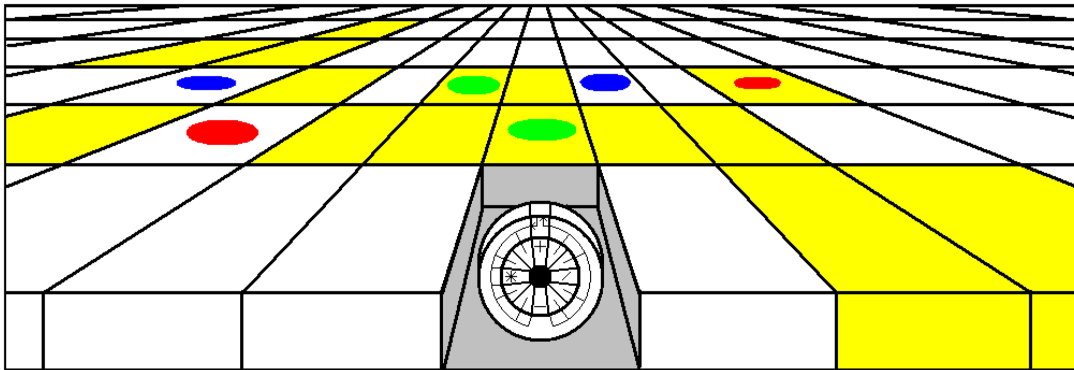
High Level Parallel Programming Language Compiling to a Cellular Automata Processing Model



Martin Mortensen - 20040896

Supervisor - Michael I. Schwartzbach

October 1st 2007

Contents

1 Prologue **5**

2 Introduction **8**
2.1 The history of Cellular Automata . . . 9
2.2 Cellular Automata . . . 10
2.2.1 Conway's Game of Life . . . 11
2.2.2 Ground Rules for my solution . . . 12
2.3 Cellular Automata Processing Model . . . 14
2.3.1 High Level Programming Language with a CAPM backend . . . 14
2.3.2 Motivation . . . 15
2.3.3 Sequential vs. Parallel . . . 17
2.3.4 Why implement a Cellular Automata Processing Model on the GPU . . . 19
2.3.5 Similar parallel languages and principles . . . 20

3 Abstractions and design of CAPM **23**
3.1 CAPM Principles . . . 23
3.1.1 Parser and compiler . . . 23
3.1.2 Meta Cellular Automata Layer Instruction Set - MCALIS . . . 24
3.1.3 Cellular Automata as binary trees . . . 24
3.1.4 MCALIS to CALIS . . . 26
3.1.5 Message passing protocol . . . 26
3.2 Time to run the Runtime CAs . . . 26
3.3 Designing a simple example - SACAM . . . 27
3.3.1 Simple Arithmetic Cellular Automata Model . . . 27
3.4 A theoretical benchmarking and comparison . . . 32

4 A tiny example language **34**
4.1 Cellular Automata Tiny Imperative Programming Language . . . 34
4.2 Tiny Imperative Programming Language . . . 34
4.2.1 Possible TIP modifications . . . 36

5 Implementing CATIP **37**
5.1 CATIP Context Free Grammar and parser . . . 37
5.2 CATIP Compiler and Static Analysis . . . 38
5.3 Sequentializing static analysis and runtime control structure . . . 39
5.3.1 SSA . . . 39
5.3.2 Using ϕ-functions as a runtime Sequentializing control structure . . . 39
5.3.3 Runtime control structure . . . 44
5.3.4 Static analysis required to create non-magical ϕ-functions . . . 45
5.3.5 Optimization . . . 48
5.3.6 Summary . . . 48
5.4 AST to MCAPM . . . 49
5.4.1 Step 1. Sequentialization . . . 50
5.4.2 Step 2. Initiate basis . . . 50
5.4.3 Step 3. Compile main function . . . 50
5.4.4 Step 4. Create Reload statements . . . 51
5.4.5 Step 5. Compile ϕ expressions . . . 51
5.4.6 Step 6. Create Clear statements . . . 52

5.5 MCAPM to CAPM . . . 52
5.6 Message Passing Protocol . . . 53
5.6.1 The CAL_CellularAutomaton message passing interface . . . 56
5.6.2 SCAMP - Simple Cellular Automata Message Passing . . . 57
5.6.3 Optimizations . . . 60
5.7 CAPM design and runtime algorithm . . . 62
5.7.1 CAL_Grid . . . 62
5.7.2 CAL_CellularAutomaton . . . 62
5.7.3 CALIS_Node . . . 63
5.7.4 Subclasses of CALIS_Node . . . 64
5.7.5 CAL_Message . . . 65
5.7.6 CAL_MessagePassingProtocol . . . 66
5.7.7 The design of the next version of CAPM . . . 67
5.8 The CAPM-VM GUI . . . 67
5.8.1 The basics . . . 67
5.8.2 Advanced features . . . 68

6 Testing CAPM **71**
6.1 CAPM vs. Parallel . . . 71
6.1.1 Parallel test 1: Hardcoded Boolean Matrix Multiplication . . . 72
6.1.2 Parallel test 2: Hardcoded Boolean Matrix to a power . . . 74
6.1.3 CAPM vs. Parallel conclusion . . . 75
6.2 CAPM vs. Sequential . . . 75
6.2.1 Sequential test 1: Hardcoded Boolean Matrix Multiplication with output . 75
6.2.2 Sequential test 2: Hardcoded Boolean Matrix to a power . . . 76
6.2.3 CAPM vs. SBPM conclusion . . . 78
6.3 Investigating SCAMP . . . 78
6.3.1 Worst case travel time estimate . . . 79
6.3.2 Average travel time estimate . . . 80
6.3.3 SCAMP conclusion . . . 80
6.4 Investigating impact of ϕ-functions . . . 80
6.4.1 Is the impact of the ϕ-functions acceptable? . . . 80
6.5 Memory usage . . . 81
6.5.1 Conclusion of Memory test . . . 81
6.6 Evaluating tests . . . 82
6.7 Test conclusion . . . 83
6.7.1 Missing tests . . . 83
6.7.2 What did the tests show? . . . 84

7 Future work - Developing CAPM **85**
7.1 Extensions . . . 85
7.1.1 Creating new logical nodes at runtime . . . 85
7.1.2 Adding pointers to CAPM . . . 85
7.1.3 Adding functions to CAPM . . . 85
7.1.4 Language extensions . . . 87
7.2 Optimizations . . . 87
7.2.1 Optimizations of the ϕ-functions . . . 87
7.2.2 Optimizations of placement of logical nodes . . . 87
7.2.3 Parallelizing static analysis . . . 87
7.2.4 Branch prediction . . . 88

7.3 Spin-off projects 88
7.3.1 Implement CAPM on more suitable architectures 88
7.3.2 Deconstruct the CAPM runtime algorithm 89
7.3.3 Cyber Foraging 89

8 Conclusion **91**
8.1 Returning to the initial questions 91
8.2 Returning to the motivation 91
8.3 Conclusion 93

9 Epilogue **94**

A Example Appendix **95**
A.1 The GetDirection method used by SCAMP 95

B Bibliography **96**
B.1 Primary Sources 96
B.2 Secondary sources 96

Abstract

English This Master's thesis provides motivation to develop a cellular automata processing model, which as a backend for high level imperative programming languages, can be a future alternative to the random access memory stack based processing model. To investigate whether or not it is possible to implement such a model and what the problems and possible solutions to these might be, compiler and virtual machine prototypes have been developed for a simple imperative programming language. The results of the development indicate that a cellular automata processing model can be considered as an alternative to random access memory stack based processing models, but that much more research is needed. The performance tests of the implemented virtual machine, shows that with random read access it can achieve performances associated with parallel processing and with static read access it can achieve performances equivalent to or better than random access memory stack based processing model.

Danish Dette speciale giver motivation til at udvikle en processeringsmodel baseret på cellulær automat som, ved at være backend for et høj niveau programmeringssprog, kan være et fremtidigt alternativ til stakbaserede processeringsmodeller der bruger RAM. For at undersøge om dette er muligt og derudover hvilke problemer og tilhørende løsninger der er, er der blevet implementeret prototyper af compiler og virtuel maskine, for et simpelt programmeringssprog. Resultaterne af dette speciale indikerer at en processeringsmodel baseret på cellulære automater, kan være et reelt alternativ til stak baserede processeringsmodeller der bruger RAM, men at meget mere forskning er nødvendig. Testene viser at den virtuelle maskine der er implementeret kan, hvis der tillades totalt frihed af læsning, opnå ydelse der svarer til parallel processering og hvis der kun tillades statisk defineret læsning, så kan der opnås ydelse svarende til eller bedre end stakbaserede processeringsmodeller med RAM.

1 Prologue

I have chosen to do my master's thesis on Cellular Automata as a general processing principle, because, in the course of my study and programming in my spare time, I have often wondered why we are using a processing model that always lacks power when needed. Some examples could be graph or tree traversal and in general problems with an intuitive parallel solution. Just think of how breadth-first traversal of a directed graph or matrix multiplication are explained. The intuitive solutions are more or less parallel, but are artificially sequentialized because of the habit of using a sequential processing model. This lead me to the work on my Master's thesis, which is:

Cellular automata as a general processing model can be an alternative to the stack based processing models. The performance of a cellular automata processing model can be efficient and scalable. Furthermore it is possible to create a high level parallel programming language that enables programmers to easily utilize the power of the cellular automata processing model.

During my studies I had the opportunity to take the course GPGPU - *General Purpose Computation using the Graphics Processing Unit* in which I and two others implemented the Graph Reachability Problem in textures and solved them by rendering. This was back in the fall semester of 2005 and already back then, our GPU Reachability solver was (empirically) about twice as fast as the CPU solver, implemented in C++. One thing that was clear from this course was that coding GPGPU was complex and furthermore quite inaccessible. Back then I proposed the idea of coding normally and have a compiler translate programs into textures, that would be run on the video card, and this concept could be used to create a GPU virtual machine and thereby a platform independent use of the processing powers available in graphics card.

From this idea, to translate programs into pixels and run it on the video card, a more general principle of a processing model was formed. This model was that entities only know only their

neighbors and all are updated every iteration, by the same update algorithm. This update is done by looking at the result of the neighbors and pixel itself in the last iteration. This is actually a well-known processing principle, namely Cellular Automata.

Cellular Automata (1940) is one of the three[1] original processing principles, proposed in the infancy of computing. The two others are the Turing Machine (TM) (1936) and the Λ-calculus (1936).

The TM have developed into the commonly used Von Neumann architecture and dominating languages such as C, C++, Java, Beta and many others.

Λ-calculus have developed into functional languages such as LISP, ML, and others.

Cellular Automata have no offspring to speak of.

Before my work on my master's thesis, the only way in which I had had anything to do with Cellular automata, was by coding Conway's Game of Life, in a couple of different languages and with a couple of modified rules. Having this experience in the back of my head, I decided that I was not going to try to achieve a CA of a complexity equal to that found in Conway's Game of Life, but rather, by having a much higher level of abstraction, show that a Cellular Automata Processing Model (CAPM) can be used as a processing model for *easy* programming languages. And that this can be done by a compiler outputting Cellular Automata instead of Assembler. In other words, the programmer does not code the Cellular Automata, the Cellular Automata runs the code he writes. A reason for this is that software development have shown again and again, that it is not the processing model that is the problem, when creating programs. It is the complexity of the program as a whole.

Introducing a new programming language, with all kinds of CA horns and whistles, would stifle the use of CA processing model, simply because it would be too inaccessible. It would furthermore carry the risk that goes with throwing years and years of experience out the window. I am convinced that using a "normal" high level language can be done, without to many unwanted consequences because the semantics of commonly used languages already contains much parallelism. It is because of the processing model used that the sequentialism is, in my view, artificially introduced.

The primary goal of my Master's Thesis is not to create a new kind of computer, but rather to prepare the principles needed to have easy access to tapping into the processing power of Cellular Automata, if the Nano-people suddenly came up with a new kind of paint, that enabled us to paint Cellular Automata farms on our walls.

I have, during my work on my Master's Thesis, spent some time looking on the web for uses of Cellular Automata and concepts like the ones I have developed during my work. One thing I have realized during this, is that the CA model of computation, is rarely referenced, even when it is obvious to do so. One example of this, could be the Wikipedia description of the Actor Model.[ACMO] The Actor Model very closely resembles cellular automata, although the Actor Model is not as restricted as the Cellular Automata model and the message passing in the CA model is something that is done at a higher level. In my case as higher order cellular automata and in the case of a Turing Machine implemented in Conway's Game of Life, by gliders.[2] There are numerous examples of this lack of reference both on Wikipedia (including concurrency[CONC] and theory of computation[THCO] where Cellular Automata is not even mentioned) and in education material. These are only a few examples, many others could be provided to drive home the point, but what brings forth the lack of cellular automata viewed as a practical processing model, is the sheer absence of references and lack of distinction or definition of what processing model is used. It is almost universally assumed to be a stack based processing model.

If the reader is interested in hands-on-experience or want to have a look at the CAPM compiler or virtual machine, the program and source code `http://www.daimi.au.dk/~u040896/`

[1] Four or five if you count the Quantum Information Processing Model and Logical programming

[2] What the glider is, will be explained later.

mastersthesis/. If a break from reading is needed, I recommend checking out the demo videos of the running CAPM virtual machine at the same url. The demo videos are quite relaxing.

