 1, as it compiles the code automatically. In our method the GHC frontend is used to preprocess the source code that includes the parser, lexer and the generation of the abstract syntax tree. Therefore, reporting of the compiler errors (Requirement 2) is ensured by GHC. Our method is based on modular structure: it is divided into frontend and backend parts, and these parts consist of inner stages as it will be detailed later (Requirement 3). Between the frontend and backend, the interface is a dataflow graph, which is an intermediate representation of the algorithm. Most of the pipeline optimization systems are based on dataflow graphs, therefore these stages can be included in our system (Requirement 4). An external database is used as operation set, therefore expanding the set of operations dynamically is simple (Requirement 5).

2. THE PROPOSED COMPILER ARCHITECTURE

In this section the architecture of our Haskell to VHDL compiler system will be introduced.

The structure of the compiler method is shown in Figure 1. As it can be seen, the structure is divided into frontend and backend parts. The first is the Haskell-EOG compiler, which produces the elementary operation graph (EOG) [1] as intermediate dataflow representation. EOG is a kind of well

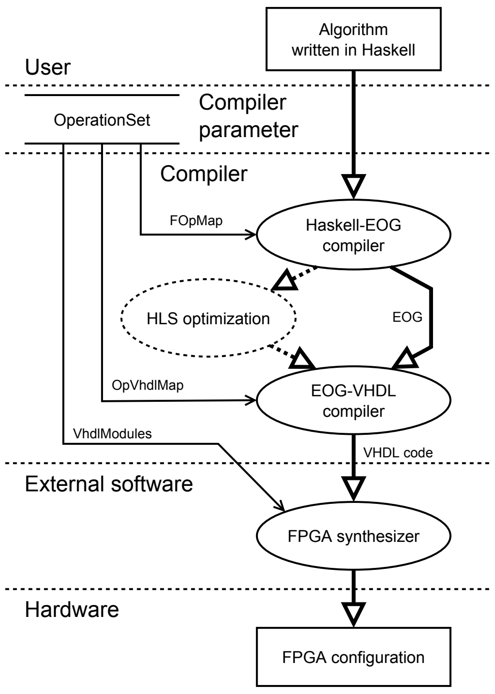

Figure 1: The structure of the compiler method

known homogeneous synchronous dataflow (HSDF) [8, 11] graph, where the vertices are the operations and the edges are dataflow links. In this representation, HLS optimization stage can be included, such as the HLS system PIPE [1]. After the optional optimization, the EOG-VHDL compiler as backend part will produce the expected VHDL output. From VHDL, an external FPGA development software can synthesize the FPGA bitstream, which can result in a real FPGA application.

2.1 Haskell-EOG compiler

The frontend of our compiler takes the source code written in Haskell, and produces the EOG via the successive stages introduced in this section.

Stage 1. Produce the AST Core
The first steps are to parse and analyze the code, and produce the abstract syntax tree (AST). These steps are made by the GHC frontend, which produces the GHC Core [13] tree. This Core tree mostly consist of the following type of nodes: Lit (constant literal), Var (using of variable), App (lambda application, a.k.a. function calling), Lam (lambda abstraction, a.k.a. function body) and Case (branching). In the further steps of the frontend pipeline these nodes have to be converted to EOG nodes and edges by successive graph transformations.

Stage 2. Eliminate of the branching
The second step is to eliminate all of the branching (Case) nodes from the Core. During the elimination every branch is to be converted to a special function calling (App) node, which performs the branching as a black box. This step is a so-called Core2Core transformation, as its input and output are Core trees with the same structure.

Stage 3. Produce the operation dependency tree
The operation dependency tree (ODT, here which is an intermediate data structure between the Core and the EOG representation) consists of the operation nodes and the dependency edges between the operations. Two types of nodes are defined in this tree: operation nodes and pointer nodes. An operation node has as many children as the number of its operands. A pointer node is always leaf and it can be considered as a reference to an operation node. If an operation node has a pointer node child, it will mean the same function as its child would be the operation node referenced by the given pointer node.

Producing the ODT is the largest part of the Haskell-EOG compiler. The main concept is supercompilation [10], where the idea is to travel the Core tree in the same order as the CPU would process the statements of this functional code. During this graph traversal, the following conversions are performed to produce the ODT (as the output tree) from the Core description:

- **Var** is transformed to **pointer node** in ODT
- **Lit** is transformed to **constant operation node** in ODT
- **App** (if it is elementary) is transformed to **elementary operation** in ODT
- **App** (if it is complex): the body of the function must be inlined, and the nodes contained by the inlined tree has to be processed as every other node
- **Lam** can be eliminated without restriction
- **Case** nodes have already been eliminated in Stage 2

Stage 4. Produce the EOG
The final step in the Haskell-EOG compiler is to transform the ODT to elementary operation graph, which is a simpler process, than the previous stages were. Every operation node in ODT will also be operation in EOG, and every dataflow link will also be link in EOG. The difference is that the pointer nodes must be resolved, and it has to be substituted by the operation node referenced by the pointer node. In practice, if the end of a link is a pointer node, this end has to be replaced by the referenced operation node. After this transformation, the dataflow structure is obtained, not necessarily being a tree anymore.

2.2 EOG-VHDL compiler

The architecture of the EOG-VHDL compiler, which is the backend of the whole method and system introduced in this paper, can be seen in Figure 2. First, the EOG goes through a preprocess task (**Preprocess EOG** in Figure 2), which collects all of the information about the EOG nodes, then saves it into the store **Operations**. Another task (**Make ModuleList** in Figure 2) creates the store **ModuleList**, which will contain all of the necessary information about the modules used in the input EOG. These modules have to exist in the external **OperationSet**, because the Make ModuleList task loads the information from that for each operation.

From the two stores created before, the produce tasks will generate the parts of the VHDL output. All of the used

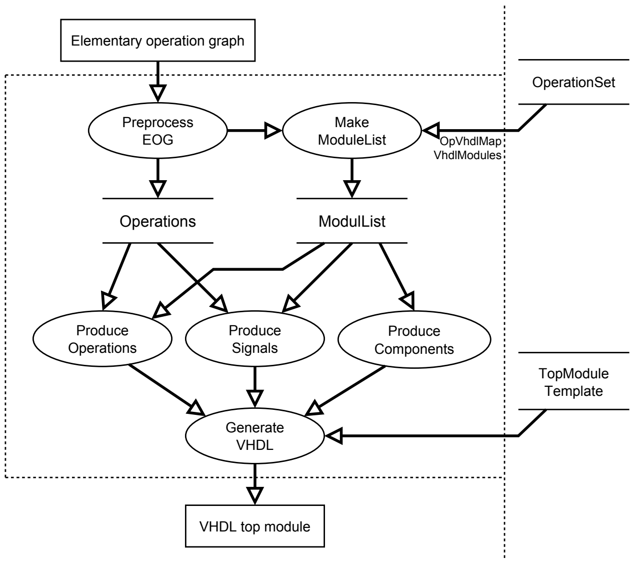

Figure 2: EOG-VHDL compiler

operations will result in component declarations in VHDL (it is generated by the **Produce Components** task). Each EOG operation will result in component instantiation and each dataflow link will result in signal declarations, which interconnect the component instantiations. These parts are generated by the **Produce Operations** and **Produce Signals** tasks in Figure 2.

Finally, the **Generate VHDL** task creates the output VHDL (the top module), from the produced VHDL parts discussed before.

The EOG-VHDL compiler produces a single, structural VHDL file, which contains only certain types of language structures: component and signal declarations and component instantiations. The behavior of the components are defined in VHDL files separately component-by-component (these files are stored in the **OperationSet**).

3. MULTI-RATE EXTENSION

The EOG, previously used as an interface between the compiler frontend and backend, is a single-rate dataflow graph, which is a huge restriction. EOG can only represent dataflow, where each operation is performed once during one restart of the system (in other words: it is performed once for each input of the system). Although, certain loops can be implemented in that case too, it is not an efficient way due to the increasing number of operation nodes. (for example, if a loop iterates between 1 and 5, the operations of the loop body have to be replicated 5 times) For overcome this restriction, a modified EOG is introduced, where groups of operations can be performed in different number of times during one restart period.

The extended version of EOG introduced in this section is called multi-rate EOG (MR-EOG). In MR-EOG, groups of operations (blocks) can be defined, where each block contains one or more operations and optionally other block(s). In this way the blocks are built up in a hierarchy, and the top of the hierarchy is a block denoted by TOP in further.

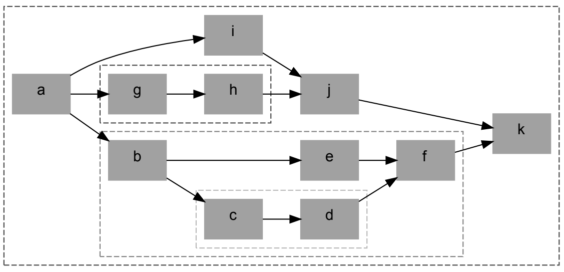

Figure 3: MR-EOG

An example can be seen in Figure 3. Here the block TOP contains the operations a, i, j and k, and two inner blocks. One of that contains the operations g and h, and the other contains b, e and f, and its inner block contains the operations c and d.

For the new multi-rate representation, there is an important purpose: ensure that the general algorithms performed in dataflow graphs (such as pipeline optimization algorithms) can also be used in MR-EOG. The solution is running the algorithms recursively, starting with the innermost block. Since the innermost block does not contain inner blocks, the methods developed for a single-rate dataflow graph can be performed without restriction. If the algorithm finish with a given block, the block has to be substituted by a virtual operation. The execution time of this operation will be the same as the execution time of the whole block was.

When a block contains only operations (including virtual operations), the same algorithm can be run on it. In the end, even the block TOP is performed, and the algorithm is finished for the whole MR-EOG hierarchy.

4. SOME CHARACTERISTIC TASKS FOR TESTING

We implemented the compiler program based on the method introduced above in Haskell language. For testing the functionality, two characteristic tasks (written also in Haskell) were chosen. The first one is a PID controller, which is a single-rate algorithm, therefore the method written in Section 2 can even compile this code to generate the desired VHDL. The second one is the synthesis filter bank (SFB) part of the MP3 decoder algorithm. Since it contains several loops, it can only be represented by multi-rate types of dataflow graphs efficiently. The multi-rate extension of the method detailed in Section 3 was used to compile this task.

A brief summary of the functional tests are shown in Table 1. Row 1 shows the characteristic numbers of the PID controller, while Row 2 shows the same for the MP3 SFB. The subtasks of the MP3 SFB can be seen in Row 3, 4 and 5 separately.

5. CONCLUSIONS

A compiler program is developed on base of the novel method introduced in this paper. The method meets the requirements that was set before: 1) it generate the VHDL from

	Program	**Num. of operations**	**Num. of links**	**Num. of blocks**
1	PID controller	29	36	0
2	MP3 SFB	93	133	6
3	MP3 ssum	19	25	1
4	MP3 uconv	32	45	0
5	MP3 usum	11	16	1

Table 1: Summary of the compiler tests

Haskell code automatically, 2) it reports in case of any compiler errors, 3) the structure of the compiler is modular (it has a frontend and a backend part), thus the parts could be used separately in other projects. 4) external pipeline optimization stage can be included, since it has a dataflow graph as intermediate representation. 5) the operation set is able to be extended, because they are from an external database separated from the compiler program itself.