Martin Mortensen

2 Introduction

As cellular automata are rarely described or even mentioned in literature on computation and in computer science as a whole, I will start by giving a very brief historic account of cellular automata and try, by examples, to give a clear picture of what the essence of cellular automata is. When this is done, I will give a thorough overview of my Master's thesis, and label the descriptions and accounts by Aristotle's four causes:

1. Material
2. Formal
3. Efficient
4. Final

The reason why I am going to use Aristotle's four causes is that it is a well tried concept, a couple of thousand years old, and it gives a good classification of the different elements involved in a product.

Definitions of the four causes

1. Material *That from which, <as a constituent> present in it, a thing comes to be ... e.g., the bronze and silver, and their genera, are causes of the statue and the bowl.*[AFC]

2. Formal *The form, i.e., the pattern ... the form is the account of the essence ... and the parts of the account.*[AFC]

3. Efficient *the source of the primary principle of change or stability, e.g., the man who gives advice, the father (of the child). The producer is a cause of the product, and the initiator of the change is a cause of what is changed.*[AFC]

4. Final *Something's end (telos)-i.e., what it is for-is its cause, as health is <the cause> of walking.*[AFC]

The definitions of the four causes given here, might seem quite odd. The main reason for this is that when we normally use the word *cause*, we tend to think of cause and effect. The meaning of the word cause here, is more a way of interpreting a question of what is an object. The question: *What is a table?*, can be interpreted in several ways, and it is the combined descriptions that defines it. If we were to answer only one of the interpretations of the question of *what is a table*, then we would not achieve a full abstraction of what a table is. If someone asked *What is a table?*, then if the answer given, was, as an example, the final cause: *Something you can use for dining, writing and playing cards*, then, in addition to this definition describing absolutely nothing of how a table looks, a lot of other objects could be interpreted as being tables, e.g. a big stone with a flat and smooth top surface.

The four causes of a table, copied from [AFC]:

1. Wood is what the table is **made out of**.
2. Having four legs and a flat top is **what it is to be** a table.
3. A carpenter is what **produces** a table.

4. Eating on and writing on is what a table is **for**.

The point is that an object is not something in itself, it is only something by our interpretation, and thus to describe what we mean by *a table* we must describe several aspects of what *causes* us to see it as a table. There could be an unlimited number of such aspects but through experience, Aristotle's four causes have shown to be sufficient.

Listing Aristotle's four causes for computing principles, is rather difficult because computing is contained in itself. This causes a self-referential structure, resembling circular argument. E.g. Lisp is run in Lisp. Another complication rises from the fact that data in computing can only be used, because a more or less arbitrary interpretation of the data is used to define actions. This fact separates computer processing from any other "mechanical" processing, where the parts are physical objects. This ambiguity of elements in CAPM could lead to many different causes, each being correct. I have tried to list the four causes of CAPM in such a way that the causes give a good framework for understanding CAPM.

The four causes of CAPM

1. Material Cellular Automata and a high level programming language

2. Formal Being able to run, as cellular automata, a parallelized/sequentialized structure which represents the semantics of code written in a high level programming language, **is what it is to be** a Cellular Automata Processing Model.

3. Efficient Code written by a programmer, is parsed and compiled to an Abstract Syntax Tree (AST). This is then used as input to a compiler that outputs Cellular Automata. This is not the highest form of abstraction, but on the other hand it gives a lot of information.

4. Final To use CA as processing model without explicitly coding Cellular Automata is what CAPM is **for**. This enables easy access to execute code so it runs close to as parallel as possible and furthermore give a model that describes how a hardware Cellular Automata Model could look.

In the in-depth description of the causes, I will not list them point-by-point nor will the order be exactly the same. When I describe or account for something that falls into one of the four causes, I will comment it.

2.1 The history of Cellular Automata

The material (the first cause), I am going to use is, among other things, Cellular Automata. The cellular automata will have to be explained first, so that the later arguments and descriptions will make sense.

The concept of cellular automata was created by John Von Neumann in 1940, in an effort to design self-replicating robots. This was, even for Neumann, a difficult task and to avoid the "problems of the real world" he, by the advice of Stanislaw Ulam, turned to a purely theoretical solution of the problem. The result was, in addition to a theoretical solution of the concept he initially sought to create, the Cellular Automata model. An interesting thing here, is that he discovered that the model needed something that in principle was DNA and this happened before the complete identification in biology of DNA as building instructions.

After the work of Neumann, two major things happened. John Conway created a cellular automaton named *Conway's Game of Life*. This very simple, but very powerful, cellular automaton

created a buzz about the Cellular Automata model, but aside from some theoretical results about Conway's Game of Life, e.g. that it was Turing complete, the interest gradually disappeared.

The other thing that happened was that Stephen Wolfram did extensive research in the properties of the simplest of cellular automata; *Elementary Cellular Automata.* This research showed that complexity can indeed arise from simplicity. Elementary Cellular Automata are 1-D cellular automata. Every cell has two neighbors and two states. By a combinatorial approach of rules, there are 256 of such elementary cellular automata. One of these, Rule 110, is proven to be universal by Matthew Cook.[MACO] But even though it is proven to be universal it is very hard to get it to do something meaningful, as is often the case for primitive Cellular Automata.[3]

2.2 Cellular Automata

The cellular automata model is somewhat complex, but fortunately the concept is quite intuitive. Basically we have entities in some internal state, which interacts with its neighbors. The interaction with its neighbors is not an intelligent interaction, but rather an interaction comparable to how the forces of nature (rules) induce a particular behavior of particles (cellular automata). These particles exist in a universe of some sort, e.g. 1 dimension or 11. The formal definition is:

Definition 1 *A cellular automaton (plural: cellular automata) is a discrete model studied in computability theory, mathematics, and theoretical biology. It consists of an infinite, regular grid of cells, each in one of a finite number of states. The grid can be in any finite number of dimensions. Time is also discrete, and the state of a cell at time t is a function of the states of a finite number of cells (called its neighborhood) at time t-1. These neighbors are a selection of cells relative to the specified cell, and do not change. (Though the cell itself may be in its neighborhood, it is not usually considered a neighbor.) Every cell has the same rule for updating, based on the values in this neighborhood. Each time the rules are applied to the whole grid a new generation is created. [WIKICA]*

With the definition in place, I want to give an overview of what knobs and buttons are available, when designing a Cellular Automaton. These are:

1. Environment
2. State
3. Neighborhood definition
4. Rule
5. Configuration

1. Environment The environment defines the world in which the cellular automata exist. This is closely related to the neighborhood definition. The CA environment is more or less a consequence of the neighborhood definition but in addition to this, the environment is also the basic entity of interpretation. As in all other processing models, it is only by interpretation of states or data, computing problems is possible. The data itself mean nothing. In the case of Cellular Automata the environment is often presented visually in grids or meshes.

[3] I do not have anything to support this sentence, other than solving problems with machine code (MC) or even micro instructions (MI) is also quite a tedious task and I would think that these, primitive CA and MC or MI levels, correspond to each other. Primitive CA probably being the most tedious.

2. State The states defines the variety of internal configurations of a Cellular Automaton. This is both used internally and by the neighbors. The state is furthermore another level available to use for interpretation. The number of states is finite.

3. Neighborhood definition The neighborhood definition describes which other Cellular Automata a single Cellular Automaton can react upon, when computing a new state. The neighborhood is defined uniformly for all CAs and is finite. I.e. the neighborhood can not be defined by a regular expression, it can only be defined as a finite set of relative coordinates.

4. Rules The rules define what new state a cellular automaton achieves from one generation to the next. This can be seen as a simple mapping:

$$(stateOfNeighbors, oldState) \rightarrow newState$$

There are no restrictions to the complexity of such a rule, but no lookups outside the neighborhood can be performed and the number of states are finite. Even though the rule is often listed as several rules, it is easy to see that they can be compacted into one rule, by simply combining the rules into one, unless conflicting rules occur. In that case the rules would be faulty. If one wanted to map a given situation into two states, i.e. alive and exception, this should be implemented by having a state, AliveAndException.

5. Configuration The configuration describes the current state of all the Cellular Automata. This is also how a special purpose problem solver is prompted to solve a specific instance of a problem. It can be seen as input data, but as data and behavior are quite intertwined in Cellular Automata, this is not the whole truth. The configuration more or less corresponds to bytecode from a java compiler.

2.2.1 Conway's Game of Life

Conway's Game of Life is a simple set of rules for two-dimensional, two-state, cellular automata inspired by a board game. These very simple rules were simulated visually on computers in the 70's and brought much attention to cellular automata. There are a couple of reasons for this. One is the visualized complexity a very simple rule-based algorithm can evolve into over time, thus emphasizing the truth of Turing's and Gödel's theorems of "one can not, in general, know what a program will do any other way than by running it". Another reason was the clever name that made people think of artificial life and living computers.

States and Rules in Conway's Game of Life The rules of Conway's Game of Life are very simple, they are:

- A cell is in one of two states; alive or dead.
- If a dead cell has exactly three neighbors in the alive state, it becomes alive.
- If a live cell has two or three neighbors in the state alive, it stays alive. Otherwise it dies.

Starting from an initial collective state (the configuration), where some cells are alive and some dead, all cells are updated according to the rules described. This leads to generation two. One of the simplest repetitive and moving forms is the glider[4], which has a 4 generation repetition. The generations can be seen in Figure 1.

[4] This is the one I mentioned in the prologue as being a means of communication if one were to process something in Conway's Game of Life-CA

Conway's game of life, is Turing complete. An implementation of a 1-tape Turing Machine can be seen in [CGLTM]. It is, however, not my intention to use the Conway's game of life, as fundamental principle in my cellular automata. The rules and states of the cellular automata I intend to use turn out to be quite different.

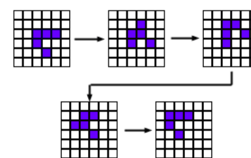


Figure 1: The glider generations. The colored cells are alive, the others are dead.[GLID]

2.2.2 Ground Rules for my solution

To avoid any confusion on what is allowed and what is not, let me set some ground rules.

Restrictions

Self changing The state of a cellular automaton can only be changed by itself. This is important because what it actually says, is that every change happens through some sort of observer model. It also has the implication that only one cell changes the state of a cell and that cell is the cell itself. I.e. every cell knows *No one can change me other than me.*

Static Neighborhood Definition The neighborhood structure, i.e. the other cell-states the single CA reads to evaluate its next state, is statically defined. This means that it is not allowed to do "lookups" according to variables. This has two major impacts. The rule is confined to only looking at a fixed collection of other CAs when a CA is updated. The other impact follows from the first: If the neighborhood is statically defined, then all associations can be implemented in hardware and no dynamic routing or random access mechanisms are needed.

Degrees of freedom

Arbitrary rule complexity The algorithm used to update the single CA can be arbitrarily complex. This is not a modification of the CA definition and therefore not really a *degree of freedom*. The reason why I stress this point anyway, is that my solution uses different logical types of cell entities, e.g. arithmetic and routing operations. This is not in conflict with the CA definition as it can be viewed as a state of the cell to be a certain type and let the rule depend on this *type* state.

Number of states A CA can have a fixed number of states, but in the design phase of the CA there is no cap. I.e. there is no restriction on the data amount I design into the CA. This too is not in conflict with the definition of CA and is only introduced to avoid confusion about the introduction of new logical cells, or cell states, later in the development of CAPM. What matters is that there are no new states introduced at runtime - the size of the *instruction set* is fixed when the program is executed.

Functions as rules This point reiterates that the state of a cell can induce it to act as for example an arithmetic expression, and how the arithmetic expression is actually executed can be abstracted away, without (severe) loss of generality. That there is no (severe) loss of generality can be viewed in two ways. One is that it is the same loss of generality as in what is done in the stack based processing model (SBPM), with the micro instructions, and applying equal standards for the two processing principles, this is acceptable. Another more important view, is that the functionality of the logical types (defined by states) of cellular automata, which are fundamental

in CAPM, could, in theory, be implemented by a lower level cellular automata, probably in a higher dimension. This follows from the fact that Cellular Automata model is Turing Complete.

Grouping into communities The cellular automata can be grouped and these groups can be linked and can communicate. This assumption follows from the lack of infinite space and computational resources and is in my eyes necessary to end up with a solution that is practical and efficient. This point conflicts with the definition of CA in two aspects. One being no different from the lack of *infinite tape* in TM processing models, that the grid in which the CA exist is finite. This is pretty hard to avoid, so I will not try to. The other is purely efficiency related. If the Cellular Automata can only look at their neighbors, then messages or communication between two CA existing far away from each other, would severely impact the performance of the program.

There are cases where such an incident occurs, and although the communication between such two entities probably could be held at a minimum, its impact would be undesirable and as it could be solved quite simple, I propose that at least a description of how to do it is provided. This I do in section 7.

Another reason to seriously consider parting from the CA definition at this point, is a more grave implication of statically defined neighborhoods. Because of the fixed dimension of the grid and each cell in it is of constant size, it is not possible to achieve an exponential *activation*[5] of other cells. Assuming that the code and the logical structure that represents it, is able to perform exponential activation of an exponential branching algorithm, we still have the following problem:

Theorem 1 *If a grid of integer d dimensions is initially configured with 1 active Cellular Automata, then after* $g \in [0, \infty]$ *generations, then the activation can grow both way in all dimensions, meaning a maximum* $O((2g+1)^d)$ *cellular automata can be active. Therefore if a parallel algorithm requires* $f(n)$ *work between each of its g(n) iterations, the running time of the algorithm is increased at least by a factor of* $O(\sqrt[d]{f(n)} * g(n))$.

That the maximum number of active cells, starting from 1 active, is $O((2g+1)^d)$ can be seen from the activity growing in each direction of each orthogonal vector making up the dimensions. This gives a growth rate of $O(2g^d)$. But as we started with one active cell, we are already one step ahead, hence we get the activity growth rate of $O((2g+1)^d)$.

Theorem 2 *If there, in a grid of an integer d dimensions, are a active cellular automata at generation g the increase of active cellular automata from g to g + 1 is at most* $a*3^d$ *- a.*

That the number of new active cells, from one generation to the next, where a cells are active can at most be $a*3^d$ - a, can be seen by using the growth rate function of theorem 1. If we assume that non of the a cells have neighbors in common, then simply by inserting one generation into the formula in theorem 1, subtracting the already a active cells and multiply by a "activation source cells" we get: $a * ((2*1+1)^d - 1) = a * 3^d - a$.

Other parallel models avoid this problem, by dynamic lookups and activation or by special purpose circuits. As CAPM exist in a fixed number of dimensions and all cells have fixed neighborhood, then the balance of iterations vs. work becomes very important, to achieve running times better than sequential algorithms in cases where the parallel solution does a lot of extra work. Examples of this will be shown in chapter 6.

If this problem should be avoided, the solution is simply to organize the Grid of the cellular automata into zones or farms, and have some *wormhole* communication between such zones or farms. These zones or farms could be organized in a way such that the time for a message to travel from a to b would be a polylogarithmic factor of the combined number of Cellular Automata. This would work because it increases the number of dimensions of the grid.

[5]The term activation could be substituted with invocation. The reason why activation is used, is because of the way CAPM runs code by activating it, telling it: run now.

My Master's Thesis There have been numerous writings on Cellular Automata, but many (if not all) of these have been focusing on creating specific CAs for specific tasks.[6] Furthermore much of this writing have had a very theoretical point of view.

I would like to re-iterate my thesis and what I hope to show:

> *Cellular automata as a general processing model can be an alternative to the stack based processing models. The performance of a cellular automata processing model can be efficient and scalable. Furthermore it is possible to create a high level parallel programming language that enables programmers to easily utilize the power of the cellular automata processing model.*

2.3 Cellular Automata Processing Model

This section describes cause number four - final, i.e. the goal. In addition to the description of the Final cause I will also give some motivation for working towards the goal.

One motivation is that CAPM is less restricted by the natural world than what is the case for stack based processing, by the simple reason that it more closely resembles the principles of nature. It furthermore resolves the problem of the Von Neumann Bottleneck[VNBN], by having data and functionality intertwined, existing side-by-side. This is of course countered to some degree by the message passing and probably also by the I/O, which is currently unknown to me how to implement.

2.3.1 High Level Programming Language with a CAPM backend

One of the goals in my Master's thesis is that the use of CAPM is hidden in the same way as assembler often is in typical programming today. The reason for this is that even though special designed *close to the metal* implementation can be considerably faster, than what a compiler can output, the increase in work needed and the error rate, by far outweigh the benefits. Except for some special cases of course. How a CAPM High Level Programming Language would look like, I do not intend to try to answer in this project, but there is undoubtedly a lot of experience to be gained from the parallel computing community. Even though there is experience to be gained from this community, existing and commonly used languages already contain a high level of parallelism and I believe that this built-in parallelism is enough for my prototype. An example of how this built in parallelism already exist, could be the *foreach* control structure in both java and C#. There is nothing in that construct that implies sequentialism, except possible side-effects, which could be conservatively approximated by static analysis.

One thing I think is very important, is that the language is easy to use and as accessible as Java or C#. The language I am going to implement (CATIP) is somewhat high level and at the same time very simple. The CATIP language is meant as nothing more than a way to develop a prototype and hence, not much effort has gone into developing the language to a parallelized environment, therefore no parallel control structures available. The reason for not giving any priority to language development in my project, is that the backend is the important part, and should in principle be able to act as backend for a lot of different language. As an example, it could become a backend to subsets of Java, C++ and others. It is also possible that it could act as backend for functional languages, but this is both beyond the scope of my Master's thesis and to some degree in conflict with the goal of being easy to use.

When I here use the term backend, I mean a grid with a configuration that represents the semantics of the program compiled. I use a 2 dimensional grid where cells look only at their

[6] [CGLTM] is not tailored for a specific problem but it is not practical either. It is proof of complexity, but as it is with Turing machines, we do not use Turing machines for efficient computation.

immediate eight neighbors, when updating their states. There are two levels of updates in the CA. The cell as an entity and a possible logical Cellular Automaton Instruction, e.g. addition. The cell as an entity handles the message passing. How this message passing works will be explained thoroughly later, as it is quite complex. For now just think of it as if a cell tries to pass messages either as a result of an internal operation or because its neighbors have messages to be forwarded through the given cell. The logical state of a Cellular Automaton is defined either as initial configuration of the grid or by a runtime creation of instructions, e.g. if function calls occur. This internal logical state, defines how messages for the cellular automaton are acted upon. This involves primitive operations, conditional reactions, and what messages sent to what logical kin. A couple of ways to picture this duality of the Cellular Automata can be seen in figure 2.3.1

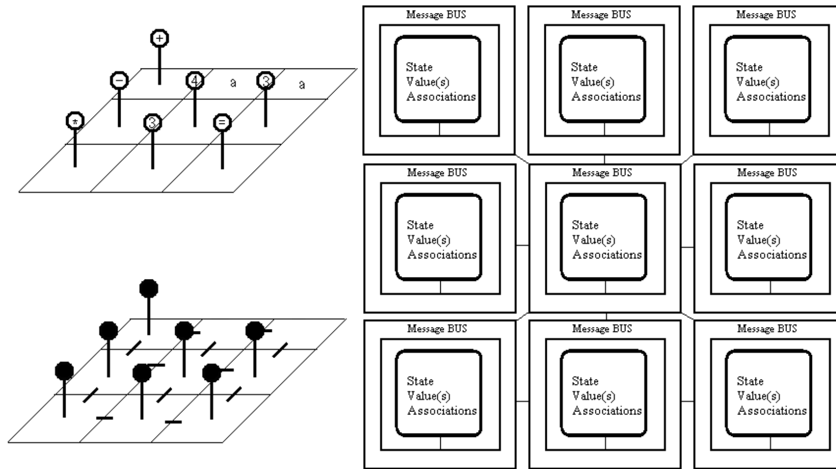


Figure 2: Depictions of how the duality of CA and its logical state (instruction represented) can be imagined. The upper left shows how different logical entities are inside the cellular automata and how two, marked *a*, contain no logical functionality. The lower left shows how the message passing is associated between cells and that the logical state of cell is not accessible from the neighbors. The figure on the right shows the enclosed nature of the logical state combined with the message passing.

The efficient cause, i.e. how the High level programming language material is converted into the cellular automata material which can be executed, is depicted in figure 2.3.1. What actually happens in the generation transitions, will be explained thoroughly later.

2.3.2 Motivation

An initial reason for implementing CAPM is that the processing principle, since it was proposed, have been used more or less as a theoretical model as opposed to a practical model. This motivates the following questions:[7]

1. What could a practical implementation of Cellular Automata as a processing principle look like?
    - What fundamental principles can be used?
    - What properties do cellular automata processing principle have?

[7]Some of the questions can be seen both from the software and the hardware angle.

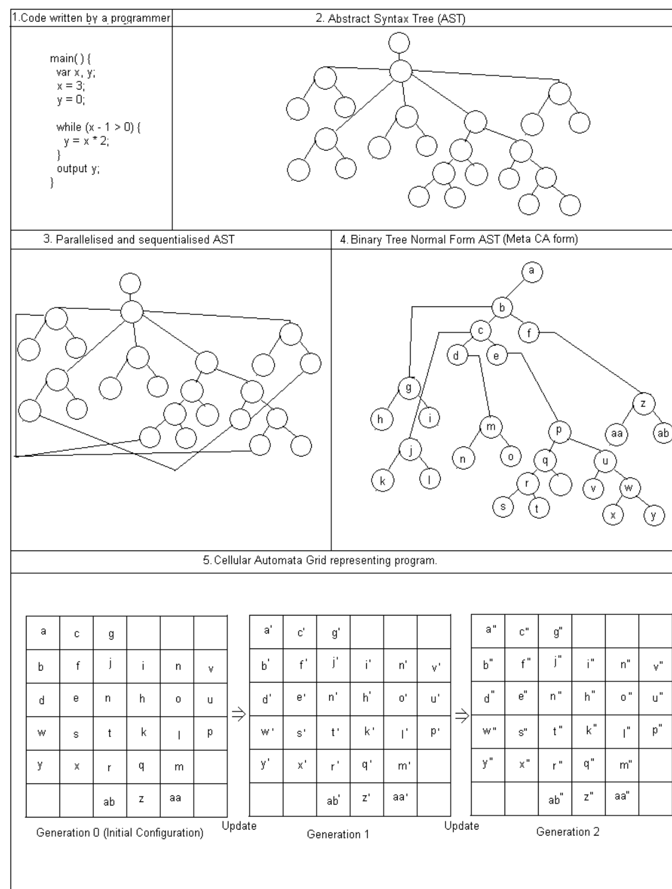


Figure 3: A depiction of the different phases of the program, from the moment the programmer is finished programming(1) and to the time the program is run as cellular automata(5).

2. What are the strengths and weaknesses?
3. What kinds of higher level abstraction tools can be used efficiently (in any sense of the word) in CAPM and what high level abstraction concepts can CAPM be used for.
    - An example of a high level abstraction tool is garbage collection, which CAPM could implement quite efficiently because of the inherent parallelism in the mark-and-sweep algorithm. My proposed association mechanism furthermore implements the *no-one-points-at-me-hence-I-die* algorithm.
    - What parallel tools works well with CAPM?
4. What kinds of higher level abstraction tools can not be used efficiently (in some sense) in CAPM?
    - An example could be that array lookups can not be done in constant time.
5. What computer architectures are possible when memory cells update themselves and reading only is done in static neighborhoods. Could it be possible to create other kinds of processing units?

- As sci-fi examples we could have the question of DNA computers, living cells (e.g. bacteria), light communication, painting cellular automata on the walls when more computing power is needed etc. Furthermore how does the read-only self changing memory affect the number of transistors compared to read all write all memory?

  Communication *Cell interaction can be via electric charge, magnetism, vibration, photons at quantum scales, or any other physically useful means. This can be done in several ways so no wires are needed between any elements. [WIKICA]*

6. What are the economic consequences of such an architecture?
   - Would the mass production of CAPM cells lower the cost of processing power considerably?
   - One prospect is that instead of throwing the old computer out, when a new is acquired, the CAs in the old computer, fits neatly into the new one.
   - One of the prospects in CAPM is the sheer number of processing units. This could mean that a mass production of these units, could be extremely massive.
7. What are the attributes of power consumption for CAPM?
8. What types of algorithms are going to be essential to CAPM and what known problems and phenomena do they relate to?
9. Could concepts in CAPM be used for the robotics industry, in massive agent environments?

Some of the questions listed here, are very hard to give specific answers to, but some circumstantial evidence can be seen from the experiences of GPUs to what the answer to questions on economics and power consumption might be. Assuming that the GPU architecture to some degree represents the attributes of a hardware CAPM and ignoring how many of the CAs updated are actually doing something useful in an update, then the power consumption of computations would be lower in CAPM than in SBPM: *"..This gives us a ratio of 1.5 Gflops/Watt for the GPU and 0,19 Gflops/Watt for the CPU."*[GPUT] Making the same assumptions [GPUT] also has something to say on the economic prospects: *"The parallelization on the GPU (Graphic Processor Unit) gives a power/price advantage 10 to 50 times better to the CPU (Central Processor Unit) solution."*

The document [GPUT] is a press kit, so of course the comparisons mentioned should be taken with a grain of salt, but nevertheless they do give a motivation for the development of CAPM.

I will return to these questions, in the summary. For now I just want to be clear about one thing: The purpose of researching cellular automata is not an expectation of the model being more efficient than other currently used models, at this given moment in time, be it on the GPU, cluster computing or other massively parallel architectures. The purpose is rather to prepare the programming side of a possible future development and hopefully give an option for the future development of computing hardware.

2.3.3 Sequential vs. Parallel

Inherently sequential problems The term *inherently sequential problems* is used to describe problems that are not expected to be parallelizable. An example of such a problem is depth first traversal of a graph, a problem almost defined by its sequentialism. As discouraging as the prospect of having to deal with inherently sequential might seem, I think there is a fallacy in putting too much weight on such problems or restrictions. To underline this fallacy I propose the term *Inherently Parallel Problems* to describe problems that are defined by their parallelism in much the same manner as DFT of graphs is by its sequentialism. A problem I see as inherently

parallel, is Graph Reachability (Or GAP - Graph Accessibility Problem), i.e. in a directed graph, what set of nodes are reachable from a set source nodes.

The reason why I bring up the term of inherent parallel and sequential structure of problems, is to preemptively avoid the whole discussion of *what about the problems that are not parallelizable, does CAPM handle these in an efficient (≈ amount of work) way.* The answer to this question is of course no, but the question in itself is as relevant as this question: *What about the problems that are inherently parallel, does stack based processing handle these in an efficient (time complexity) way?*

This section is not meant to bash the stack based processing principle, nor is it meant to be a defensive speech. The purpose of this section is merely to introduce a discourse, with which not too much time is spent on the matters discussed here. CAPM will not handle all problems faster than a stack based processing principle and it does perform a lot of unnecessary work, but it is by definition unavoidable. The unavoidable extra amount of work is one of the main differences between stack based processing and processing by cellular automata.