The method, including its extended variant for multi-rate tasks, was tested with practical applications, such as a PID controller and an MP3 synthesis filter bank algorithm.

6. REFERENCES

[1] Peter Arato, Visegrady Tamas, and Istvan Jankovits. *High Level Synthesis of Pipelined Datapaths.* John Wiley & Sons, Inc., New York, NY, USA, 2001.

[2] C. P. R. Baaij. Cλash: From haskell to hardware. Master's thesis, Univ. of Twente, December 2009.

[3] Per Bjesse, Koen Claessen, Mary Sheeran, and Satnam Singh. Lava: hardware design in haskell. *SIGPLAN Not.*, 34:174–184, September 1998.

[4] Simon Frankau. Hardware synthesis from a stream-processing functional language, 2004.

[5] Shaori Guo and Wayne Luk. Compiling ruby into fpgas. In *Proceedings of the 5th International Workshop on Field-Programmable Logic and Applications*, FPL '95, pages 188–197, London, UK, 1995. Springer-Verlag.

[6] ISO/IEC. Ieee 1076-2008: Ieee standard vhdl language reference manual. Technical report, Institute of Electrical and Electronics Engineers, Computer Society, 26. January 2009.

[7] M. Kooijman. Haskell as a higher order structural hardware description language, December 2009.

[8] E.A. Lee and D.G. Messerschmitt. Synchronous data flow. *Proceedings of the IEEE*, 75(9):1235 – 1245, sept. 1987.

[9] Simon Marlow. Haskell 2010 language report, 2010.

[10] Neil Mitchell. Rethinking supercompilation. *SIGPLAN Not.*, 45:309–320, September 2010.

[11] K.K. Parhi. Algorithm transformation techniques for concurrent processors. *Proceedings of the IEEE*, 77(12):1879 –1895, dec 1989.

[12] Mary Sheeran. ufp, an algebraic vlsi design language - phd thesis, 1983.

[13] Andrew Tolmach. An external representation for the ghc core language, 2009.

A system for parallel execution of data-flow graphs

Andrej Bukošek
FRI, University of Ljubljana
Večna pot 113,
1000 Ljubljana, Slovenia
andrej.bukosek@gmail.com

ABSTRACT

This paper describes the system we developed for performing arbitrary operations on data in parallel using a data-flow graph [3].

Each operation is implemented in a dynamically loadable module or using a domain-specific language, which was designed specifically for this purpose. We also implemented a compiler for this language. Our domain-specific language is functional and strongly typed. We designed its type system to be modular. Every data type is implemented in an external dynamically loadable module, which the compiler loads during its initialisation. Each module contains functions for generating an intermediate representation for the LLVM system, which optimises it and translates it into machine code.

As an example usage of our system, we developed operations for image manipulation, compositing, and rendering of 3D scenes. Such a set of operations is commonly used in the film industry for the creation of special effects.

We also implemented a renderer based on the path tracing algorithm, which creates an image from the description of a 3D scene. This method is based on a physically-correct simulation of light bouncing around the scene.

Categories and Subject Descriptors

D.1.3 [**Programming Techniques**]: Parallel programming; D.3.2 [**Programming Languages**]: Functional languages; D.3.2 [**Programming Languages**]: Specialized application languages; D.3.4 [**Programming Languages**]: Compilers; E.1 [**Data Structures**]: Graphs; I.3.4 [**Computer Graphics**]: Graphics packages; I.3.7 [**Computer Graphics**]: Raytracing, Animation; I.4.0 [**Image Processing and Computer Vision**]: Image processing software

General Terms

Design, Languages, Algorithms

Keywords

data-flow graphs, parallelisation, domain-specific language, compositing, rendering, computer graphics

Supervisors

Dr. Andrej Brodnik, Dr. Borut Robič

1. INTRODUCTION

Computer generation and manipulation of images is becoming more and more prevalent in the film industry [2]. From compositing of complex rendered scenes with live actors to simple colour grading, such tasks are accomplished with specialised software that allows its users to create a data-flow graph of operations, which are applied to the footage.

Our goal was to create a generalised and more flexible software package for performing arbitrary operations on any data using a data-flow approach.

The core functionality of the system is the parallel execution of data-flow graphs. Each node in a graph represents an operation on data, which passes through its edges.

To ease creation of new graph operations, we developed a domain-specific language. Operations can also be implemented in any language that supports compiling code into shared objects with a standard C ABI (these plugins are then loaded at runtime). Our domain-specific language is functional and strongly typed. An interesting feature is its type system. Each type is implemented in an external shared object (plugin), which enables users to add custom types and parsers for constant values.

As a practical usage of this system, we also implemented a variety of operations and types. A particularly interesting operation is a 3D scene renderer, which employs the Monte Carlo path tracing algorithm to generate physically-correct images of 3D scenes [4, 5].

2. SYSTEM ARCHITECTURE

Figure 1 shows an overview of the system components.

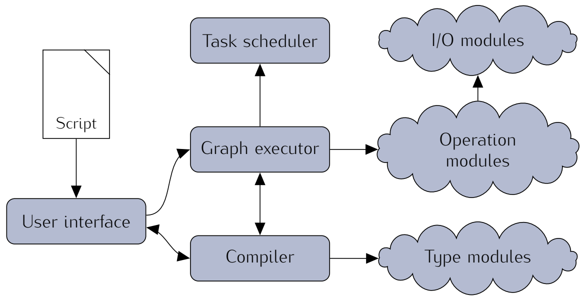

Figure 1: System architecture.

In the most common use case, the system accepts a script written in our domain-specific language. This script describes the data-flow graph (nodes and connections) using standard function calls (this could be achieved in any other language if the system was converted into a library).

First, the system loads and initialises I/O and operation modules. This is followed by initialisation of the compiler, which also loads the type modules. Then the script is compiled into machine code in memory and executed. The script forms the data-flow graph to be executed in parallel. The graph is executed after script execution finishes.

The main advantage of having external modules for I/O, operations, and types is greater flexibility, as the main program doesn't need to be rewritten or recompiled to support new file formats, operations, and data types.

The subsequent sections of this paper contain more detailed explanations of the various subsystems.

3. OUR DOMAIN-SPECIFIC LANGUAGE

The main purpose of the language we created is to make developing new graph operations easier, especially for people who aren't professional programmers. Its secondary purpose is to provide a language in which users can describe the data-flow graph, however, this functionality could easily be accomplished using an existing language.

Our language is functional, strongly typed and compiled into machine code for greater performance.

Optimisations and machine code generation are provided by the LLVM framework (`http://www.llvm.org/`). The compiler we developed compiles source code into LLVM intermediate representation, on which we run various LLVM optimisation passes and finally generate native machine code for the architecture of the machine the program is running on. The code is compiled directly to memory, no temporary files are created during compilation.

3.1 Type system

All types in the language are implemented as external shared objects (plugins). Each plugin module implements functions for generating code for operators (e.g. adding two real numbers, multiplying a matrix with a vector, etc.) and parsing constants (optional). The code generation functions actually generate LLVM intermediate representation instructions.

Apart from operators, each module can also generate code for various library functions for that type (e.g. functions for dot and cross product in the case of a 3D vector type). One of the library functions for composite types (e.g. vectors and matrices) is also the constructor for that type. The constructor function allocates memory for the structure and sets its members to the values passed to the constructor.

One of the advantages of generating code for operators and functions (as opposed to simply implementing them and generating function calls instead) is that this enables further optimisations.

Each type module can also define a parser hook for constants. While looking for constants, the lexer gives full control to such hooks. This means that the syntax of constant values can be completely arbitrary. One could even include an interpreter for another programming language and use that to generate a constant by running a program.

3.2 Grammar

The grammar for our language in BNF [1]:

```
⟨identifierexpr⟩      ::= ⟨identifier⟩ | ⟨identifier⟩ '(' ⟨expression⟩* ')'
⟨constexpr⟩           ::= ⟨const⟩
⟨parenexpr⟩           ::= '(' ⟨expression⟩ ')'
⟨ifexpr⟩              ::= 'if' ⟨expression⟩ 'then' ⟨expression⟩
                          'else' ⟨expression⟩
⟨forexpr⟩             ::= 'for' ⟨type⟩ ⟨identifier⟩ '=' ⟨expression⟩ ','
                          ⟨expression⟩ [',' ⟨expression⟩] 'do' ⟨expression⟩
⟨withexpr⟩            ::= 'with' ⟨type⟩ ⟨identifier⟩ ['=' ⟨expression⟩]
                          (',' ⟨type⟩ ⟨identifier⟩ ['=' ⟨expression⟩])* 'do'
                          ⟨expression⟩
⟨primary⟩             ::= ⟨identifierexpr⟩
                        | ⟨constexpr⟩
                        | ⟨parenexpr⟩
                        | ⟨ifexpr⟩
                        | ⟨forexpr⟩
                        | ⟨withexpr⟩
⟨unary⟩               ::= ⟨primary⟩ | ⟨unop⟩ ⟨unary⟩
⟨binoprhs⟩            ::= (⟨op⟩ ⟨unary⟩)*
⟨expression⟩          ::= ⟨unary⟩ ⟨binoprhs⟩
⟨prototype⟩           ::= ⟨type⟩ ⟨identifier⟩ '(' [⟨type⟩ ⟨identifier⟩ [','
                          ⟨type⟩ ⟨identifier⟩]*] ')'
⟨func_definition⟩     ::= 'func' ⟨prototype⟩ ⟨expression⟩
⟨extern_definition⟩   ::= 'extern' ⟨prototype⟩
⟨program⟩             ::= (⟨func_definition⟩ | ⟨extern_definition⟩)*
```

The symbol ⟨*identifier*⟩ represents the name of a variable or function. It can contain alphanumeric characters and underscores, but cannot start with a digit.

⟨*const*⟩ represents a constant value, which can be parsed by any of the type modules. When encountering this symbol, the lexer goes through all the defined hooks in type modules until it finds one that accepts the constant.

⟨*type*⟩ represents a type keyword. Each type module defines a unique keyword for its type.

Comments in the language begin with a `#` sign and continue until the end of the line.

The precedence of operators is shown in table 1.

Operators	Precedence	
`;`	1	(binds weakest)
`=`	2	
`or`	6	
`xor`	7	
`and`	8	
`==`, `<>`	9	
`<`, `>`, `<=`, `>=`	10	
`+`, `-`	20	
`*`, `/`	40	(binds strongest)

Table 1: Precedence of binary operators.

Currently, only two unary operators are implemented — unary minus (`-`) and logical negation (`not`).

3.3 Code examples

All constructs in our language are functional expressions and can be combined in the same way as arithmetic expressions. The language is not whitespace sensitive.

Let's look at how we might calculate the n-th Fibonacci number in our language:

```
func int fib(int n) if n < 3 then 1 else fib(n-1) + fib(n-2)
```

An iterative version using real numbers would look like this:

```
func real fibr(real n)
    with real a = 1.0, real b = 1.0, real c do
        (for real i = 3.0, i < n do
            c = a + b;
            a = b;
            b = c);
        b  # this is the value that the function returns
```

Variables are declared with the keyword `with` and are valid in the expression following the keyword `do`. The `for` loop is enclosed in parentheses due to the priority of the `;` operator. This operator behaves similarly to `,` (comma) in C.

4. DATA-FLOW GRAPHS

Data-flow graphs consist of nodes, which represent operations, and edges, which facilitate the flow of data between nodes. In our implementation, the graphs are directed and acyclic, as this is powerful enough to represent all current use cases and simplifies parallelisation.