Stack Based Processing Model An SBPM tries to do the least amount of work possible, in the smallest amount of space possible. The consequence of this priority is that the time complexity for solving problems will be bounded by the prediction of what work is actually essential. I feel somewhat ambivalent to underline and use the *smallest amount of space possible* goal of SBPM, as this has changed a great deal over time. Today the need for linear DTIME(n) complexity requirements, actually pushes the amount of data being produced to solve a problem upwards. This is enabled by a great increase in RAM capabilities and a decrease in price. So actually this goal is almost reversed, using a lot of space to do as little work as possible.

Cellular Automata Processing Model CAPM is a lot more eager than SBPM. The most vivid picture describing a program being processed by CAPM is the grid being a bus and all passengers/instructions constantly asking their neighboring passengers/instructions *Are we there yet?*. An error in this picture is that they don't ask *Are we there yet?*, but rather *Can I do something now?* The point is that the cells are actively, constantly polling their neighbors to see whether they can pass along a message or checking whether they themselves have received a message, in which case they might act and send one or messages. This means that there is no actual central unit of control. Substituting control with power, what I am actually trying to say, have already been said by the philosopher Michel Foucault, in his analysis of power and control:

> "Omnipresence of power: not at all because it regroups everything under its invincible unity, but because it is produced at every instant, at every point, or moreover in every relation between one point and another. Power is everywhere: not that it engulfs everything, but that it comes from everywhere."[MFOUC]

This has several implications. One is that the program as a whole achieves a new state for each generation passed. A second is, as the control is not centralized, programs do not have, as often is the case, a single point of failure. This I think will lead to a more stable program execution, where some error at one point, an infinite loop or something else, does not crash the entire program. This is of course something that is both a good thing and potentially a very bad thing, as there is the risk of the semantics being changed by such errors. Hopefully such runtime changes of semantics can be avoided. As good as it might seem that there is no single point of failure, this also implies all cells are points of failure, which implies that error correction will have to be handled at several levels in CAPM. And in cases where a Cellular Automata becomes faulty, CAPM will have to react on such things, both with regards to data loss and especially message passing. A last point I want

to make on the *impacts of CAPM* is that I expect that the impact of I/O in I/O-heavy algorithms, if handled correctly, would be less severe than it currently is. This will be discussed later.

SBPM vs. CAPM One could say that SBPM tries to use as little energy as possible and I do believe that SBPM is very energy efficient, at least in principle. At the same time I also believe that the physics of the model taken to its extreme, will (/has) become energy inefficient. This can be seen by long communication paths, the need for active cooling among other things. This is one of the reasons why I think that even though CAPM initially and in theory seems to waste large amounts of energy/work, in the long run, taken to its extreme, it will turn out to use much less energy than initially/intuitively expected, this by structural properties, by schemes involving dynamic hibernation of cells, local variation of clock frequencies according to local activity levels, and/or something completely different.

Another important difference between CAPM and SBPM is that when an SBPM has nothing to do, then it does nothing. This can be seen by monitoring the processor load in your own computer when writing a LaTeX-document and look at how much time the process indicating "System Idle" has actually used. More often than not, this is 90% or more. But what happens when you compile the document you are writing? The CPU is used a 100% and if any other programs are running on the computer (e.g. playing music) these programs are severely affected by one program suddenly wanting a lot of processing being done. This often turns into a frustrating experience. CAPM is also affected by the same problem, but in quite a different way. Updating the grid of Cellular Automata, could be compared to the entire active memory of the computer achieving some new state. This has as a consequence that if there is not a lot of work to do, then a lot of work is wasted (unless handled by some of the means mentioned), but when there is a lot to do, I expect there will be no significant system wide slow-down in processing. I.e. the processing load of one program does not greatly impact other programs. There is of course a slowdown and its impact depend on the distribution of cells, the message passing protocol and how I/O is handled. But nevertheless I expect this to be one of the great differences, from a user point of view, if alternating between a CAPM computer and an SBPM computer.

After all these different potential impacts, both good and bad, have been introduced, I will abstract them away from the solution I am going to implement. I do this because, considering these problems underway, would complicate matters extremely much and because experience and experiments with CAPM will have to be in place before attacking or utilizing the impacts or possibilities mentioned.

2.3.4 Why implement a Cellular Automata Processing Model on the GPU

There are companies[GPUT] specializing in implementing platform independent frameworks that makes the processing power of graphics cards easy to utilize. I do not intend to compete with such solutions and it is important to note, that the reason why I would like to implement CAPM on the GPU is primarily because it is a way of comparing theoretical results with practical implementations, executed on a hardware processing model which lies, conceptually, closer to a Cellular Automata machine than the CPU.

In addition to the theory vs. practice question, I believe that GPUs could be used to develop languages, techniques and optimizations until CA-machines are available and furthermore indicate what the performance of such CA GPU virtual machines are.

In other words, the question is not whether CAPM is the most efficient solution for parallel processing on the GPU, but rather if it is practical and usable already today for prototyping and empirical studies?

Because of the time available to write a Master's Thesis, I am not able to implement this part, but I will describe how such a solution could be implemented and what the prospects of such an implementation would be.

2.3.5 Similar parallel languages and principles

In this section I will sketch some similarities and differences between CAPM and two more or less similar parallel processing principles. The purpose of this comparison is to see what common attributes CAPM has and what makes it different. This can be used to identify what performance differences should be expected and what ground rules should change for CAPM to achieve the same performance.

Turing's Desert The *Desert Turing Machine*[8] (DesTM) is a term describing an Alternating Turing Machine (AST). The AST has the following definition:

Alternating Turing Machine An ATM starts out as a single TM. At any point in time this TM can split in O(1) copies of itself. It dies but leaves a boolean gate describing how to combine the sub results.

The Desert Turing Machine metaphor extends the ATM description, as a circuit building its silicone components of the desert in which it lives, when it needs more computational power. This desert is more or less the grid in CAPM. The logical entities being the spawning Turing Machines. Even though CAPM does not yet support recursive functions, the Desert Turing Machine is a descriptive image of what such an extension of CAPM would look like.

The differences between the DesTM and CAPM The main difference between the DesTM and CAPM is that, in CAPM the place a logical entity or component can "create itself", is in an empty cell, in other words: *In CAPM the desert is structured.* Because the neighborhood structure of CAPM introduces a restriction of how many neighbors are empty, that *can be used to process something*, and in general the dimensional polynomial slowdown created by it, the DesTM can achieve much better performance because no restricting assumptions are made with regards to space and communication between processing units.

The similarities between DesTM and CAPM Not knowing yet how the recursive mechanisms in CAPM would look, makes it difficult to describe what the similarities between DesTM and CAPM are, as that is the main similarity between the two models. This similarity, as described in the differences, exists only at the logical level. The other similarity is purely the imagery of the models, i.e. that in the DesTM the processing units build themselves out of sand and in CAPM the logical operations find a place to process.

Actor Model The Actor Model is a very good description of how the logical nodes in CAPM works.

Description of the Actor Model copied from [ACMO] The Actor model adopts the philosophy that everything is an actor. This is similar to *the everything is an object* philosophy used by some object-oriented programming languages, but differs in the aspect that object-oriented software is typically executed sequentially, while the Actor model is inherently concurrent.

An actor is a computational entity that, in response to a message it receives, can concurrently:

[8] Coined by Peter Bro Miltersen [BROMILLE]

- send a finite number of messages to other actors.
- create a finite number of new actors.
- designate the behavior to be used for the next message it receives.

There is no assumed sequence to the above actions and they could be carried out in parallel.

Communications among actors occur asynchronously : that is, the sending actor does not wait until the message is received before proceeding with computation.

Recipients of messages are identified by address, sometimes called "mailing address". Thus an actor can only communicate with actors whose addresses it has. It can obtain those from a message it receives, or if the address is for an actor it just created.

The Actor model is characterized by inherent concurrency of computation within and among actors, dynamic creation of actors, inclusion of actor addresses in messages, and interaction only through direct asynchronous message passing with no restriction on message arrival order.

Reducing the Actor Model to meta-CAPM This description could, with a few modifications, be used to describe the attributes of the logical entities in CAPM.[9] The logical entities in CAPM is basically the instruction set of CAPM, but they act much like actors, in the sense that they:

- Can have logical relatives; parent, children and a non-identical twin. When a message is received, zero or more messages can be sent to zero or more relatives.
- Currently CAPM does not support the creation of new logical entities, but the goal to let pieces of code (an associated set of logical entities) spawn or copy themselves, so that recursive processing and persistence[10] of data can be performed.
- When receiving a message, the behavior of a logical entity in CAPM depends on what other messages have been received in the past. Because the logical entities are of constant size which should be minimized, this behavior has a very limited number of states.

A similar reduction from the Actor Model to the other elements of CAPM could be done, but the important similarity here, is the meta/logical abstraction level of CAPM.

The differences between Actor Model and CAPM The differences between the actor model and CAPM arise because CAPM is a special case of the actor model, a very restricted special case. The restrictions in CAPM have been described in section 2.2.2. Looking past the neighborhood restriction, which is not relevant at this abstraction, the logical entities in CAPM are restricted to have *mailing addresses* of at most four associated logical neighbors. The computation aspect is another serious restriction. If we ignore the state maintenance and message passing, then because the logical entities in CAPM can only perform one computation of complexity equal to an assembler instruction, it should not be looked upon as constantly processing, rather it only processes when the right messages have been received.

The similarities between Actor Model and CAPM The main similarity between CAPM and the Actor Model is the philosophy; That every single entity is an active agent reacting on its environment. The other similarities are, as mentioned above, caused by special case vs. generalized case.

[9]The logical entities represent a meta cellular automata layer in CAPM. These are cellular automata, without the neighborhood restriction. This could be compared to the intermediate AST in compilers and static analysis.

[10]It is currently unknown whether or not other persistence than output are needed in CAPM. In section 5 it is shown that the concept of variables more or less disappears in CAPM. Whether this will also be the case when introducing pointers remains to be shown.

Quirks of CAPM The quirks of CAPM, compared to processing principles with similar goals, are mainly the very restrictive nature of the Cellular Automata definition. Most of the parallel processing models commonly mentioned, have as a success criteria that they can perform exponential speed-ups, at least for some problems. This is much like the polynomial speed-up gained in Turing Machines, by introduction of Random Access Memory. The neighborhood restriction in CAPM, makes it impossible to gain the exponential speed-up, achieved by other parallel processing principles.

If one was to allow the neighborhood restriction to be ignored, then CAPM too can achieve exponential speed-ups by simply replacing the message passing rule. This will be explored in section 5.6 and chapter 6. The reason why I do not ignore the neighborhood restriction, is that the difference between CAPM and parallel processing models that try to achieve exponential speed-up, is the expected gains by having all communication occur very locally, which potentially increases the engineering possibilities of communication, increases the potential clock frequency, decreases the power consumption and energy loss. See section 2.3.2 for other reasons.

Even though I want to respect the neighborhood restriction, to enable the possibilities described in section 2.3.2, I intend to make it possible to run CAPM with exponential speed-ups available by making the message passing protocol an easily replaceable component. The structure of the grid is another element that should exist more or less as a component, such that if a structure is shown to be highly efficient and scalable then it should be possible to have the Cellular Automata in that structure.

To enable the goals of the above, another requirement must be met. If a problem is implemented in a fashion such that logarithmic processing time is achieved, then the structure of the logical Cellular Automata should only restrict that potential speed, within a constant factor. In other words, the logical structure of CAPM should support exponential activation such that the only restricting factor is the speed of the message passing protocol being used in a certain grid structure.

3 Abstractions and design of CAPM

The abstractions I have made during the development of CAPM, have primarily had to do with creating an easy way to talk about and document different aspects of the problems involved. Some of these abstractions may be trivial, others may border the limits of what I described in the Ground Rules (2.2.2) for my implementation. The danger of abstracting sub problems away is, as always, whether some important, fundamental problem is ignored. I am quite certain that the abstractions I have made, does not do this, and in chapters 6 and 7 I will discuss if such problems exist and the areas where they do exist.

3.1 CAPM Principles

I will in this section describe what the principles of my implementation of CAPM are. One early decision I made, was to represent the logical structure of programs in CAPM by instructions in binary trees. This was a design choice and I could have chosen otherwise. The logical structure could have been represented as form of matrix multiplication, where the multiplication was the update of the grid as a whole or overlapping. But as I expected that this would require rethinking a lot of processing principles and start me off in a too unknown direction (so that I might have ended up not implementing anything), I chose the binary trees as fundamental logical structure. The motivation for this choice and exactly how it is implemented, is explained in the descriptions of the CAPM principles.

Another choice was, as mentioned earlier, to use message passing to avoid ending up in problems that could turn out to be unsolvable or NP hard.

The choice of having the cellular automata living in a 2-dimensional grid, was made for a couple of reasons, having to do with latency and complexity of message passing and that it is easy to visualize. I could have created a 3 or higher dimensional grid, but first of all, it would complicate the solution. Secondly it would not be easy to implement a hardware version of a 128-dimension grid in the real world. The reason why I did not implement CAPM as a 1 dimensional CA was that I was not sure that such an implementation would illuminate all the problems involved in the creation of a Cellular Automata Processing Model. This is certainly the case for the message passing protocol, which in 1 dimension is trivial and in 2 dimensions or higher, somewhat complex.