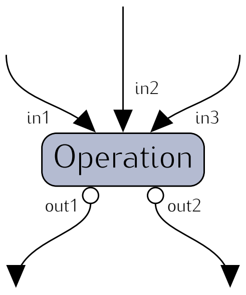

Figure 2: Node structure.

Each node can have input and output ports (see Figure 2). An input port is merely a pointer to an output port of another node. All input ports must be connected to something. Output ports also contain the data or results of the operation of its node. Output ports can remain unconnected.

Nodes without any input ports are usually generators of constants, while nodes without output ports usually save or display the results.

Each port has a data type associated with it. If the types of an input and an output port don't match, no connection between them can be made.

5. GRAPH CREATION

Graphs can be created using our domain-specific language. Every operation gets its own constructor, which returns a node handle and automatically adds the newly created node into the global data-flow graph. Nodes that output constants have an additional argument in their constructor. Individual nodes can be connected using the function `connect(node1, "outputName", node2, "inputName")`, which connects output port `outputName` on `node1` to input port `inputName` on `node2`.

To get a better idea of how this works, let's take a look at a real-world example:

```
func int main()
    with node s1, node s2, node s3, node s4,
         node s5, node c1, node r1, node r2,
         node li1, node li2, node ck1,
         node cb1, node si1
    do (
        # Create constant generators
        s1 = String("bg.png");
        s2 = String("fg.png");
        c1 = Color(color(0.0, 1.0, 0.0, 1.0));
        r1 = Real(0.05);
        r2 = Real(0.10);
        s3 = String("over");
        s4 = String("f");
        s5 = String("out.png");

        # Create operations
        li1 = LoadImage();
        li2 = LoadImage();
        ck1 = ChromaKey();
        cb1 = Combine();
        si1 = SaveImage();

        # Connect ports
        connect(s1,  "const",    li1, "fileName");
        connect(s2,  "const",    li2, "fileName");
        connect(c1,  "const",    ck1, "keyColor");
        connect(r1,  "const",    ck1, "tolNear");
        connect(r2,  "const",    ck1, "tolFar");
        connect(li2, "imageOut", ck1, "imageIn");
        connect(s3,  "const",    cb1, "mode");
        connect(s4,  "const",    cb1, "clipTo");
        connect(ck1, "imageOut", cb1, "fgImage");
        connect(li1, "imageOut", cb1, "bgImage");
        connect(cb1, "imageOut", si1, "imageIn");
        connect(s5,  "const",    si1, "fileName");
    );
    0
```

This script generates the graph shown in Figure 3.

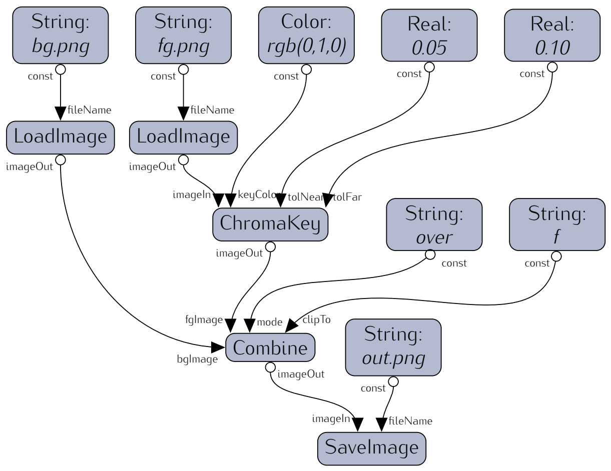

Figure 3: Graph example (image compositing).

The generated graph loads two images (`bg.png` and `fg.png`), performs chroma keying on the foreground image, overlays the result over the background image, and saves the final composite into `out.png`.

6. GRAPH EXECUTION

During execution, each operation gets a parameter, which tells it for which moment in time ($t \in \mathbb{N}_0$) it should perform that operation. This is useful in graphs that generate animations, in which case t represents the frame of the animation.

The currently implemented ways of executing a data-flow graph are:

- **single execution** — executes the graph only once for a given t
- **multiple execution** — executes the graph for many given t's
- **continuous execution** — executes the graph until interrupted by the user (t starts at 0 and rises monotonically).

To execute the graph, we convert operations within it into tasks for our batch job scheduler and set their priorities according to the dependencies from the graph.

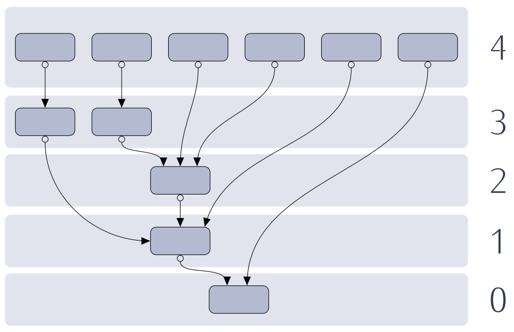

Figure 4: Decomposing the graph into levels.

As shown in Figure 4, we can decompose the graph into levels. Operations which share the same level are mutually independent and can be executed in parallel.

The parallel schedule is generated using a greedy approach, starting at the nodes with no outputs and working upward to satisfy dependencies.

The batch job scheduler is currently custom-made. It spawns as many threads as there are processor cores in the system. These threads then obtain work from a global priority queue, which contains the tasks. Threads with higher priority have to finish before those with a lower one can be run.

7. OPERATIONS

Implemented operations, sorted by area of usage:

- **compositing** [2] — CorrectGamma, Combine, Premultiply, Unpremultiply, ToneMap, ChromaKey, DiffKey, Resize, Crop, AffineTransform, Convolve2D
- **rendering** [4, 5] — Sphere, LoadTriangleMesh, Camera, LoadSpectrum, BRDFMaterial, LambertianBRDF, AshikhminBRDF, ApplyEmission, ApplyNormalMap, SceneGraphNode, AddChild, PathTrace
- **general** — LoadImage, SaveImage, LoadFrame, SaveFrame

Figure 5 shows an example image generated with our system. Rendering and compositing of elements in the picture was made entirely using our system.

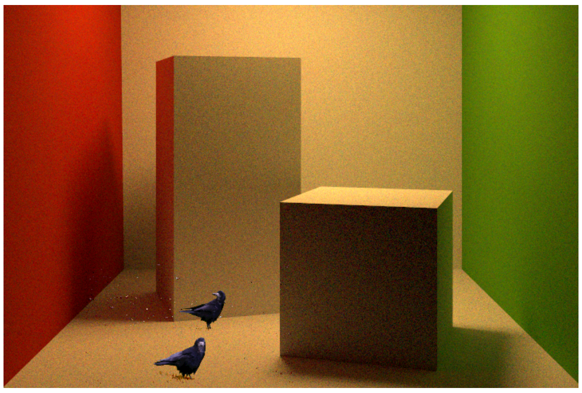

Figure 5: George, I think we're lost.

8. CONCLUSION

The main goal was to create a generalised and flexible software package for performing arbitrary operations on any data using a data-flow approach. To this end we developed a system for parallel execution of data-flow graphs with its own domain-specific language, created to facilitate the implementation of graph operations. As an example of the system's flexibility, we implemented various compositing, rendering, and general image processing operations [3].

Although the system is already in a usable state, many future improvements could be made. Implementing a graphical user interface for creating graphs would be most beneficial, as it would make the system more accessible and easier to use. The scheduler could be improved to handle distributed scheduling.

The system gets more useful as more operations are added. Possible areas for future development of operations include: digital signal processing, computer vision, data mining, and hardware control. Utilising operations from these areas, the system could collect data from sensors, analyse it, and visualise the results. It could also be used to control a robot or industrial machinery.

9. REFERENCES

[1] J. W. Backus, F. L. Bauer, J. Green, C. Katz, J. McCarthy, A. J. Perlis, H. Rutishauser, K. Samelson, B. Vauquois, J. H. Wegstein, A. van Wijngaarden, and M. Woodger. Revised report on the algorithm language ALGOL 60. *Communications of the ACM*, 6(1):1–17, Jan. 1963.

[2] R. Brinkmann. *The art and science of digital compositing.* Morgan Kaufmann Publishers Inc., 1999.

[3] A. Bukošek. *A system for parallel execution of data-flow graphs.* FRI, University of Ljubljana, 2013.

[4] M. Pharr and G. Humphreys. *Physically Based Rendering, Second Edition: From Theory To Implementation.* Morgan Kaufmann Publishers Inc., 2010.

[5] P. Shirley and R. K. Morley. *Realistic Ray Tracing, 2nd edition.* A. K. Peters, Ltd., 2003.

An Algorithmic Framework for Real-Time Rescheduling in Public Bus Transportation

Balázs Dávid
University of Szeged
davidb@jgypk.u-szeged.hu

János Balogh[*]
University of Szeged
balogh@jgypk.u-szeged.hu

ABSTRACT

In this paper, we describe an algorithmic framework for the vehicle rescheduling problem. This framework is based on problems arising in the operative planning of real-world public transportation companies.

Categories and Subject Descriptors

H.4 [[]: Information systems applications]; H.4.2 [[]: Types of Systems]: Decision support (e.g., MIS), Logistics

General Terms

Vehicle scheduling, Disruption management, Public transport

1. INTRODUCTION

Based on the available transportation data, the vehicle- and driver schedules of a transportation company are created in advance. A unique schedule is created for each day (or day-type) of the planning period in such a process. A daily schedule of a company is made up of vehicle and driver duties. A duty is considered as a series of tasks, which the corresponding vehicle and driver has to complete in the given order. The most important tasks of these duties are the timetabled trips, which come from the timetable of the company and have to be executed.

However, there are several difficulties that can arise while executing such a schedule in real-life: a problem with the vehicle itself, sickness of the driver, or other such things can make the pre-planned schedule infeasible. These unforeseen difficulties are called disruptions. If a disruption arises, a new feasible schedule has to be created, usually by rescheduling the old one.

Disruptions have to be managed almost immediately, which is usually the task of a company operator. Companies usually have a backup vehicle with which they address more complicated disruptions, but this solution is usually really expensive. Operators might not be able to solve complicated disruptions in short time, but a decision support system could provide them with helpful suggestions in such a case.

[*]supervisor

The goal of our paper is to introduce a solution framework for the rescheduling problem in public transportation, which could be used in aiding company operators responsible for managing disruptions. An important requirement for this framework is to guide the solution process regardless of the applied solution method. As the problem has to be solved in almost real time, one of the main requirements for such a system is to provide suggestions quickly. Optimization systems are already present for long-term planning [2, 11, 12], but these are all built around given solution methods. However, the only paper to our knowledge that deals with a system for bus rescheduling is the one by Li et al. [8].

Although there are published mathematical models for the problem [6, 10], these cannot be solved in short enough time. This leads to the design of efficient heuristic methods [6, 9], which are able to provide quick solutions. The system we present in this paper is designed to work with any kind of solution method, and gives results for the problem based on needs of the operators, which they can control through different parameters. The heuristic methods we use to demonstrate the system were published in [6].

The outline of our paper is the following: first, we give a quick overview of the rescheduling problem and disruption management. We present the basic regulations, and give a list of these that can be regarded in a flexible manner through parameters and penalizing. Based on these design thoughts, we introduce the suggested framework itself. Finally, we give some ideas of solution methods that we propose for the framework.

2. TRANSPORTATION SCHEDULING AND DISRUPTION MANAGEMENT

The daily schedule of a transportation company consists of several duties. Every duty is executed by a driver, and also has a corresponding vehicle. Each duty is a series of tasks that have to be carried out in the given order. The most common tasks are the ones corresponding to the trips of the timetable, and the so-called deadhead trips, which are responsible for moving the empty vehicle from one location to another. There can also be several different vehicle specific tasks (eg. parking, refueling) and driver specific tasks (eg. breaks, administration).

2.1 Disruption management

When a disruption happens, usually one of the vehicles in service becomes unavailable for a period of time. This leads to the pre-planned schedule becoming infeasible, as one or more of the tasks are not executed by a vehicle anymore. These trips are addressed from now on as disrupted trips. A vehicle schedule becomes infeasible if it contains such trips. Companies usually have a backup vehicle and driver ready, which are dedicated to such situation, but this solution might not be the best one.

Moreover, most real-life cases have a so-called multiple depot problem. There are several different depot locations and/or vehicle types, and every trip can only be serviced by vehicles belonging to a set of pre-determined depots. Because of this fact, our problem is NP-hard in the case of 2 or more depots. The vehicle rescheduling problem can be reduced to the vehicle scheduling problem, which was proven to be NP-hard by Bertossi et al. [1]. However, the problem itself should be solved as quickly as possible, and order should be restored with a new feasible solution.

2.2 Related work

To our knowledge, the literature regarding bus disruption management is scarce. Disruption management connected to different transportation fields has been researched for a longer period of time.

The earliest papers regarding disruption management were published about the airline industry. An overview paper by Clausen et al. can be read in [4, 3]. A bit younger field is disruption management is railway transportation. Some results regarding this area can be read in [7].

Both airline and railway disruption management differ from bus disruption management in their underlying structure. While buses are quite easy to move around with the help of deadhead trips between different geographical locations, the deadheading of airplanes is mainly prohibited by the high arising cost, and railway deadheading is subject to the underlying limited rail capacities.

As we have mentioned, the problem of bus rescheduling as defined above was only considered by Li et al. In their papers, they studied the single depot BRP. They give a quasi-assignment model and an auction algorithm for the problem in [9], and a network flow model is described in [10] which is solved with the help of Lagrangian relaxation. In [8], they also describe a possible decision-support system for this problem, which is illustrated with the help of a small real-world instance.

2.3 Real-life criteria

In a real-life application, there are several rules and constraints that make the problem more complicated. There are regulations that cannot be violated, while the violation of others should be penalized. As we mentioned earlier, we can define daily schedules of both vehicles and drivers, thus we classify our rules into two groups. In this subsection, we will give a list of the most important rules in both groups. Note, that these are only the most common regulations. Other ones might arise depending on a specific company or country.

- **Vehicle regulations**
  - *Vehicle depot and trip compatibility*: As we mentioned earlier, vehicles can be classified into depots. Trips can only be serviced by a fixed set of depots, and this should be respected when creating a new schedule to manage the disruption.
  - *Vehicle type and trip characteristics*: Similarly to depot compatibility, some trips can only be executed by vehicles of a certain type. For example, trips that are carried out between different cities, or have a long distance, must have a bus with special equipment (eg. air conditioning).
- **Driver regulations**
  - *Maximum driving time*: Each driver has a maximum daily driving time, which they can not exceed.
  - *Driver breaks*: After given time periods, drivers have to be assigned breaks. Moreover, these breaks have to be assigned at specific geographical locations.
  - *Maximum working time*: Similarly to driving time, the daily working time of drivers is also maximized. This does not equal driving time, as driver duties have other events as well, which do not require a vehicle (eg. administration).

Besides these regulations, some structural modifications should also be considered in the schedule. For an illustrative example, refer to Figure 1.

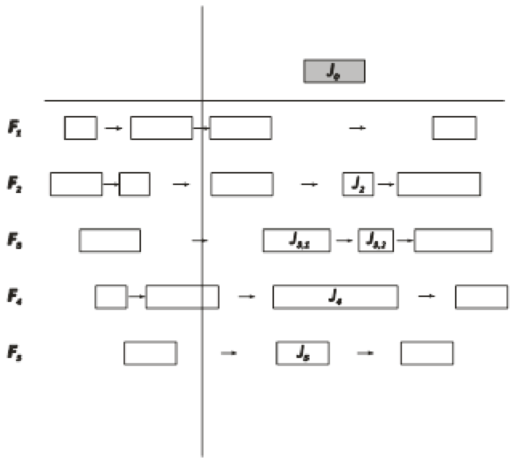

Figure 1: An illustrative example

On the above figure, a disrupted daily schedule can be seen, with a disruption represented by a vertical line. There is one disrupted trip J_0. Depending on some circumstances, we can give several solutions for the problem. Here are a few examples:

- If trip J_0 is compatible with duty F_1, and there is enough time to insert it to the available gap (together with any necessary deadhead trips), then we can solve the problem.
- If we want to insert J_0 to duty F_2, we have to remove trip J_2. There are several different places we can insert this newly removed trip. It might fit into the gaps in duties F_1 or F_5 (if compatible, and there is enough time for deadheads). We might also be able to insert them to duties F_3 or F_4. However this would mean we have to remove trip $J_{3,2}$ or J_4 respectively, and finding a new duty for them.
- Following the above logic, there are several different scenarios that utlizie removing trip from a duty, and inserting it to another one.
- We might delay trip $J_{3,2}$ in duty F_4. This can give us a big enough gap to insert our trip J_0 (if it is allowed by compatibility and deadheads).

As it can be seen from the above examples, there are some other constraints as well that are connected to the original duties or tasks (eg. modifying starting time, or removing a trip from its original duty).

A solution method for this problem need to know the extent to which it can violate the constraints we introduced above, and it also needs to know the hardness of the constraint. This information can be given easily with the use of parameters.

2.4 Parameters

In this subsection, we introduce parameters for the different rules and constraints given in the previous subsections. These parameters need to be considered by any algorithm solving the bus rescheduling problem:

- A binary parameter that allows the violation of depot-compatibilities. A penalty parameter for each violation of depot-compatibility is also needed.
- A binary parameter that allows the violation of vehicle type correspondence. A penalty parameter also has to be introduced for each violation of vehicle type correspondence.
- A binary parameter that allows the introduction of lateness. A penalty parameter also has to be introduced per 1 unit (minute) of lateness.
- A binary parameter that allows the movement of trips between feasible schedules. A penalty parameter is also needed that gives the cost of each such move.
- An integer parameter that limits the maximum amount of lateness which can introduced by the algorithms to the schedule.
- An integer parameter that limits the maximum amount of lateness an algorithm can introduce to a single event.
- An integer parameter that limits the maximum amount of lateness an algorithm can introduce to one duty.
- An integer parameter that gives the maximum number of feasible schedules which can be modified.
- A parameter that limits the maximum length of the newly introduced deadhead trips.
- A parameter, which gives the latest point in time, by the end of which the algorithms should not modify any more feasible schedules.
- A parameter on the number of suggestions (feasible solutions).
- A parameter on the maximum running time.

As it can be seen in the list of proposed parameters, there are none that correspond driver rules. Driver regulations are very strict, and most of them are defined by the EU, and cannot be violated by any means. Depot compatibility and vehicle type correspondence might also be strict, but we decided to let the operator of the system decide about their violation.

3. THE SOLUTION FRAMEWORK

In the previous sections, we presented our basic ideas behind the methodology for the problem. We described a framework that does not depend on the solution algorithm it executes, and can be controlled through a list of different parameters. In Figure 2, we give a layout of the different parts of the system.

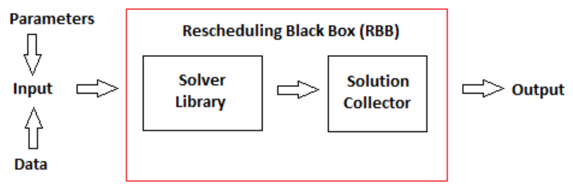

Figure 2: The structure of the framework

The input for the system consists of two parts. One part is the list of parameteres, that we described in the previous section. The other part is the problem data itself, containing the problem specific data tables. The type and structure of these tables of course can vary between different implementations of such a system, but it is important that it contains the disrupted schedule and the disrupted trips. For the remainder of the paper, we will refer to the pair of $\{disruptedschedule, setofdisruptedtrips\}$ as a $configuration$.

The input determines the starting configuration of our problem, which is a schedule that contains feasible duties, and a set which contains the disrupted trips, that are not executed currently by any vehicle. A configuration is supposed to be feasible, if its set of trips is empty, and all the duties in its schedule are feasible and do not violate any regulations or parameters.

Once all the input is read, it is then transferred to the main module of the system, which we call the Rescheduling Black Box (RBB). This is the part that carries out the solution process, and consists of two parts:

- The *Solver Library (SL)* manages the different solution methods that are built into the system. The system can have any number of implemented solution methods, and the desired method can be invoked by parameter setup. If there is a possibility to parallelize solution processes, multiple methods can also be executed at once.

- The output of a solution method is sent to the *Solution Collector (SC)*. This sub-module is responsible for managing feasible solutions. If more solution methods are running in parallel, all of them send their results to the SC. The SC then filters any duplicate solutions, and also gives an ordering of the remaining ones based their cost.

Once all solution algorithms finish their execution, or the desired maximum runtime is reached, the SC returns a number of feasible solutions. The desired number can be given by the operator as an input (number of suggestions), and their order is determined mainly by their cost, but they can also be filtered considering the different parameters (eg. ask for ones with no lateness, if possible). The operator receives this data, and can use these to decide on how to solve the arising problem.

4. A REAL LIFE APPLICATION

In this section, we briefly describe a real-life application of the above system. As we mentioned earlier, solution for the rescheduling problem has to be adequatly quick, as the order in transportation has to be restored as soon as possible. Because of this, the implemented heuristic methods must have a running time of at most a couple of seconds to be acceptable.

Also, because of the flexibility of the SC module, it is useful to provide solution methods which find multiple feasible solutions during their runtime. This gives the operator more suggestion options to choose from. A simple approach that we implemented in our system is a *naive search*, which basically finds all the possible trivial insertions in the starting configuration (if any), and inserts them to the solution collector. If there is any trivial insertion, it is highly likely that it will be the cheapest solution with regards to any of the parameters.

In one of our previous papers, we formulated a *mathematical model* for the bus rescheduling problem [6]. The size of the problem is big, and it cannot be solved efficiently even for smaller random instances. However, we also proposed two solution heuristics in the same paper. These methods have also been implemented to the system.

The first method is a recursive search, which uses the initial configuration as an input, and inserts a disrupted trip into one of the schedules in each step. If the disrupted trip cannot be inserted trivially, then other trips are moved from the schedule to the disrupted set. To avoid exponential explosion, we gave a depth limit to the recursive calls, and also have a function that cuts certain branches of the search tree. The recursive search finds a feasible solution, if the disrupted set is empty.

The second method is a local search heuristic (where a tabu list can also be applied effectively). This method creates a (probably infeasible) pseudo-schedule from the disrupted trips, without any regards to the regulations. In each iteration step of the search, either a move or a swap operation is executed. A trip is either moved from a schedule to another, or is swapped with events from another schedule. One important rule is that trips cannot be moved onto the pseudo-schedule. The local search finds a feasible solution, if the pseudo-schedule is empty.