3.1.1 Parser and compiler

The parser and compiler of CAPM are no different than *normal* parsers and compilers, except for the two important aspects, sequentialization and binary tree normalization and balancing. These two aspects are handled at the time where a compiler would normally output the resulting assembler-code of the compiled program. How sequentialization is handled, both compile time and runtime, will be explained in detail in 5.3 and how to translate all associations to binary trees and balance them to Binary Tree Normal Form (BTNF), will be handled in 5.4.

In addition to these two requirements, a last optimizing requirement is that the outputted cellular automata, is put into the grid in a way, such that the distances between associated cellular automata are minimized. How to rank the different while loops etc. is of course undecidable, as it depends on the runtime behavior of the program. In addition to this I expect the conservative approximation to be NP-hard, in which case the conservative approximation will probably have to be approximated. This approximation should try to give repetitive structures a higher priority to close proximity than the ones only evaluated once. This is without doubt a research area on its own and will not be covered in this master's thesis.

3.1.2 Meta Cellular Automata Layer Instruction Set - MCALIS

The MCALIS layer graph is an intermediate representation of the program, the meta prefix is introduced because the MCALIS nodes are not represented in a grid and does not have a statically defined neighborhood. MCALIS nodes are associated by references to other MCALIS nodes.

Some of the of MCALIS node classes, e.g. assignments, while and others, represent entities of higher abstraction. These higher level entities are binary trees, the functionality being based on knowledge of the structure of the tree, the MCALIS_Assignment node knows that the target of the assignment is the right child of the left child of the root, but when we want to set the target, we simply call the function SetTarget(MCALIS_Node node).

What the MCALIS nodes represent is (in addition to the higher level entities) the complete instruction set of CAPM. E.g. if CAPM had a modulo operator, then there would have to be a MCALIS_Modulo node.

3.1.3 Cellular Automata as binary trees

I have not tried to implement the Cellular Automata *close to the metal* as it is expected that such a solution would needlessly complicate both the design and implementation of a CAPM. Instead I have sought a solution which is both intuitive and could be expected to be implemented efficiently. If this turns out to be successful, a deconstructionistic downward/reverse engineering approach would be the next step, i.e. translating the CA algorithm to a set of simpler Cellular Automata, until a satisfactory level of abstraction *close to the metal* is achieved. The Cellular Automata have a logical structure, best described as a Binary Tree version of a linked list. I.e. the nodes in the binary trees can have one association with another node in another tree, their data is their association. The binary tree solution ensures that the size of the logical types of cellular automata is fixed and thereby maintains the constant size restriction of the Cellular Automata rules. I could, of course, have had trees with one-thousand-children-nodes, but I guess that the deconstruction would be quite complex in such cases. This logical structure is implemented implicit in the CAs of the grid, by knowing the addresses of their kin (parent, children and outer association). This distinction of the logical and physical structure of cellular automata is equal to the physical address of elements in a linked list being irrelevant to the logical placement of the elements in it.

Using binary trees as fundamental logical structure of logical CAs is a design decision but I see this solution as a balanced tradeoff between:

- Minimal Data structure
- Fast activation - A short shortest path between main function and all other nodes
- Intuitive structure of code
- Generic principle
- Support reusability

Binary trees are for data structures what the bit is for information; the smallest unit of data with which efficient combinatory are possible. To say it another way, binary trees are for lists what binary numbers are for unary numbers. I have put in an extra association in reference nodes, the linker, which is an association to another node in another tree, one might call it a pointer. This is to avoid the need to create one binary tree, which describes the entire program, as this could be as difficult as the problem I am trying to avoid by introducing binary trees. The linker is not another child, as it has another parent. Just by adding the linker, the problem of translating code or ASTs to cellular automata in binary trees, become considerably simpler. The principles in the structure are:

1. Statements or other higher order structures are embodied by their root. The right subtree of the root consist of ingoing references. The left are outgoing references or the functionality associated by the statement or higher order structure.

2. Arithmetic or logical expressions, always have an ancestor that makes sure that the result is propagated to the correct receiver(s).

3. References, both ingoing and outgoing, are the primary source of control flow in CAPM. There are several other different types of references, but the common attribute is that they do not embody any functionality, other than manipulating and controlling the control flow.

An example of principle 1 An instruction called NumberGenerator, that generates numbers, used by others. These numbers could follow some arbitrary rule, but for simplicity let's say when it is asked the i'th time, it returns i. There could be several other nodes that used this number for some arbitrary reason, for example customer IDs. This NumberGenerator would have a form looking like figure 4, where all ingoing references are in the right subtree of the root. Entities like this, where some central form of control have to exist, are simply handled by having ingoing references, that acts as a request buffer, a single point of entry - the root, and a function generating the desired output - the left subtree. This structure is used for many different purposes, and is, in my experience, a good way to structure the binary trees.

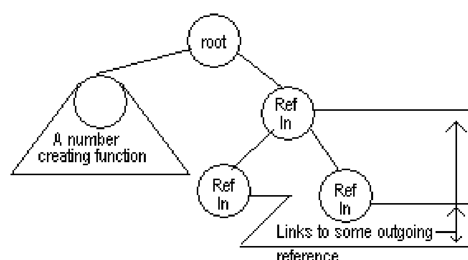


Figure 4: The binary tree for a NumberGenerator

An example of principle 2 The assignment has, at least on the surface, a very close resemblance with a binary tree, the '=' being the root. In CAPM statements are associated with a block, by an ingoing reference as right child of the assignment statement. This has as a consequence that we have to push the assignment and the expression down the left subtree. This leads to the structure of assignment statements depicted in figure 5. The structure of the assign statement, is probably the most used in CAPM. It more or less acts as a template for all statements, including *while* and *if*, although they are a bit more complicated.

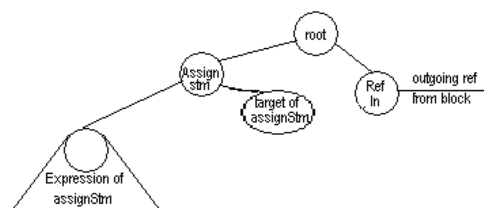


Figure 5: The binary tree for an Assignment Statement

Binary trees can be packed compactly into 2D grids [BITREE], which means that the compiler output can be made to reduce the communication time considerably. Unfortunately packing different sized squares into one large square is a complex problem, (I would expect it to be NP hard as the sorting criteria is the sorting itself) and is furthermore complicated by the fact that new associations will occur at runtime, so it would be needed to keep these binary tree boxes updated at runtime. For now, let's just say that the compiler outputs closely related cellular automata close to each other in the grid, thereby minimizing the communication time.

3.1.4 MCALIS to CALIS

When the MCALIS nodes have been created and different optimizations have been made, they must, if we want to run the program, be translated into Cellular Automata Layer Instruction Set (CALIS) nodes. The CALIS nodes are the logical runtime Cellular Automata and when a CALIS node sends messages to or receives messages from its kin, it does so purely by message passing. To avoid complexity of the update algorithm, a CALIS node communicates through another cellular automaton, which interacts with the statically defined neighborhood. This *other* cellular automaton is of course not another type of cellular automaton. It is in fact the real cellular automaton which is updated according to a rule that makes it act as the logical operation, that it currently implements.

3.1.5 Message passing protocol

One of the requirements to a community of Cellular Automata is that the neighborhood of each Cellular Automaton is uniformly statically defined. I.e. no dynamic lookups can be performed. Hence if it is not possible to lookup a neighbor defined by a parameter or calculation, a generic model to compute in this way would need a fundamental paradigm shift in the way we think of algorithms. I believe that this need for a paradigm shift, is one element in what has stopped the use of cellular automata from being a used processing model and this is the main reason why I use message passing. Using message passing is without loss of generality. This I claim because in CA implementations of Turing machines, gliders and others are actually being used as an internal message passing mechanism.

I will not yet go into how the Message Passing Protocol (MPP) works and I have in the Java prototype development of the different aspects of CAPM used a protocol I call DirectRouting. The interface between the MPP and the Cellular Automata in the grid is handled in a way, such that the MPP rule can be substituted with another, without changing any code in the update method of the Cellular Automata. This is done to ensure that MPPs can be easily compared and hopefully a standard of how to code them can be created, either by a Mark-up language, by implementing the CAL_MessagePassingProtocol interface or something completely different.

The DirectRouting protocol, ignores the restriction of the static neighborhood definition, and simply works by looking at a CALIS_Node and depending on whether it itself contains a message either tries to read a message form one of the neighbors or checks whether the message it contains, has been read by the logical neighbor targeted by the message.

An MPP developed later in the course of my master's thesis is the Simple Cellular Automata Message Passing protocol (SCAMP-protocol) which obeys the static and uniformly defined neighborhood restriction. This MPP will be described in detail in section 5.6.

3.2 Time to run the Runtime CAs

The cellular automata processing model is, in essence, the universal machine that processes the cellular automata. I have implemented this as a virtual machine in Java, where the 2-dimensional grid updates the cellular automata. The Cellular Automata in the grid are updated by what can be seen as a double buffered memory structure. The new state of a CA in CAPM depends on the old state of the CA and the "old" states of its neighbors. A depiction of this can be seen in figure 6 where the neighborhood structure are the horizontal and vertical immediate neighbors.

The algorithm consists of two "simple" steps:

1. Update top layer, based on bottom layer.

2. Update bottom layer, by copying upper layer.

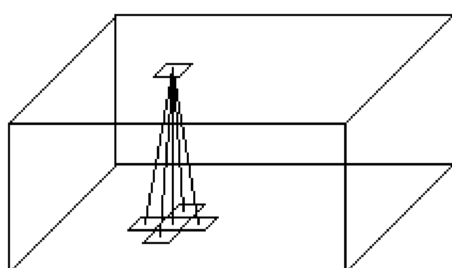


Figure 6: The double buffered principle used in the update.

It is important to remember here, that the cellular automata in CAPM have two kinds of behaviors. These two can be called physical behavior and logical behavior. The physical behavior has to do with the interaction with neighbors and message passing. The logical behavior has to do with the logical CALIS node, currently represented by or contained in a physical cellular automaton. So what actually happens, is that a cellular automaton acts on two different sets of neighbors: The physical neighbors and the logical neighbors. In my implementation of the CAPM Virtual Machine (CAPM-VM) in Java, I have made this point explicit by having the grid consist of a matrix of CAL_CellularAutomaton which represents the physical part of the cellular automata, and enable the CAL_CellularAutomaton objects to contain a CALIS_Node. Exactly how this is implemented, will be handled later in 5.7.

3.3 Designing a simple example - SACAM

I will now sketch a very simple and not too practical language. The purpose of this very simple example is to introduce some of the basic properties and inner workings of CAPM. The terms and concepts introduced are:

- Control flow
- Message passing
- State information and how they are reacted upon
- How to control the initial configuration
- Types of nodes

3.3.1 Simple Arithmetic Cellular Automata Model

Simple Arithmetic Cellular Automata Model (SACAM) is, as the name implies, quite simple. It is meant as nothing more than an example which helps to explain the design and architecture of CAPM. SACAM consists of a main block, output statements and constant valued arithmetic expressions. The output statements are for the sake of simplicity handled sequentially. Another approach could have been to handle the arithmetic expressions in parallel by putting the results into variables and only handle the output statements sequentially.

It would be simple to create a language to code it in, but to avoid any unnecessary layers, the code is produced by building structures of Meta cellular automata, which will be compiled to

runnable cellular automata.[11]

Context free grammar for SACAM There is a small grammar in table 1 describing the language. As it is only an example language, the programmer will have to obey the rules when creating the meta cellular automata.

program	$\rightarrow$	*block*
block	$\rightarrow$	*stm*∗
stm	$\rightarrow$	output exp
exp	$\rightarrow$	*intconst*
	$\rightarrow$	(exp)
	$\rightarrow$	exp binop exp
		where binop $\in \{+, -, *, /, ==, >\}$

Table 1: Context Free Grammar for SACAM

Let us take a walkthrough example to get a feeling of how the principles in CAPM works. The code example will, when run, output 7 and 1. To do this the programmer must translate the code he/she wants, into a structure of Meta CA nodes. The code needed to produce the structure depicted in figure 3.3.1 on page 30 is shown in table 3 on page 29. The associations of the nodes have the intuitive meaning of:

- Parent association is the line going from the top of the node.
- Left child association is the line going from the lower left side of the node.
- Right child association is the line going from the lower right side of the node.
- Linker association is the horizontal line going from the from the vertical center of the node.

In the paragraph at page 28 and on I have listed a detailed step-by-step description, to document the control flow.

```
1 {
2   output 4 + 3
3   output 4 == (3 + 1)
4 }
```

Table 2: Very simple SACAM code

The step by step behavior of the example program

1. The block starts with an evaluate message bound for its left child.
2. The outgoing reference below the block receives the EvaluateMsg. This causes it to send an EvaluateMsg to its linker.(the horizontal association)
3. The LTR (LeftThenRight) node receives an EvaluateMsg from its linker. This causes it to send an EvaluateMsg to its left child.
4. The ingoing reference of the first output statement receives the EvaluateMsg from the left child of the LTR node, it sends an EvaluateMsg to its parent; the Assignment node.