Test results of the framework in real life have been promising. All instances have a short running time (they all finish under 1 minute even for bigger instances). It can also be seen, that these methods will find multiple feasible solution while exploring there solution space. Because of the SC in the system, the solutions can all be saved, and the ones that best suit the parameters of the operator can be chosen at the end.

Besides the real instances, we also tested our system on randomly generated data. In Table 1 and Table 2, we present the results of the above methods. The tables give the instance name, the number of original trips in the schedule, the number of the disrupted trips, and the running time of the recursive and local search methods on seconds. All instances in Table 1 are generated using the method in [5].

Table 1: Solution on random instances from [6].

Instance	Depots	Trips	Disrupted trips	Rec. (s)	Loc. (s)
random1	2	12	1	0.02	0.001
random2	4	100	1	0.05	0.004
random3	4	500	1	0.08	0.01
random4	4	800	1	0.08	0.05

We also present two real-life test results from the city of Szeged, Hungary. The smaller instance is only for a district of the city on a workday, while the bigger instance is that of a Saturday.

Table 2: Solution on real-life instances.

Instance	Depots	Trips	Disrupted trips	Rec. (s)	Loc. (s)
szeged_small	4	206	2	0.399	0.548
szeged_sat	4	1983	1	0.037	0.059

As it can be seen from the above tables, all test results have a good running time for both of our heuristics. The methods also returned several suggestions for the test cases.

5. CONCLUSIONS AND FUTURE WORK

The long-term plans of public transportation companies are disrupted on a daily basis. These disruptions have to be addressed as soon as possible. In this paper, we introduced a decision support framework for the rescheduling problem in public bus transportation.

We analyzed both vehicle and driver regulations for the daily schedules, and determined the parameters that have to be considered during the rescheduling process. The framework we described is independent of the solution method, and is also able to run different solution methods in parallel. It can also store and return multiple solutions, which gives the flexibility to the company operator to choose according to his or her needs.

6. ACKNOWLEDGMENTS

This work was partially supported by the European Union and co-funded by the European Social Fund through project HPC (grant no.: TÁMOP-4.2.2.C-11/1/KONV-2012-0010).

7. REFERENCES

[1] A. A. Bertossi, P. Carraresi, and G. G. Gallo. On some matching problems arising in vehicle scheduling models. *Networks*, 17(3):271–281, 1987.

[2] J. Békési, A. Brodnik, D. Pash, and M. Krész. An integrated framework for bus logistic management: case studies. *Logistik Management*, pages 389–411, 2009.

[3] J. Clausen, A. Larsen, J. J. Larsen, and N. J. Rezanova. Disruption management in the airline industry-concepts, models and methods. *Computers & Operations Research*, 37(5):809–821, 2010.

[4] J. Clausen, A. Larsen, and J. Larsen. Disruption management in the airline industry - concepts, models and methods. Technical report, Informatics and Mathematical Modelling, Technical University of Denmark, DTU, 2005.

[5] B. Dávid and M. Krész. Application oriented variable fixing methods for the multiple depot vehicle scheduling problem. *Acta Cybernetica*, 21(1):53–73, 2013.

[6] B. Dávid and M. Krész. A model and fast heuristics for the multiple depot bus rescheduling problem. *PATAT 2014: Proceedings of the 10th International Conference of the Practice and Theory of Automated Timetabling*, pages 128–141, 2014.

[7] J. Jespersen-Groth, D. Potthoff, J. Clausen, D. Huisman, L. G. Kroon, G. Maróti, , and M. N. Nielsen. Disruption management in passenger railway transportation. Technical report, Erasmus University Rotterdam, 2007.

[8] J.-Q. Li, D. Borenstein, and P. B. Mirchandani. A decision support system for the single-depot vehicle rescheduling problem. *Computers & Operations Research*, 34(4):1008–1032, 2007.

[9] J.-Q. Li, P. B. Mirchandani, and D. Borenstein. The vehicle rescheduling problem: Model and algorithms. *Networks*, 50(3):211–229, 2007.

[10] J.-Q. Li, P. B. Mirchandani, and D. Borenstein. A lagrangian heuristic for the real-time vehicle rescheduling problem. *Transportation Research Part E: Logistics and Transportation Review*, 45(3):419–433, 2009.

[11] K. Nurmi, J. Kyngäs, and G. Post. Driver rostering for bus transit companies. *Engineering Letters*, 19(2):125–132, 2011.

[12] A. Tóth and M. Krész. A flexible framework for driver scheduling. *Proceedings of the 11th International Symposium on Operational Research in Slovenia SOR'11*, pages 341–345, 2011.

Traffic Sign Symbol Recognition with the D2 Shape Function

Simon Mezgec and Peter Rogelj
Faculty of Mathematics, Natural Sciences and Information Technologies
University of Primorska
Koper, Slovenia
simon.mezgec@student.upr.si, peter.rogelj@upr.si

ABSTRACT

This paper describes our application of a novel method in the field of traffic sign recognition - the D2 shape function. We first give an overview of the advances and research in this field. We then describe the D2 shape function that was originally used to classify 3D models of various objects - because of its robustness we propose its use in the field of traffic sign recognition. We describe how the program that was developed using this method works and present the results on a testing set of self-acquired speed limit traffic signs, evaluate its performance and compare it to the performance of statistical moments.

General Terms

Algorithms

Keywords

Symbol recognition, traffic signs, D2 shape function

1. INTRODUCTION

While driving a vehicle, the driver has to be constantly aware of a number of factors, one of which are traffic signs. These signs can have specific properties, such as color and shape, which allow their recognition with the use of Computer Vision. So it is no surprise that this is one of the important steps towards the automatization of driving vehicles. Once the traffic sign is detected, we would usually like to know what kind of symbol (if any) the traffic sign contains. Recognition is useful for many purposes: the driver could forget what the last speed limit he drove by was, an inexperienced driver could see a traffic sign for the first time and he would like to know its meaning etc. It is therefore apparent that traffic sign recognition has a real-world use.

There have been many different methods proposed in the field of traffic sign detection and recognition in the last two decades. Let us mention some implementations that attracted the most attention from the community. In 1996, Piccioli presented a robust method for traffic sign detection and recognition [10]. One year later, de la Escalera developed an algorithm that focuses on the recognition of traffic signs - the algorithm uses neural networks for the classification of traffic signs [4]. De la Escalera built on that work in his 2003 article where he used a genetic algorithm for the detection step, which allowed further invariance [3]. In 2000, Paclík proposed his own algorithm for traffic sign classification using the Laplace probability method and an Expectation-Maximization algorithm to maximize the likelihood function [9]. In 2004, Fang developed a traffic sign detection and recognition system that is based on a computational model of human visual recognition processing [5]. Two years later, Gao improved upon the human vision model recognition using separate models for extracting color and shape information, respectively [6]. In 2007, Maldonado-Bascón described a traffic sign detection and recognition system that is based on support vector machines which is invariant even to partial occlusions [7]. In the same year, Cyganek also used support vector machines in his system for traffic sign detection, whereas the recognition is performed using neural networks [2]. Finally, in 2009, Baró presented a system that performs traffic sign detection with the boosted detectors cascade and traffic sign recognition with a forest of optimal tree structures that are embedded in the ECOC (Error-Correcting Output Code) matrix [1], whereas in the next year, Ruta added a tracking procedure between the steps of traffic sign detection and recognition - this tracker predicts the position and scale of the sign candidate to reduce computation. That system uses Color Distance Transform for the classification of traffic signs [11].

In this paper we present an alternative method for traffic sign recognition, based on the D2 shape function that promises high robustness required for such applications. The method is presented in the next Section and its practical application in Section 3.

2. METHOD DESCRIPTION

For the recognition of traffic sign symbols we use the D2 shape function - a method first described by Osada in his 2001 article [8] - it is therefore a fairly new Computer Vision method. The function samples random pairs of points and creates a histogram of distances between these point-pairs. Figure 1 shows how the D2 shape function creates the distance histogram between pairs of points on a cup [12].

The idea behind the function is that with a large enough

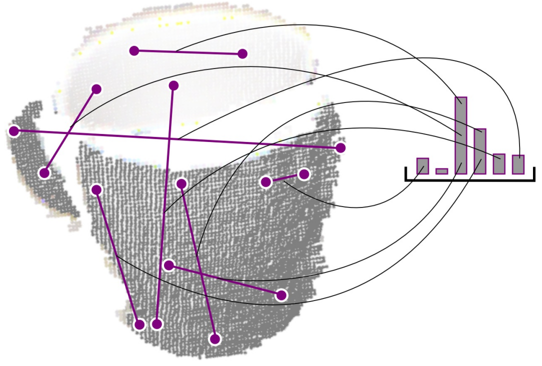

Figure 1: The distances between randomly generated pairs of points are converted to a distance histogram with the appropriate use of normalization. The histogram represents the D2 shape function for the cup and therefore defines the object description.[1]

number of sampled point-pairs on the object we get such a distribution of the distance histogram that is specific for the object, which means that we are able to recognize that object. In our case we compare the distance histogram with the histograms created from each of the images in the learning set. The procedure itself is therefore simple - we randomly generate a large number of point-pairs on the surface of the object and compute distances between them. It is worth noting that aside from the D2 shape function, Osada introduced other randomized shape functions [8]:

- A3: computing the angle between three randomly generated points on the surface of the 3D model of the object,
- D1: computing the distance between a fixed point and a randomly generated point on the surface of the object,
- D3: computing the square root of the surface of the triangle, defined by three randomly generated points on the surface of the object,
- D4: computing the cube root of the volume of the tetrahedron, defined by four randomly generated points on the surface of the object.

Out of all these functions, the D2 shape function turned out to be the most robust [12], which is why we decided to use it for our application in traffic sign symbol recognition.

3. PROGRAM DESCRIPTION

The program for traffic sign symbol recognition was developed in the Matlab programming language and in this Section we explain how it works. Here we do not focus on traffic sign detection and assume that traffic signs are already segmented and symbols cropped out of them. The detection and cropping is performed within the program for traffic sign detection which was previously developed and is not a part of the traffic sign recognition program.

[1]The image is taken from Wohlkinger's paper [12].

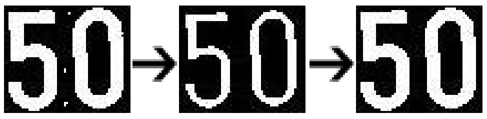

Figure 2: On the left we have the original segmented image. We then perform two morphological operations - first erosion (middle image) and then dilation (right image). The result is the symbol with the original size but without the noise.

3.1 Preliminaries

First we read the image, change its color space from RGB to HSV and perform the segmentation of the image by using the mean of all the brightness values - if a pixel has a brightness value smaller than the mean brightness of the image, we set the corresponding pixel of the binary image to 1, otherwise to 0. Once the image is successfully segmented, we have to make sure that aside from the traffic sign symbol we did not detect any anomalies that are not part of the symbol. These errors can occur due to different light reflections, dirt or dust on the camera lens as well as on the traffic sign etc.

We use two morphological operations for noise reduction - erosion and dilation. With erosion, we shrink all the areas in the image and as a result the noise is removed, but we also get smaller areas of the detected traffic sign symbol. Because we only want to delete the noise and preserve the symbol itself, we then use the morphological operation of dilation which expands the symbol to its primary size. Figure 2 shows the effect of these operations.