[11]There is no SACAM virtual machine. But as constant folding on the code written would work wonders, it has no practical relevance.

```
1 MCALIS_StatementCollection block = new MCALIS_StatementCollection();
2 MCALIS_Output output = new MCALIS_Output();
3 MCALIS_ConstInt firstOutputConst4 = new MCALIS_ConstInt(4);
4 MCALIS_ConstInt firstOutputConst3 = new MCALIS_ConstInt(3);
5 MCALIS_AdditionExp firstOutputAddition = new MCALIS_AdditionExp();
6 firstOutputAddition.SetLeftChild(firstOutputConst4);
7 firstOutputAddition.SetRightChild(firstOutputConst3);
8 MCALIS_AssignmentExpression firstOutput = new
     MCALIS_AssignmentExpression();
9 firstOutput.SetTarget(output);
10 firstOutput.SetEvaluateExpression(firstOutputAddition);
11
12 MCALIS_AssignmentExpression secondOutput = new
     MCALIS_AssignmentExpression();
13 MCALIS_ConstInt secondOutputConst4 = new MCALIS_ConstInt(4);
14 MCALIS_ConstInt secondOutputConst3 = new MCALIS_ConstInt(3);
15 MCALIS_ConstInt secondOutputConst1 = new MCALIS_ConstInt(1);
16 MCALIS_AdditionExp secondOutputAddition = new MCALIS_AdditionExp();
17 secondOutputAddition.SetLeftChild(secondOutputConst3);
18 secondOutputAddition.SetRightChild(secondOutputConst1);
19 MCALIS_EqualExp equalExp = new MCALIS_EqualExp();
20 equalExp.SetLeftChild(secondOutputConst4);
21 equalExp.SetRightChild(secondOutputAddition);
22 MCALIS_AssignmentExpression secondOutput = new
     MCALIS_AssignmentExpression();
23 secondOutput.SetTarget(output);
24 secondOutput.SetEvaluateExpression(equalExp);
25
26 block.AddElement(firstOutput);
27 block.AddElement(secondOutput);
28
29 //Set an evaluate message going to the left child of block when the
     MCALIS nodes created above have been translated.
```

Table 3: Code to create the simple SACAM example

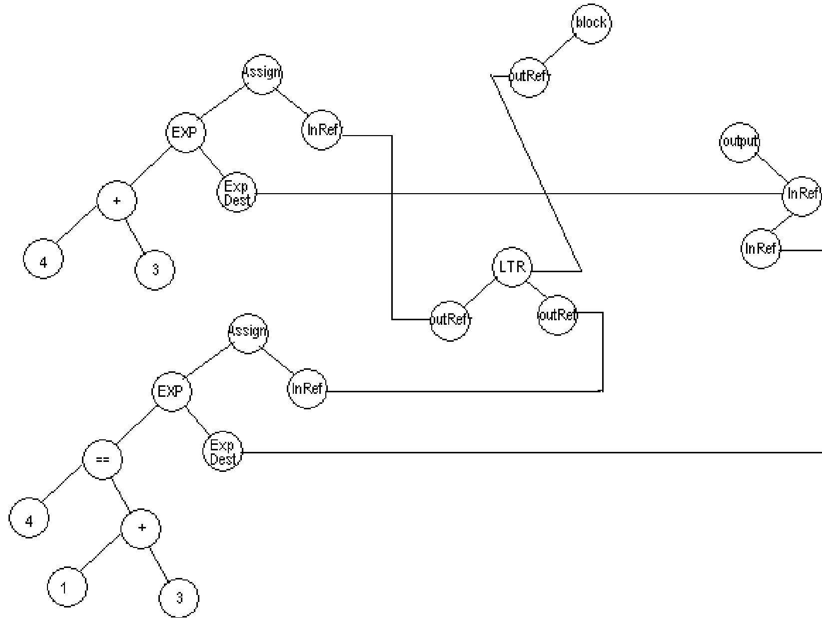


Figure 7: The graph implementing the semantics of the simple SACAM example

5. When the assignment node receives the EvaluateMsg from its right child, it sends an EvaluateMsg to its left child.

6. When the expression node receives an EvaluateMsg from its parent, it sends an EvaluateMsg to its left child. (it actually acts much like an LTR node)

7. The Addition node receives an EvaluateMsg and sends Evaluate to both of its children.

8. When the integer constants receives an EvaluateMsg they reply with a message of type EvaluationComplete with their value attached.

9. When the addition node has received EvaluationComplete from both its children it will add the two numbers received and send and EvaluationComplete msg to its parent with the result.

10. The Expression receives an EvaluationComplete from its left child, and sends an Evaluate with the value of the message received from the left child, to the right child, i.e. Expression Destination.

11. When the ExpDest node receives an EvaluateMsg from its parent, it sends an Assign msg with the value of the received message, to its linker.

12. The ingoing reference of the output node, which receives the assign message sends an Assign message up through the tree, in this case, the distance is 0, so it sends the message directly to the Output node.

13. When the output node has outputted the value of the assign message received, it returns an EvaluationComplete to its right child.

14. The ingoing reference receives the EvaluationCompleteMsg and sends EvaluationCompleteMsg to its linker, i.e. the ExpDest node. [12]

15. The ExpDest node receives an EvaluationCompleteMsg from its linker and sends Evaluation-Complete upwards.

16. The Exp receives EvaluationComplete and sends EvaluationComplete to its parent.

17. When the parent, i.e. the statement, receives EvaluationComplete, it routes this information back to the ingoing reference who initiated it. (quite easy in this case as it only has one..)

18. The LTR node receives EvaluationComplete from its left child, and sends an EvaluateMsg to its right child.

19. The second statement acts in the same as the first, the only difference or new thing is that the equality comparator returns 1 if the arguments are equal, 0 otherwise.

20. When the right child of the LTR node sends an EvaluationCompleteMsg, the LTR node sends an EvaluationCompleteMsg to its linker.

21. The outgoing reference of the block receives EvaluationComplete. If it had had any children it would wait for those to also send EvaluationComplete, but in this simple example it does not, and therefore sends an EvaluationCompleteMsg to its parent.

22. When the block receives EvalutionComplete the program is finished.

Is this an adequate example? Even though this example is very simple, it adequately illustrates how the control flow works. It furthermore introduces the basic node types in CAPM and how they work. The message passing, i.e. how send and receive are handled, is left out for now. In section 5 a detailed description of this is given. For now just imagine that messages are sent instantly to their destinations and when a message is received the right action is taken.

Control flow The control flow in the execution of a program can be compared to function calling. The main difference is that here it is an instruction that invokes zero to three other instructions. To emphasize this difference, I will name it instruction calling. All these instruction calls must receive answers from all the calls they made, before returning their own answer. This basic principle handles a lot of problems associated with parallel programming in an easy way. It is used to identify when code has finished computing, read and write values, and later certain other things.

If I should compare it to assembler, then the control flow can be seen as a multidimensional stack of instructions.

Node types The different node types in the example, differs much in how sequential or parallel they are. A good example of a strictly sequential node, is the LeftThenRight node, which embodies the principle that something must be completed before something else begins. The ingoing reference, although not seen in this example, is also strictly sequential. The binary arithmetic expressions and outgoing references are perfect examples of strictly parallel nodes. They send messages to several other nodes, and thus as the answers can arrive in any order, this have to be handled by their logical reactions to these answers. The sequential vs. parallel node types, is an abstraction that can be used to describe what a node does. The nodes used in this example are the fundamental building blocks of CAPM and will, with a few more types, be enough to create

[12] If the target was further down the tree some routing would be performed.

if- and while-statements. The instruction set will be extended later, but the extensions basically work in much the same way as the principles described here.

When reading the detailed list of what happens, during the execution of the simple example, it is possible that it can be interpreted as being somewhat sequential in behavior. This is truly not the case. First of all, there is some parallelism in the evaluation of the arithmetic expressions. If the example had had 2 + 2 == 1 + 4, then the 4 + 4 would be computed in parallel with 1 + 4. Secondly all the nodes act very eagerly, all get updated all the time. The list only contains the interesting events, not all the *Have I received any messages and do I have to do something now? No.*-updates, which is what happens in all the other nodes.[13] This simple example does not conform to any strict neighborhood definition. This will be handled later, when the message passing protocol is explained in section 5.6.

3.4 A theoretical benchmarking and comparison

There are several important ways in which a processing model can be benchmarked. In addition to the different benchmarks, the performance can be compared to parallel models and sequential models, each of these performed in several ways.

Analyzing Parallel algorithms When analyzing parallel algorithms, the performance[14] is evaluated by two factors. The first is of course time, but it is a bit different than what we usually identify as time. When analyzing the performance of sequential algorithms, work and time are more or less interchangeable. This is not the case when analyzing parallel algorithms. The asymptotic time complexity of a parallel algorithm is measured in the amount of iterations of some parallel action. The work is equal to the work required to perform the parallel action multiplied by the iteration amount. To illustrate this, just think of how bubblesort works. If one wanted to perform a bubblesort of a list of n integers, it could be done by letting each element at each index look at its neighbors. This algorithm is a bit more complex than the sequential; it has two phases, one looking forward and one looking backward, but the work needed to perform one iteration would be $O(n)$. The amount of iterations needed would in worst case be $O(n)$ as this is the amount of steps it would take the last element to move down to index 0. So to summarize: paraBubble of n elements takes $O(\frac{n^2}{n})$ where the O(work) is the numerator and O(iterations) is the denominator.

CAPM vs. Parallel To get a feeling of what can be expected from CAPM I will introduce some comparisons between the CAPM principle and known algorithms and complexities for inherently parallel problems. The reason for analyzing how CAPM handles inherently parallel problems, is to get a sense of what the peak performance of CAPM is. As mentioned in Degrees of freedom in 2.2.2 CAPM is not able to perform exponentially branching algorithms, only polynomial activation is possible. This also means that the fastest CAPM can handle a problem, ignoring message passing latency, is some polynomial of the input. Whether this polynomial is lower, equal or above linear performance will be investigated for different problems and solutions.

In addition to benchmarking CAPM with parallel algorithms, it is also important to compare it to how other parallel architectures handle these problems. What architecture to compare with depends on what problem is being analyzed.

The results in section 6 show that the logical structure of CAPM is able to achieve logarithmic time bounds on matrix multiplication, if the message passing is performed with optimal speeds (i.e. no strict neighborhood).

[13] I.e. the bus example
[14] The asymptotic performance.

CAPM vs. Sequential In addition to the benchmark analyzing the performance of CAPM by parallel problems and comparing CAPM with other parallel processing structures, a TIP-comparative benchmark will be introduced later. The TIP-comparative benchmark will be used to make empirical comparisons between the same piece of code processed by a TIP-SBPM and by TIP-CAPM. As CAPM can not achieve the very low logarithmic boundaries typical for parallel processing, this comparison is more important and interesting than what one might have assumed at first glance.

This comparison will be handled in several different ways, but primarily involve the number of changes to the stack in the TIP-SBPM evaluation of the code and the number of generations and CALIS nodes in the TIP-CAPM evaluation of the code.

The results in section 6 show that CAPM is, when scaled, able to outperform the SBPM computation of matrix multiplications.

4 A tiny example language

I have already mentioned that I believe a simple imperative programming language, if used correctly, contains enough parallelism to be efficiently processed by a CAPM backend. To avoid using too much time on developing a compiler, I chose early in the process to use an example language used in the Static Analysis Course[STATIC] at Daimi[15]. This choice supplied me with a SableCC file as parser, a compiler where different static analysis tools and concepts were implemented[16], and last but not least an SBPM virtual machine to process the compiled code. This saved me a lot of work and furthermore enabled me to do some good comparative tests.

As the current version is a prototype, I have not changed the Context Free Grammar, accepted by the SableCC grammar, to only accept the subset of the imperative programming language the CAPM backend currently supports. The reason for not restricting the set of programs accepted is prototyping and removing the need to reinstate parts of the syntax if future essential extensions were implemented. In short, input programs are assumed to only use the subset of TIP supported by CAPM. If one were to use other features, I grant no guarantees of what will happen.

4.1 Cellular Automata Tiny Imperative Programming Language

The reason why I have used an existing language is, in addition to the reasons already mentioned, that I would try to find out how difficult it would be to implement a CAPM backend of an existing language. By not focusing on the frontend or programming language, I have tried to avoid locking CAPM to a particular domain specific language and achieve the goal of creating a generic cellular automata processing model supporting general purpose programming languages. Although the current version lacks many of the fundamental functionalities needed to support a general purpose programming language, I expect that these functionalities can be achieved and offer conceptual solutions to the implementation of them in section 7.

4.2 Tiny Imperative Programming Language

The Tiny Imperative Programming (TIP) language is, as mentioned, an example language used in the Static Analysis course at Daimi. The TIP language is created to be as simple as possible and yet contain all the complex problems existing in larger imperative programming languages. I will not go into the details of TIP here, an overview of the TIP base and the CATIP base should suffice.

The TIP base:

- Main function
- Arithmetic operators
  - addition, subtraction, division and multiplication (and modulus, so CATIP is not really a subset, but conceptually it is)
- Boolean operators
  - greater than and equals
- Integers
  - Boolean values are implemented as false: 0, true: not 0

[15] Computer Science at the Faculty of Science at University of Aarhus.
[16] I have not used the static analysis optimizations.

- Variables
  - variables are non-typed, unless being assigned to something of known type. The type of what a variable contains, may not change at runtime.
- Pointers
  - Pointers can be both pointer-pointers, integer-pointers, function-pointers or different combinations of these.
- Input and Output
  - The input and output are integer streams.
- IfThen, IfElse and While control structures
  - The classical control structures, with zero not satisfying and not zero satisfying.
- Functions
  - Functions are also the traditional kind, they return integers and can take a number of arguments.

CATIP Base

- Main function
- Arithmetic operators
  - addition, subtraction, division and multiplication
- Boolean operators
  - greater than and equals
- Integers
  - Boolean values are implemented as false: 0, true: not 0
- Variables
  - variables are integers.
- Output
  - The output is the only I/O in the CAPM. The output is not sequentialized, so therefore no guarantees are made about the order of outputted integers. In the current version of CAPM, the output functionality is assumed to not being used. The return statement of the main function is transformed into an output statement and this is the only output that should be used.
- IfThen, IfElse and While control structures
  - The classical control structures, with zero not satisfying and not zero satisfying.