In this paper we focus on speed limit traffic signs, where the symbol can be divided into numerals that are recognized separately, which is why the next step is searching for connected components. This search is important for the recognition of numerical symbols, because with it, we can divide the number into numerals and since the numerals do not overlap, we can safely assume that one connected component represents one numeral. The reasoning behind this search is that we can achieve greater recognition accuracy by recognizing each of the numerals than by recognizing the whole number at once because we do not have to model additional dependencies between the numerals which can be different from traffic sign to traffic sign - the most obvious of these is the distance between the numerals in the number. In the case of non-numerical symbols, the components would be recognized separately as well.

3.2 Computing the D2 shape function

At the beginning of the D2 shape function computation we visit each of the connected components (except for the first one which represents the background). The first step is the generation of random x and y coordinates between values 1 and the image height for the x coordinate and between 1 and the image width for the y coordinate - the number of generated coordinates is defined by an input parameter. Then we go through the array of generated coordinates and for each pair of coordinates we check if the pixel with that coordinates lies on the numeral that is currently being recognized. If it does, we copy this pair into a new coordinate

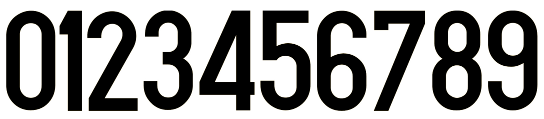

Figure 3: Learning set of images with numerals in the standard font of Slovenian traffic signs. Each numeral is saved in its own image.

array where we keep only the pixels that lie on the numeral.

After that there is the main part of the D2 shape function computation. We first check if we managed to find a sufficient number of coordinate pairs from pixels positioned on the numeral - if we did, we continue with the computation and if we did not, we abort the computation for the recognition of the current region and continue with the next one. The reason for this check is further insurance from the presence of noise - if we did not find a sufficient number of coordinate pairs, then we can conclude that the detected area is too small and consider it being noise.

Then we visit the first half of the array with randomly generated x and y coordinates and in each step we compute the Euclidean distance between a pixel in the first half of the array and a corresponding pixel in the second half. When we are done with the computation, we sort the distances by size and normalize them so they all have values from 0 to 1 - we do that by dividing every distance with the maximum distance obtained in the analyzed region. The latter is necessary for the comparison of images with different sizes. As a result we therefore return a matrix of these normalized and sorted distances for each of the detected numerals.

3.3 Learning and comparing the input image with the learning set

Apart from the main function of the traffic sign symbol recognition program, we use two more auxiliary functions to properly recognize the traffic sign - one for teaching the program on the images from the learning set and one for the recognition of the input image using the images from the learning set. Figure 3 shows the learning data set. It is important to note that the font for Slovenian traffic signs is standardized which means that one sample per numeral is sufficient. We will describe the learning process and the comparison in this Section.

For the learning, we go through the 10 images from the learning set and for each of them we run the symbol recognition code as many times as is defined by the input parameter and we then average the results. For the comparison of the input image with the learning set, we repeat the recognition multiple times so that we increase the probability of an accurate recognition - we then return the result that was recognized in most of the repetitions. It should be noted that we convert distances into histograms, which is needed because a direct comparison of distances would provide inaccurate results as the distance values are too specific. We compare each bin of the distance histogram of the current numeral on the input image with the histogram of the current image from the learning set and compute the Euclidean

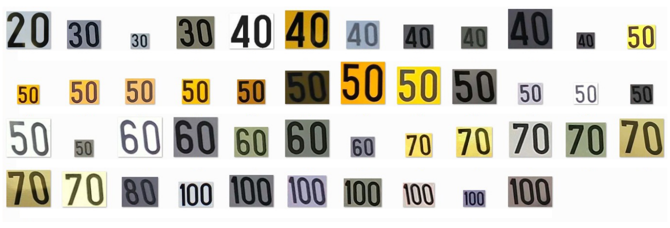

Figure 4: Testing set of speed limit traffic signs that was used for the definition of the parameters and testing the performance of the traffic sign symbol recognition program.

distances between pairs of bins at the same positions. We use the 1-nearest neighbor method to classify the numerals in the image because we only have one sample per numeral and one numeral represents one class.

4. RESULTS

For the definition of the parameters and testing the performance of the traffic sign symbol recognition program, we acquired a testing set of 46 speed limit traffic signs. Figure 4 shows this testing set of traffic signs. It contains images of traffic signs taken under different conditions - different lighting, orientation of the sign, size of the sign, quality of the image etc. This difference between the conditions of each image was achieved on purpose - to prove the robustness of the program.

The procedure of defining the parameters was performed with the help of auxiliary functions and the final values of the parameters are saved in a configuration file. They are as follow:

- number of randomly generated pixels for the D2 shape function computation: 100000,
- percentage of randomly generated pixels that have to lie on the symbol: 0.1 (10%),
- number of learning iterations for the images from the learning set: 5,
- number of iterations of comparison between the input image and images from the learning set: 10,
- number of bins in distance histograms: 20.

Once the parameters were defined, we ran the traffic sign symbol recognition program on the testing set of traffic signs to test the performance of the program. The final result is 42 of 46 accurately recognized speed limit traffic signs, which is approximately a 91.3% accuracy.

5. COMPARISON WITH STATISTICAL MOMENTS

When developing the program for traffic sign recognition, we first tried using statistical moments as the recognition method. In particular, we used a combination of normalized central moments, which are invariant to the position as well as to the size of the object, and Hu moments, which are further invariant to rotation. The framework of the prototype is very similar to the program described in Section 3, and the learning and testing sets of traffic sign images are the same, so the only difference is in the method that was used for each prototype. We can therefore compare the results of the two traffic sign recognition programs.

After testing the statistical moments prototype, it quickly became obvious that it would not be a good fit for traffic sign recognition - we tested the program several times, every time with different maximum orders of statistical moments (from 5 all the way up to 40). The problem was that the calculated values of statistical moments did not yield any discernible patterns, which is the basis for object recognition. The prototype thus behaved unpredictably and often recognized a false symbol. After using the same program framework to develop a prototype using the D2 shape function and getting promising results, we concluded that the fault was not in the way we developed the program or in the set of acquired images, but in the method used. We therefore abandoned statistical moments and moved on to search for more suitable methods, until we found the D2 shape function.

While statistical moments have a wide range of use, we found the D2 shape function to be much more suitable for traffic sign recognition.

6. CONCLUSION

Shape functions up until now were not used in traffic sign recognition - we presented a prototype using the D2 shape function that recognizes traffic signs. It should be noted that while the testing set, presented in Section 4, is a collection of traffic sign images that are as varied as possible, therefore maximizing the robustness of the traffic sign recognition program, the set is still quite small - only 46 images. The problem is that there is no publicly available depository or collection of Slovenian traffic sign images and the set therefore had to be acquired by hand, which, due to the fact that we want to capture as many different traffic signs as possible, is very time-consuming. That is why we focused only on speed limit traffic signs, but we could easily expand the scope to all traffic signs as the D2 shape function is not limited to any number of components. To add a new type of traffic sign, all we would need to do is to add the corresponding traffic sign symbol to the learning set.

As described in Section 4, the final performance of the program is 91.3% accurately recognized speed limit traffic signs, which is an encouraging result and an initiative for further use of the D2 shape function in the field of traffic sign recognition and possibly other fields too. It is a robust procedure which also works on very small and low quality images - the smallest image used for testing had a resolution of 38×31 pixels and the program correctly recognized it. It is worth noting that this method for recognition is much simpler than traditional methods, in particular comparing to solutions, mentioned in the Introduction, where methods like neural networks and support vector machines were used. In terms of performance we can not directly compare it to these solutions since we do not have a testing set that would be large enough, but we did get a sense that it is a robust method.

As a final note, the D2 shape function performs much better than statistical moments for traffic sign recognition, as is described in Section 5. Considering our results we can conclude that the D2 shape function seems well suited for this field of Computer Vision.

7. REFERENCES

[1] Xavier Baró, Sergio Escalera, Jordi Vitrià, Oriol Pujol, and Petia Radeva. Traffic Sign Recognition Using Evolutionary Adaboost Detection and Forest-ECOC Classification. *IEEE Transactions on Intelligent Transportation Systems*, 10(1):113–126, March 2009.

[2] Bogusław Cyganek. Circular road signs recognition with soft classifiers. *Integrated Computer-Aided Engineering*, 14(4):323–343, 2007.

[3] Arturo de la Escalera, José M. Armingol, and Mario Mata. Traffic sign recognition and analysis for intelligent vehicles. *Image and Vision Computing*, 21(3):247–258, March 2003.

[4] Arturo de la Escalera, Luis E. Moreno, Miguel A. Salichs, and José M. Armingol. Road Traffic Sign Detection and Classification. *IEEE Transactions on Industrial Electronics*, 44(6):848–859, December 1997.

[5] Chiung-Yao Fang, Chiou-Shann Fuh, Pei-Shan Yen, Shen Cherng, and Sei-Wang Chen. Road traffic sign detection and classification. *Computer Vision and Image Understanding*, 96(2):237–268, November 2004.

[6] Xiaohong W. Gao, Lubov N. Podladchikova, Dmitry G. Shaposhnikov, Keum-Shik Hong, and Natalia A. Shevtsova. Recognition of traffic signs based on their colour and shape features extracted using human vision models. *Journal of Visual Communication and Image Representation*, 17(4):675–685, August 2006.

[7] Saturnino Maldonado-Bascón, Sergio Lafuente-Arroyo, Pedro Gil-Jiménez, and Hilario Gómez-Moreno. Road-Sign Detection and Recognition Based on Support Vector Machines. *IEEE Transactions on Intelligent Transportation Systems*, 8(2):264–278, June 2007.

[8] Robert Osada, Thomas Funkhouser, Bernard Chazelle, and David P. Dobkin. Matching 3D models with shape distributions. *SMI 2001 Proceedings of the International Conference on Shape Modeling and Applications*, pages 154–166, May 2001.

[9] Pavel Paclík, Jana Novovičová, Pavel Pudil, and Petr Somol. Road sign classification using Laplace kernel classifier. *Pattern Recognition Letters*, 21(13-14):1165–1173, December 2000.

[10] Giulia Piccioli, Enrico De Micheli, Pietro Parodi, and Marco Campani. Robust method for road sign detection and recognition. *Image and Vision Computing*, 14(3):209–223, April 1996.

[11] Andrzej Ruta, Yongmin Li, and Xiaohui Liu. Real-time traffic sign recognition from video by class-specific discriminative features. *Pattern Recognition*, 43(1):416–430, January 2010.

[12] Walter Wohlkinger and Markus Vincze. Analysis and Evaluation of Shape Functions for Object Class Recognition. *Proceedings of the 17th Computer Vision Winter Workshop*, February 2012.

On numerical simulation of filtration gas combustion processes on the shared memory machines.

Tatyana Kandryukova
Institute of Computational Mathematics and Mathematical Geophysics SB RAS
630090, prospect Akademika Lavrentjeva, 6
Novosibirsk, Russia
kandryukova@labchem.sscc.ru

ABSTRACT

This work is devoted to the searh of the efficient algorithms for the simulation of filtration gas combustion processes. In particular, a two-level parallel algorithm based on the explicit finite difference scheme using an adaptive mesh is constructed. Two ways of parallelization, namely, the direct usage of OpenMP directives and special distribution of data across threads are carried out. It is numerically shown, that the last one provides significant performance advantage.

Categories and Subject Descriptors

D.1.3 [**Programming Techniques**]: Concurrent Programming; G.1 [**Numerical Analysis**]: Miscellaneous; J.2 [**Physical Sciences And Engineering**]: Mathematics and statistics, Physics, Chemistry

General Terms

Algorithms, Performance

Keywords

Combustion wave, explicit difference schemes, adaptive grids, parallelization on shared memory, OpenMP-threads

Supervisor

Yuri Laevsky
Institute of Computational Mathematics and Mathematical Geophysics SB RAS
630090, prospect Akademika Lavrentjeva, 6
Novosibirsk, Russia

1. INTRODUCTION

The phenomena of filtration gas combustion (FGC) had been discovered in the 70s of the previous century and it has still not been studied fully. Meanwhile the knowledge of the properties of FGC processes is essential in solving many problems of chemical technology, ecology, fire safety, etc. In particular low-speed mode of filtration gas combustion process, which is under consideration, is the physical basis for construction modern industrial flame arrester, environmentally friendly burners and other. Therefore numerical simulation of FGC is the problem of current interest that has important practical application.

From the physical point of view the process presents the propagation of the region of gaseous exothermic reaction in chemically inert porous medium, as gaseous reactants are being supplied into the region of chemical transformation [2]. It is implemented as follows. Let there be a tube filled with a porous material, measuring about 10 cm. From one edge of it a combustible gas mixture is supplied at a rate of v. Then the mixture is ignited, resulting in a flat combustion front, which can either be stationary or move in any direction (at a rate of u), depending on the model parameters (see Figure 1). It occurs due to the heat recuperation. In the heating zone fresh gas mixture gets hot and the chemical process occurs in the reaction zone that causes heat release. In the relaxation zone high-temperature products of combustion exchange heat with the porous solid and then heat transfers through it back to the heating zone by means of thermal conduction.

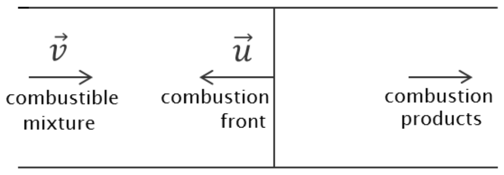

Figure 1: The scheme of the physical model of FGC

The main difficulty in the simulation of this process is its very different scales; it is especially hard to deal with the kinetic-diffusion distinction.

In this study the simplest case is considered, when the model is one-dimensional, and still it takes hours to simulate the process that lasts less than a half of a minute. It doesn't seem possible to proceed to multidimensional case under such conditions. At the present time great expectations for having the opportunity to deal with such tasks are related to the appearance of multi-processor machines and the development of parallel methods.

This work is an expansion of [3] and devoted to the analysis of the possibilities to improve the performance of some computational models of FGC in low-speed mode. In particular the introduction of adaptive nested grid [5] and parallelization on shared memory [1] are discussed. All the constructed algorithms provide us with the solutions of the problem that coincide with the experimental data. The special method of parallelization on the machines with shared memory was developed and applied to the constructed algorithms, that reduces the computational time greatly not only compared to the sequential implementation, but to the classical way of parallelization with the direct usage of OpenMP pragmas as well.

2. MATHEMATICAL BASIS

2.1 Mathematical model

The simplest one-dimensional FGC model in the enthalpy formulation includes three equations:

$$\frac{\partial T}{\partial t} = a_s \frac{\partial^2 T}{\partial x^2} + \alpha_s (H - T - \frac{Q}{c_g}\eta), \tag{1}$$

$$\frac{\partial H}{\partial t} = a_g \frac{\partial^2 H}{\partial x^2} - v\frac{\partial H}{\partial x} + \alpha_g (T - H + \frac{Q}{c_g}\eta), \tag{2}$$

$$\frac{\partial \eta}{\partial t} = a_g \frac{\partial^2 \eta}{\partial x^2} - v\frac{\partial \eta}{\partial x} - W(\eta, H). \tag{3}$$

Here $a_i = \lambda_i / c_i \rho_i$ is the coefficient of thermal diffusivity of the i-th phase, c_i, ρ_i, λ_i are respectively, specific heat at constant pressure, density and thermal conductivity of i-th phase ($i = s$ for porous solid, $i = g$ for gas), $\alpha_s = \frac{\alpha}{(1-m)c_s\rho_s}$, $\alpha_g = \frac{\alpha}{mc_g\rho_g}$, m - porosity, α - interphase heat transfer rate, v - flow rate of the combustible mixture, $T \equiv T_s$, $c_g H = c_g T_g + Q\eta$ is full gas enthalpy, where T_i is the temperature of the i-th phase, Q is energy release of the reaction, $W(\eta, H) = k_0 \eta e^{-E/R(H - \frac{Q}{c_g}\eta)}$ - the chemical reaction rate according to Arrhenius law, η - relative concentration of reactive component of the combustible mixture, k_0 - pre-exponential factor, E - activation energy, R - universal gas constant. It is worth noting here that the speed of the wave u is a priori unknown.
The Cauchy problem is stated by adding the Dirichlet boundary conditions on the left edge and the Neumann ones on the right. The initial data correspond to the preheated porous medium.

2.2 Implementation on a regular and adaptive meshes

As there is no analytical solution of the problem the suitability of all the constructed algorithms is estimated by comparison with the solution obtained on the regular fine mesh by using the explicit difference scheme:

$$T_i^{n+1} = T_i^n + \tau(a_s \frac{T_{i-1}^n - 2T_i^n + T_{i+1}^n}{h^2} + \\ + \alpha_s(H_i^n - T_i^n - \frac{Q}{c_g}\eta_i^n)), \quad i = 1, ..., M \tag{4}$$

$$H_i^{n+1} = H_i^n + \tau(a_g \frac{H_{i-1}^n - 2H_i^n + H_{i+1}^n}{h^2} - v\frac{H_i^n - H_{i-1}^n}{h} + \\ + \alpha_g(T_i^n - H_i^n + \frac{Q}{c_g}\eta_i^n)), \quad i = 1, ..., M \tag{5}$$

$$\eta_i^{n+1} = \eta_i^n + \tau(a_g \frac{\eta_{i-1}^n - 2\eta_i^n + \eta_{i+1}^n}{h^2} - \\ - v\frac{\eta_i^n - \eta_{i-1}^n}{h} - W(\eta_i^n, H_i^n)), \quad i = 1, ..., M. \tag{6}$$

Here M is the number of spatial steps, size of which h is chosen from the condition:

$$\frac{\max_i |T_i^N(h) - T_i^N(2h)|}{\max_i T_i^N(2h)} < 0.02, \tag{7}$$

where N is the number of time steps. Size of time step τ must satisfy a stability condition for explicit difference schemes, that is theoretically unknown, and is fitted experimentally. The table 1 represents values of $K(h)$ and $\tau(h)$ corresponding to different values of h. Here $L = 0.1$ is length of the tube.

h	$L \cdot 2^{-10}$	$L \cdot 2^{-11}$	$L \cdot 2^{-12}$	$L \cdot 2^{-13}$	$L \cdot 2^{-14}$
$\tau(h)$	2^{-16}	2^{-17}	2^{-19}	2^{-21}	2^{-23}
$K(h)$	0.159	0.096	0.053	0.028	0.014

Table 1: Values of $K(h)$ and $\tau(h)$ corresponding to different values of h

Meanwhile away from the flame front, the solution functions are quite smooth and don't need such small steps (see graphic example at Figure 2). Thus, one way to speed up

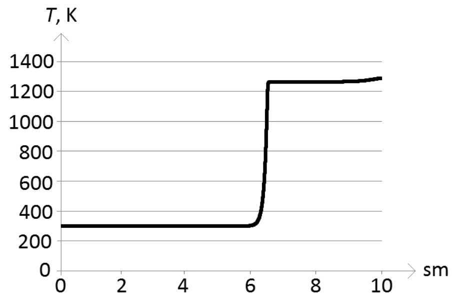

Figure 2: Graph example of the temperature of solid T, obtained numerically

the computation seems to be the introduction of an adaptive grid. It should depend on the solution from the previous time step and be concentrated in the area of the chemical transformation. It's carried out as follows. Let there be a full-length regular coarse spatial grid. Implementation of one time step of the difference scheme on it with the corresponding big time step provides us with the boundary conditions on the next time layer for the small task, that is stated in the vicinity of the chemical reaction zone similarly the large one. For this enclosed task the denser time-space mesh is used. The initial data and boundary conditions at the every time layer are obtained by the means of linear interpolation from the coarse mesh. After the execution of all the steps of the subproblem the values of coarse-grid solutions in the nodes used for interpolation are replaced with the corresponding values from the embedded problem (see Figure 3, where n is the number of the current time layer of the general problem, p is the number of time steps of the subproblem, i_0 is the number of the node where W gets its maximum). The dense grid moves according to the prop-

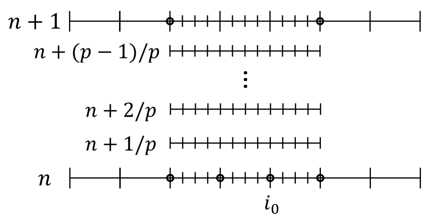

Figure 3: The scheme of the two-level algorithm

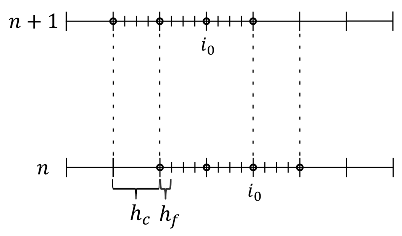

Figure 4: The scheme of the movement of the adaptive fine grid

agation of the combustion front: as soon as the maximum point of W function moves by the spatial step of the coarse grid, the fine mesh moves by the same distance (see Figure 4). This approach allows us to take big spatial and time steps outside the chemical reaction zone. The numerical experiments show that to achieve the same accuracy grade one may use the coarse grid with the spatial step four times bigger than the one of fine grid. That implies about sixteen times bigger time steps. It is obvious thus that the usage of such two-level time mesh is indispensable in the problem like the one in question.

3. PARALLELIZATION

3.1 Classical way

The simplest and the most obvious way to implement the program on a multiprocessor node is application of OpenMP pragmas to the inner loops of the program. All the arrays in this case are shared and all the threads work to fill each of them. As a result data of different types are held in the storages of different threads and the threads have to communicate intensively. In the case of the regular mesh it concerns to the separate computations of the exponent values and the values of η in the scheme equations of which they are used, as well as to the move from one time layer to another. The situation is far more complicated for the algorithm with the enclosed fine mesh, because in this case there are a lot of bottlenecks besides the mentioned ones, related to the repeated transition from coarse mesh to the fine one and back. Such way of parallelism provides some acceleration for the execution of regular mesh algorithm, but it leads even to the slowdown of the computation when the adaptive mesh is used.