So basically, all the fun parts of TIP is not in CATIP. Even though the implementation of pointers in CATIP would have proven to a far stronger degree that a Cellular Automata processing model can be used as backend of imperative programming languages, because of time constraints, I have been forced to restrict CAPM to only support the functionality in the CATIP base.

4.2.1 Possible TIP modifications

If the entire TIP language was implemented, there would probably be a few more functionalities that would boost the effeciency and usability of CAPM if they implemented. These could either supported directly in the langauge or by some standard library. A language construct that would certainly boost the parallelism in CAPM, would be an explicit parallel foreach statement. It should be considered whether it is the programmer or CAPM that takes responsibility for side-effects. One of the most used constructs in Stack based programming, is the array. This is caused by the random access memory, that enables constant time lookups in such a data structure. In CAPM the primary collection data structure would probably be binary trees, as these are also the basis of the logical structure of CAPM programs.

I wrote in the beginning that I would not focus on the programming language, but there are certainly some optional constructs that could increase the performance of CAPM. But these will have to wait for future work.

5 Implementing CATIP

Until now, the descriptions of CAPM have been very general and the principles have only been described in broad terms. This chapter describes the prototype I have developed in the work on my Master's thesis.

As this prototype is quite complex, I will try to describe the principles and implementations chronologically from the point when the programmer is finished writing the CATIP code and wants to compile it, to the time the program is running and terminates.

When the programmer wants to compile the program written, the first thing that happens is that a slightly modified version of the normal frontend of a TIP compiler parses the program. The modifications are oriented towards the AST and will be described in section 5.1. When the program has been transformed to an AST, it is time to perform some static analysis that enables the program to be evaluated by the highly parallelized environment of the CAPM-VM. How this is implemented is explained in sections 5.2 and 5.3.

When the program has been modified by the static analysis, it still needs some transformations to conform to the requirements imposed at the meta cellular automata layer, which is the next phase of the compilation. These requirements are somewhat simple, but nevertheless important, as it is here the requirement of constant size of logical cells is introduced. In section 5.4 the simple technique used by the compiler, translating the AST to a meta cellular automata layer instruction set (MCALIS) graph, is described. The MCALIS graph is undirected and the vertices are MCALIS_Nodes that have at most degree four.

At this point we could choose to save the program and view it as a distributable abstract compilation. This option is not implemented in the prototype, but would certainly be a must if further work was to be done on CAPM and CATIP. The reason why such an abstract distributable file should be created at this level, is that the program at this level is completely independent of what CAPM architecture it is run by, i.e. there is no knowledge of dimensions or message passing protocol used. This abstract distributable compilation, could also be translated to other parallel architectures, e.g. translated into the Actor Model and run on an architecture built for that model.

The abstract compilation (or MCALIS graph), is taken as input by the Cellular Automata Layer Instruction Set (CALIS) compiler. The CALIS_Compiler creates CALIS_Node graph version of the MCALIS_Node graph, after this translation, which is more or less a mapping, the newly constructed CALIS_Nodes, are each put into a Cellular Automata Layer (CAL) automaton. The CAL_CellularAutomaton, are the cellular automata making up the grid. When all CALIS nodes, have a place to live the CALIS_Compiler translates associations into coordinates or "living quarter addresses" of the associated nodes. The CALIS_Compiler is described in section 5.5.

We are now at the point where the program is ready to run, except for one little detail; The message passing protocol (MPP). Section 5.6 contains the details of the implementation(s) of the MPP.

With the MPP in place, we are now ready to run the program. The runtime structure and algorithm is explained in detail in section 5.7.

5.1 CATIP Context Free Grammar and parser

In chapter 4 I described what programming constructs were supported by CAPM and thus supported in the CATIP subset of TIP. Now I would like to define it, so no confusion on what the CATIP language is should arise.

Table 4 contains the Context Free Grammar of CATIP, which should suffice as documentation of the syntax of the language. I will just reiterate, that the boolean expressions in the three control structures are true if the value of the expression is not zero, false if the value is zero. It basically states that when writing a CATIP program, then it should start by

program	$\rightarrow$	*main* { declaration S return exp;}
declaration	$\rightarrow$	var $id_1,...,id_n$;
stm	$\rightarrow$	id = E;
	$\rightarrow$	S S
	$\rightarrow$	if (E) S
	$\rightarrow$	if (E) S else S
	$\rightarrow$	while (E) S
exp	$\rightarrow$	*intconst*
	$\rightarrow$	*id*
	$\rightarrow$	(exp)
	$\rightarrow$	exp binop exp
		where binop $\in \{+, -, *, /, ==, >\}$

Table 4: Context Free Grammar for CATIP

```
main(){
var v1, v2, ..., vn;
```

and end with

```
return exp;
}
```

The things in between are standard notation. I.e. like the notation of a subset of Java, C# or C++ equal to CATIP.[17]

The parser used, as mentioned in chapter 4, is basically the TIP parser. There are however some extensions, not to do with syntax, but the AST tree created. These extensions have exclusively to do with the sequentializing analysis described later in this chapter.

5.2 CATIP Compiler and Static Analysis

The front-end of the CATIP compiler is not different from that of TIP. The parsing of code is handled in exactly the same way. The main difference between a "normal" compiler and a CATIP-compiler is that we have to handle parallelism. The code is activated in what can be seen as a horizontal sequence:

All statements in the outermost block of the main function will be activated in parallel, thus potentially run in parallel.

My approach to parallelization in CATIP is somewhat indirect. I have chosen, at least as a starting point, not to use explicit parallelizing static analysis with loop unfolding and others. I have instead opted for a more subtle solution, based more on runtime control structures than explicit parallelizing static analysis. One reason for this approach is that as CATIP does not yet support pointers, functions and collection, the loop unfolding analysis and others would be more or less useless to perform at this point. In addition to this, the parallelizing techniques often takes processor amount into consideration, and I have at the moment not found any techniques that has an environment sharing the fundamental attributes of cellular automata, as a goal. I.e. the "update all", the locality problems and the intertwined nature of code and data that have the consequence that a parallel read can not be performed directly. In addition to not fitting well with known techniques the properties mentioned also have the consequence that things are, as

[17] Assuming every conditional statement in Java and C# were extended by if/while(false == (0 == exp))

usual in CATIP, turned upside down, because of the Cellular automata processing principle: *all instructions run in parallel unless controlled not to.* That things are turned upside down, can off course be handled by abstraction, so that we do not need to think of this to much. The way I handle the parallelization of the code is by ***sequentializing*** it.

5.3 Sequentializing static analysis and runtime control structure

To be sure that there are no misconceptions on what I mean by the word *sequentialize* I will start by giving it a proper definition:

Definition 2 *Code is sequentialized by an analysis that describes the sequential semantics of a program by identifying sequential dependencies between changes to and use of data. This sequentialized code will need a runtime control structure that obeys the invariants described by these dependencies.*

To perform the analysis described I have opted for Static Single Assignment form and the use of ϕ-functions as a fundamental principle. The SSA and ϕ-functions are subsequently used to create runtime control structures that ensure that values of variables are used in expressions only when it is safe to do so. How this runtime control structure works will be described later, but for now it is sufficient to imagine that the ϕ-functions gives values to variable expressions by conditional events.

5.3.1 SSA

Static Single Assignment form is a simple transformation of a program. At all places where a given variable is assigned, the variable in that assignment is renamed with a unique suffix, such that a variable is only assigned once. After this, variable expressions are replaced with a collection of variables that can define their value. This is more or less equal to knowing the assignments that can have defined the value of a variable at a given program point.

Source	SSA
1 `x = 3;`	1 `x1 = 3;`
2 `if (input > 4) {`	2 `if (input > 4) {`
3 `        x = x*2;`	3 `        x2 = phi(x1)*2;`
4 `}`	4 `}`
5 `if (input > 2) {`	5 `if (input > 2) {`
6 `  x = x*2;`	6 `  x3 = phi1(x1, x2)*2;`
7 `}`	7 `}`
8 `output x;`	8 `output phi(x1, x2, x3);`

Table 5: Example of SSA

This transformation of the program enables various forms of static analysis, including constant propagation, interval analysis, and many others. These different static analysis opportunities are not the focus of my use of SSA. My use of SSA is to use the ϕ-function, so that at runtime it will know exactly what assignment have defined the value of the variables represented, when it is evaluated.

5.3.2 Using ϕ-functions as a runtime Sequentializing control structure

ϕ-functions are usually used for static analysis and not as runtime structures. It is used as a "magical" instruction and acts more or less like the *guess*-instruction in non-deterministic Turing Machines. Obviously CAPM is not able to do magic or perform divine guessing and as predicting

the exact runtime behavior of a program is unsolvable/undecidable, a conservative approximation will have to be performed. To create this conservative approximation, let us start by reflecting a bit on conservative approximations and undecidability.

Undecidable questions, often referred to as all the interesting ones, are either caused by depending on a problem that is undecidable by nature or by some sub computation relying on some external factor, which we do not know anything about. When a sub computation, as an example I will use the condition of an if-else-statement, relies on input, then it is impossible for us to know whether or not the body of the if-statement will be evaluated. There are however structural properties of some problems that makes the problems decidable. E.g. if the input was an integer and the condition was (input $\leq$ Integer.MaxValue). The problem of undecidability with regards to compile time questions, is that if the input is used at the beginning of the program, then the statements whose actions depend on the outcome of that statement, are undecidable.[18] Conservative approximation identifies cases like the $\leq$ Integer.MaxValue, although they might be much more complicated. If we considered that the first statement of the program made the program go in either one direction or another, e.g.

```
1 if (input)
2         while (true) {}
3 else
4         return 1;
```

then if confronted with the question of whether or not the program terminates, we must choose one of two conservative answers:

1. It is unknown whether or not the program terminates
2. If the input is not 0, then it loops forever, else it terminates.

or try running the program on every given input and see what happens. If we try running the program on any given input, then all input except input = 0, will result in the program never terminating, and hence we will never be able to answer whether or not it terminates. Answer 1 is useless, and does not provide any information. Answer 2 actually gives valuable information, although it might not seem like it at first glance. Answer 2 enables us to input an error message that could say something like this: *The program just entered an infinite loop, press F1 to continue.* My approach to the use of ϕ-functions, is to perform an analysis that provides answer 2. This is done by the sequentializing analysis by avoiding some of the approximations, and instead create a structure that at runtime answers compile-time undecidable questions, at the very moment they get decidable. The important undecidable question here is of course, what the value of a given variable in the program is at a given point of use. Answer 1 to this question, is that it is unknown (except some cases) what assignment defined the value of a variable v at program point p and would force us to evaluate the program in a sequential manner, to be sure that the correct values are used. Running the program on all inputs, does not really apply here, it would only give the values that are the same on all inputs.[19]

Answer 2 provides us with a set of conditions and cases. E.g. *the value of variable v1 is at program point p1 defined by the assignment a1 if if1 evaluated to true and if2 evaluated false.* This answer enables us to run parts of the program in parallel, something that was not possible if we used the answer 1 approach.

It is important to notice that there still is a conservative approximation in this scheme and I do not expect that it can be safely avoided, as it would decide an undecidable question. This

[18] unless the case is equal to the one just sketched.

[19] Or maybe give us a giant table, which would tell the value of variables on all different inputs. By giant I mean too big.

approximation has to do with the question: *Exactly how much information is needed for some execution of a program to safely answer the question about the value of a variable.*

The ambiguous value of a variable is a consequence of assignments in conditional control structures, i.e. if- and while-statements. From this it follows that the approximation we need to postpone, is acting on the outcome of conditional statements. And this is exactly what is done in the non-magical ϕ-function.

When a variable is used in an expression, it has to have been assigned by some earlier assignment in the program.[20] Because of conditional control structures, several assignments could have given a variable its current value. As CAPM does not have a single program counter, stack top etc. usage of a variable, in addition to the need for explicitly sequentializing the evaluation of statements, would require a non-parallel read of a variable value, which would very likely slow down the program and needlessly inhibit the parallelism in the program evaluation. There is furthermore the likelihood of the mirror image of the problem of reusable local variables would occur. In sequential programming, analysis is made to lower the amount of memory used on local variables. The parallel programming version of this problem is that one variable can be used as several different variables, with no relations to each other. When this is the case, they can actually be run in parallel. An example of this, will come at the end of this chapter.

To avoid the problems described above or at least minimize their impact, I have chosen to use an event driven use of variables.

The key to using ϕ-functions as basic sequentializing control structure in CAPM is to let each of the arguments in the ϕ-function receive assign events from their unique assignments and let their use depend on the outcome of conditional statement evaluation. In other words, an argument is used as the value of a phi function if:

1. It has received a value from its unique assignment
2. It can be guaranteed that the value of the variable represented by the argument is not sequentially written over by an assignment prior to use.

To get an overview of exactly what these criteria means, let us take the following code examples, which covers all basic cases of ϕ-functions and argument use.

A simple if-statement The first example in table 6 illustrates the fundamental reason for using ϕ-functions. What argument to use in the ϕ-function in line 7, depends on the outcome of the evaluation of the if-statement in line 4. If it evaluates to true, then the x2 argument will be used otherwise the x1 argument will be used. If there were several if-statements looking like line 4-6 between the x1 assignment and the return statement, then the use of x1 would depend on all of them being false. The other arguments would depend on all following if-statements to evaluate to false. In cases where the first if-statement evaluated to false and no other, then the arguments that depends on the rest being false, are not used, because they are not evaluated and hence, have not received any value to propagate to the ϕ-function.