3.2 Data distribution

To reduce the time for the exchange of data flows we propose to distribute the data between the threads. In the case of regular mesh by these words we mean the following procedure. The spatial grid is divided into z pieces, where z is the number of threads used, and each thread gets control over one of the pieces: thread number zero deals with nodes $[0,\dots,M/z]$, the next one – with $[M/z,\dots,2M/z]$ and so on. The last one deals with nodes $[(z-1)M/z,\dots,M]$. Each thread creates its local arrays, fills them in accordance with the difference scheme, and outputs the results on its assigned grid nodes. On a regular grid we thus obtain practically independent tasks exchanging only boundary conditions.
In the case of adaptive grids communication becomes more complicated because of the need to transfer information from the nodes of the fine mesh to those of the coarse one. In this case the data distribution algorithm has few stages. After n time steps of general task we have data distributed across z threads: $[0,\dots,M_c/z],[M_c/z,\dots,2M_c/z],\dots,[(z-1)M_c/z,\dots,M_c]$ and shared arrays of size (M_f+1), where M_c, M_f are the numbers of nodes of the coarse and fine meshes respectively. The first stage is the implementation of one time step of coarse grid by every thread separately (see Figure 5). The second stage prepares the data for the

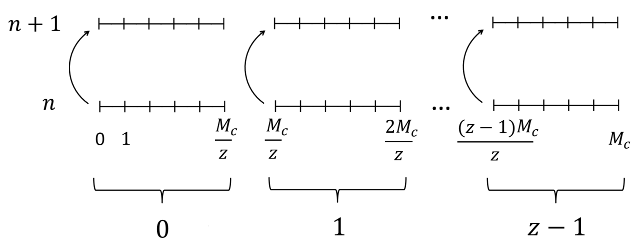

Figure 5: The first stage of data distribution algorithm in the case of adaptive grid

inner problem. It is meant that the position of the fine grid with respect to the coarse one is known and i_{left}, i_{right} are the numbers of the nodes being the edges of the enclosed grid. Initial data obtained by interpolation from the layer n of coarse grid are entered to the shared arrays and then redistributed across z threads, forming the layer $(n+0/p)$ of the enclosed problem (Figure 6). Then the enclosed problem is

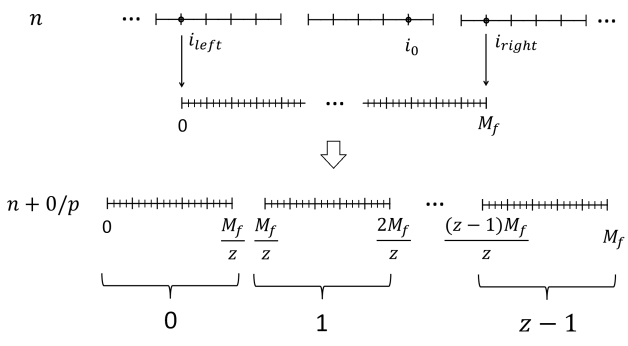

Figure 6: The second stage of data distribution algorithm in the case of adaptive grid

solved in parallel by analogy with the first stage, i.e. steps $(n+1/p), \ldots, (n+p/p)$ are implemented by every thread separately (Figure 7). The last stage is the opposite to the

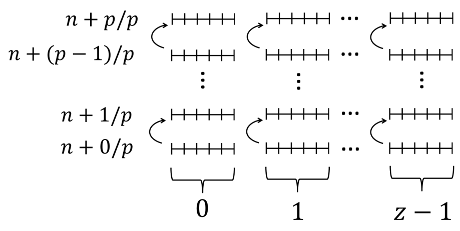

Figure 7: The third stage of data distribution algorithm in the case of adaptive grid

second one. The solution obtained from the layer $(n+p/p)$ of the inner problem is entered to the shared array and then necessary data are transmitted to the sertain threads in according to the data distribution on the coarse grid. Besides that the position of the adaptive grid is determined at this step.

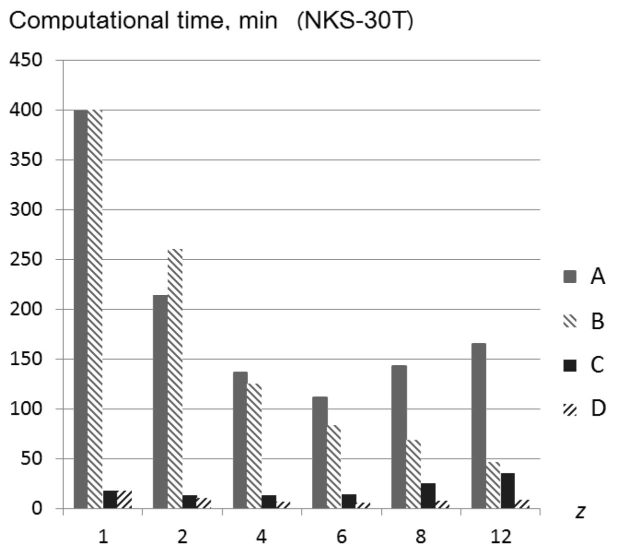

Figure 8: Computational time of all the algorithms depending on the number of threads z: A - regular mesh and direct usage of OpenMP pragmas, B - regular mesh and usage of parallelism based on the data distribution, C - adaptive mesh and direct usage of OpenMP pragmas, D - adaptive mesh and usage of parallelism based on the data distribution; obtained with NKS-30T of SSCC

To get more detailed information about the construction of the adaptive grid and implementation of the respective algorithm with the data distribution see [4].

4. RESULTS

Numerical experiments were held for the model with the tube length $L = 0.1$ m for the physical time $t = 15.0$ s. All the calculations were performed either on NKS-30T Cluster of Siberian Super Computer Center (SSCC), each node

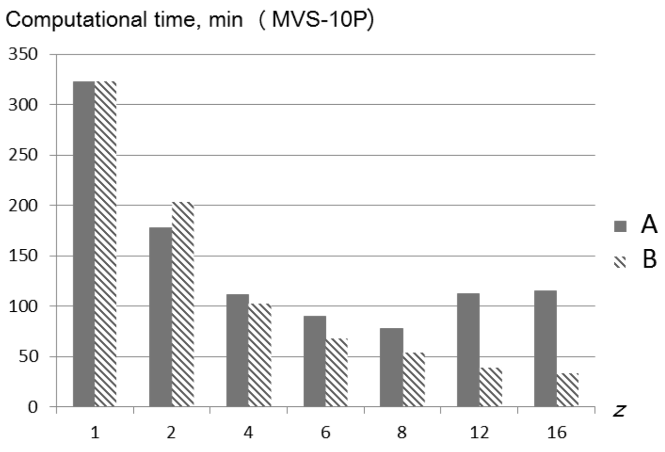

Figure 9: Computational time of all the algorithms depending on the number of threads z: A - regular mesh and direct usage of OpenMP pragmas, B - regular mesh and usage of parallelism based on the data distribution; obtained with MVS-10P of JSCC RAS

of which contains two processors Intel®Xeon®X5670 (12M Cache, 2.93 GHz, 6.40 GT/s Intel®QPI, 6 cores), or on MVS-10P of Joint Supercomputer Center of the Russian Academy of Sciences (JSCC), each node of which contains two processors Xeon®E5-2690 (20M Cache, 2.90 GHz, 8.00 GT/s Intel®QPI, 8 cores). Hyper-threading is turned off, the KMP_Affinity environment variable is set equal to "compact". At the Figure 8 one may see a bar chart of the computational time of each algorithm executed on NKS-30T SSCC. First of all the importance of introduction of the adaptive grid in the case of sequentially running programs is well seen from this picture. The use of this technique reduces computational time by a factor of 22 when the only one thread works. Secondly one could easily compare the efficiency of both methods of parallelization applied. In the case of the regular mesh the direct application of OpenMP directives provides some improvement for the number of threads less or equal 6. As the number of threads grows data amount becomes less while the quantity of data transmissions increases greatly. So for more than 6 threads there is slowdown of parallel computation based on direct usage of OpenMP pragmas. The proposed data distribution algorithm, in opposite, shows rather good scalability per number of threads up to $z = 12$ in this case. The same data are presented by the means of the Table 2 as well.

Analogous results for the regular mesh obtained on MVS-10P cluster are presented at the Figure 9 and in the Table 3. It is seen that the same calculations take a little less time, but the general situation is still the same up to 16 threads. As concerns the algorithm with the enclosed problem the first method of parallelization is inadmissible since the more threads is used the longer computation lasts. At the same time it is seen that for the algorithm with the enclosed problem even the proposed method doesn't provide good scalability though has better effect than the usage of OpenMP pragmas. So there is an opportunity that with the greater number of threads the usage of the adaptive grid might become ineffective.

number of threads z	1	2	4	6	8	12
A	400	214	137	112	143	166
B	-	260	125	83.0	68.3	46.1
C	17.8	13.7	12.8	14.0	25.4	35.3
D	-	10.4	6.69	5.82	7.82	8.69

Table 2: Computational time of all the algorithms depending on the number of threads z: A - regular mesh and direct usage of OpenMP pragmas, B - regular mesh and usage of parallelism based on the data distribution, C - adaptive mesh and direct usage of OpenMP pragmas, D - adaptive mesh and usage of parallelism based on the data distribution; obtained with NKS-30T of SSCC

number of threads z	1	2	4	6	8	12	16
A	323	178	112	90.2	77.4	113	115
B	-	203	103	68.2	54.2	38.7	33.5

Table 3: Computational time of all the algorithms depending on the number of threads z: A - regular mesh and direct usage of OpenMP pragmas, B - regular mesh and usage of parallelism based on the data distribution; obtained with MVS-10P of JSCC RAS

5. CONCLUSIONS

The appearance of supercomputers and elaboration of parallel technology have given renewed impetus for the further development of numerical methods and has opened the doors for the modeling of complex tasks. The problem of FGC is the one of such problems. The specific of its solutions causes the usage of an adaptive mesh to be extremely efficient for the numerical simulation of FGC processes in the case of sequential implementation. However, one should pay special attention to the parallel realization of such an algorithm, since unfortunate parallelization might not only show unsatisfactory scalability, but even augment the computational time with increasing number of threads. At the same time the fact, that the number of cores on the computational node constantly grows and the influence of parallelism increases, implies the need to construct new algorithms permitting almost perfect scaling in the number of threads.
In the paper a special approach to the issue of shared memory parallelism based on the distribution of data across threads has been proposed. It has been applied to the algorithms both with the regular and the adaptive grids. The results of comparison this method with the classical one, when OpenMP directives is applied to all the internal loops of the program, have shown that the proposed method is more efficient for the task in question and is acceptable to the parallel implementation of the algorithm with the usage of embedded fine mesh. All calculations were performed for the problem with characteristic dimensions and empirically chosen parameters of the mathematical model. Solutions produced by all the constructed algorithms have the required accuracy grade of approximation and correspond to physical data. Meanwhile the computational time is reduced by 10 times in the case of the regular mesh and by 3 times in the case of the adaptive one by the use of the proposed method of parallelization.

6. ACKNOWLEDGMENTS

This work was supported by RFBR (12-01-31046, 13-01-00019, 13-05-12051).

7. REFERENCES

[1] *The OpenMP®API specification for parallel programming.* http://openmp.org.

[2] V. S. Babkin and Y. M. Laevskii. Seepage gas combustion. *Combustion, Explosion, and Shock Waves*, 23(5):531–547, September-October 1987.

[3] T. A. Kandryukova and Y. M. Laevsky. Numerical simulation of filtration gas combustion. In *Proceedings of the 11th International Conference on Mathematical and Numerical Aspects of Waves*, pages 43–44, June 2013.

[4] T. A. Kandryukova and Y. M. Laevsky. Simulating the filtration combustion of gases on multi-core computers. *Journal of Applied and Industrial Mathematics*, 8(2):218–226, April 2014.

[5] Y. M. Laevsky and L. V. Yausheva. Simulation of filtrational gas combustion processes in nonhomogeneous porous media. *Numerical Analysis and Applications*, 2(2):140–153, April 2009.

matcos-13

University of Primorska Press
www.hippocampus.si
ISBN *978-961-6984-20-1*
Not for resale