A simple while-statement The second example in table 7 illustrates what happens in loops. This example is simple enough not to have any of the issues that will be handled later, but there are a couple of things to notice. The x2 argument of phi3 can receive several values, but the value is only used when the condition of the while-statement evaluates to false. Another thing to notice is that the x1 argument in phi3 will always, unless the program loops indefinitely, have its

[20] The terms: Earlier, prior etc. refers to the sequential semantics of the program, i.e. the order in which a sequential processing model would evaluate the statements in the compiled code

Source	SSA and ϕ
1 `main() {` 2 `  var x;` 3 `  x = 3;` 4 `  if (input) {` 5 `    x = x + x;` 6 `  }` 7 `  return x;` 8 `}`	1 `main() {` 2 `  var x;` 3 `  x1 = 3;` 4 `  if (input) {` 5 `    x2 = phi1(x1) + phi2(x1);` 6 `  }` 7 `  return phi3(x1, x2);` 8 `}`

Table 6: Simple if example

condition satisfied, even though the x2 assignment was evaluated. So somehow the value from the x1 assignment must be suppressed if the while loop evaluates to true at least once.

Suppressing values of assignments, because an argument *knows* its value is obsolete, works by deleting the value received when the use conditions are satisfied, not by actively clearing the value. Why and exactly how is explained later. We now have use and clear conditions and these are the only events depending on the outcome of the evaluation of conditional statements. There are a few more elements to be introduced, but in essence these two types of events are what drive the ϕ-functions.

Source	SSA and ϕ
1 `main() {` 2 `  var x;` 3 `  x = 3;` 4 `  while (input) {` 5 `    x = x + x;` 6 `  }` 7 `  return x;` 8 `}`	1 `main() {` 2 `  var x;` 3 `  x1 = 3;` 4 `  while (input) {` 5 `    x2 = phi1(x1, x2) + phi2(x1, x2);` 6 `  }` 7 `  return phi3(x1,x2);` 8 `}`

Table 7: Simple while example

Reload assignments The last simple example in table 8 illustrates a concept that makes the use of while-statements a lot easier than what would otherwise be the case. The concept is basically to reset and reload all state information in the condition and the body of while-statements after the body is evaluated. When the state of all ϕ-functions have been reset, reload assignments are evaluated. This gives new values to the variables/ϕ-functions used in the body and condition of the while-statement. In addition to the simplification of the internal workings of the while-statement, it furthermore gives the simplification that reload assignments are the only values exported from the while-statements. Unfortunately this solution may restrict the parallelism of loops, e.g. cases where the beginning of the loop can be evaluated at iteration i + 1 before all of the statements in the body have finished evaluating and if a variable is only used in a while-statement, but not assigned, then it could be used afterwards, but this is unfortunately currently restricted in the sequentialization. Despite this potential risk, I think that this solution will work well as it avoids several difficult problems, but this is something that could be looked into later on. A possible solution for reducing the sequential restriction of while loops, could be to implement the ϕ function in a manner, where the conditional statement would begin to evaluate itself, when the variables involved in it were ready. Exactly how this would be done, is quite complex but if it was possible to show that the

principle was sound, then a runtime one-step unfolding of the loop could take place. More on this in chapter 7.

Source

```
1  main() {
2    var x;
3    x = 3;
4    while (input) {
5      x = x + x;
6      if (x > 100) {
7        x = x + 1;
8      }
9    }
10   return x;
11 }
```

SSA and ϕ With reload

```
1  main() {
2    var x;
3    x1 = 3;
4    while (input) {
5      x2 = phi1(x1, x4) + phi2(x1, x4);
6      if (phi3(x2) > 100) {
7          x3 = phi4(x2) + 1;
8      }
9      x4 = phi5(x2, x3);//the reload statement
10   }
11   return phi6(x1,x4);
12 }
```

Table 8: Introducing reload assignments

Argument information in detail To give a sense of the information needed to let the arguments of ϕ-functions decide whether they are the value to be used, I will, in table 9, list the use and clear events for the arguments in the third simple code example in table 8. The ϕ-functions with the same values, i.e. used at the same interval of changes, are for the sake of simplicity listed in the same row, despite their arguments being different entities/nodes. Each argument has two sets of conditions. In this simple example, there is only zero or one use and clear condition. In more complex code, each argument could, and often would, have many conditions. How these are generated and what form they have, will be handled later. When neither use condition or clear condition have any values, the value received by the argument is instantly forwarded as a value to be used by ϕ-function.

ϕ fct	Arg	Who overrides	Use Cond	Clear Cond
phi1, phi2	x1	None	None	None
	x4	None	None	None
phi3, phi4	x2	None	None	None
phi5	x2	x3	¬if	if
	x3	None	None	None
phi6	x1	x4	¬while	while
	x4	x4	¬while	None

Table 9: Argument use and clear conditions

Dominating assignments The column *Who overrides*, shows for each argument what later assignments of the variable represented can define the value of that variable, at the program point of the given ϕ-function. In the case of the x1 argument in phi6, this is only x4. This is because even though the x1 assignment value can be overridden by that of x2 and x3, it is only the value of the x4 assignment, that can have defined the value at the time of the phi6. When each argument in each ϕ-function knows their dominating assignments, we can define their use and clear conditions. This is done by observing that in the AST there is a unique path to each assignment. These paths can be described by a list containing pairs of a conditional statement and an outcome. To be sure

a value of an assignment is not dominated by another assignment, then it is enough to know that none of the dominating assignments were evaluated, in other words that all the paths leading to a dominating assignment were avoided. The clear conditions are much the same, but somewhat simpler. They are produced by simply looking at the closest surrounding conditional statement of a dominating assignment and the outcome leading to the evaluation of the assignment. Except for a special case regarding reload assignments, an argument must be cleared if an outcome of a condition leads to the immediate evaluation of a dominating assignment.

5.3.3 Runtime control structure

Before I describe exactly how to generate the information needed to create the runtime control structure, I want to describe in detail how this structure works. This description also sheds light on some of the reasoning behind some of the choices made in the previous section.

As conditional statements are what introduce the need for ϕ-functions, I will start by describing the structure of and control flow in If and While-statements.

If-statements In CAPM there is no distinction between if-then and if-then-else statements. The structure of the If-statements consists, as usual, of a condition and two bodies; then and else. The condition is handled like an assignment expression, or to compare to the SACAM example, the output statement. Instead of having an ExpTarget node to the right of the Exp node, there is a type of node, called NotZero. NotZero sends an evaluate message to either its linker or its left child, when it receives an EvalauteMsg from its parent. Who it sends the evaluate message to, depends on the value of the message from its parent. If the value is not zero, it sends Evaluate to its linker, i.e. the then block. Otherwise it sends an evaluate message to its left child. The two blocks contain, in addition to the statements from the actual source code, also listeners from ϕ-function arguments.

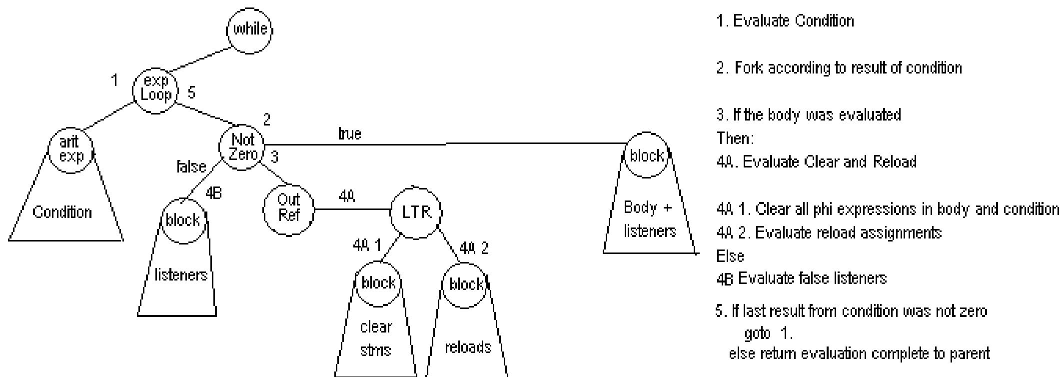


Figure 8: The structure and flow of control of while-statements

While-statements The structure of the while-statements is almost equal to that of the if-statement. There are two differences. Instead of an Exp node as left child to the root, it has an ExpLoop. ExpLoop *remembers* what value it sends to its NotZero node, and evaluates the condition again if the value of the sent evaluate message was not zero, when receiving evaluation complete from the NotZero node. In other words, the ExpLoop can be seen as a NotZero and an Exp node merged into one self-referential code. The other difference is the right child of the

NotZero node. The right child of the NotZero node, is an outgoing reference to a LeftThenRight node, which it sends an Evaluate message to, when finished evaluating the body of the while-statement. This node resets all the ϕ-functions used in the body and condition of the while loop, but not the reload statements. After all ϕ-functions have been reset, the reload statements are evaluated, thereby preparing for the potential next iteration of the while-statement. When the while-statement evaluates to false, then all ϕ-functions inside the while-statement including the reload statements are reset.

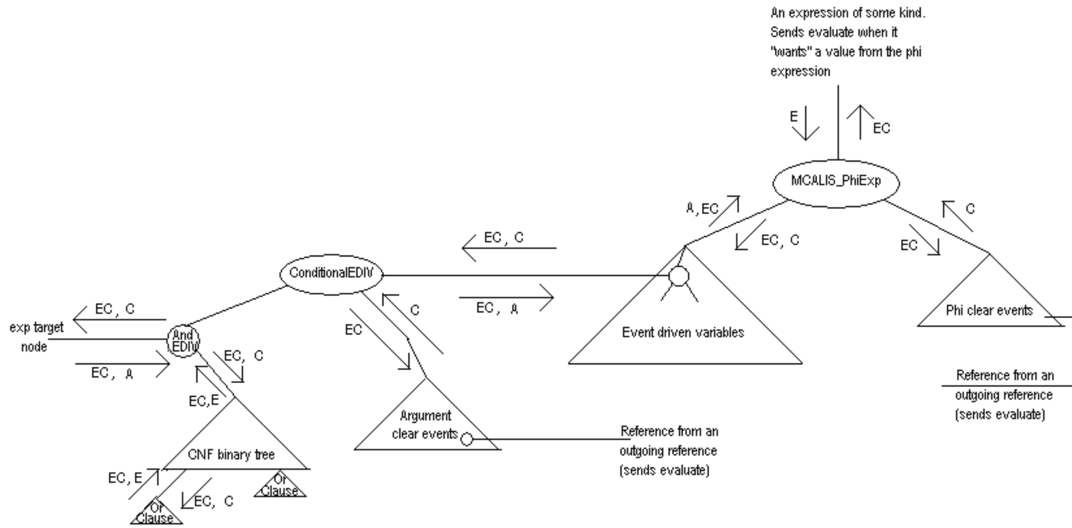


Figure 9: The phi function sequentializing control structure. Possible message types sent by edge: E = evaluate, EC = evaluationComplete, A = assign, C = clear.

ϕ-functions The control structure for the ϕ-function is the fundamental sequentializing control structure, so to be sure that it is sound, I will try to argument for the correctness of the structure.

GIVE A CONVINCING ARGUMENT FOR CORRECTNESS HERE!

5.3.4 Static analysis required to create non-magical ϕ-functions

I have described the information needed to create the runtime control structure and even though it may appear complex, it can be handled in eight simple steps.

Step 1: Introduce load and reload assignments in while loops This step introduces load and reload assignments of all variables used inside a while loop. The load assignment handles the use and clear conditions so the reload assignment has no state information.

```
1 for all WhileStm w in program do :
2         for all variables v used in w do :
3                 add stmLoad : v = v at end of body of w
4                 add stmReload : v = v after stmLoad , mark stmReload as
                     reload assignment
```

Step 2: Initialize SSA - Rename identifiers of assignments This step does not completely create the static single assignment form, it simply renames left sides of assignments in the AST.

```
1 Associate a counter with each variable declared
2 for all AssignStm a in program do :
3         counter = var assigned by a;
4         add suffix to identifier of a, where suffix is [seperatorSymbol] +
             counter++
```

Step 3: Replace all variable expressions with ϕ-functions This step replaces all variable expressions in the program with ϕ-functions. This is done to prepare the AST to the Control Flow Graph created in step 4.

```
1 for all VarExp v in program do :
2         Replace v with new PhiExp p
3         associate p with v.//so it knows what variable it represents.
4         Add p to set phiExpsIntroduced //so they are easily accessed later
```

Step 4: Initialize ϕ arguments This step uses the Reaching Definitions analysis model[STATIC], to associate assignments with ϕ-functions. This is done by producing arguments for each assignment that can potentially have defined the value of the variable which a given ϕ-function have replaced in the AST.

```
1 CFG = InitiateCFGFrom(currentAST) //fine-grained, statement level is not
     enough, unless following the analysis phi-functions lookup the args in
     their respective expressions or statements.
2 RunLocalAssignmentDominationAnalysis(cfg) //Basically Reaching Definitions.
      However it also associate each phi expression, assign statement and
     conditional statement with their respective program points.
3
4 for all PhiExp phi in phiExpsIntroduced do :
5         for all AssignStm assign program point of phi do :
6                 if assign assigns the variable phi represents then
7                 add an argument to phi that represents assign
```

Step 5: Initialize conditional outcomes leading to evaluation of assignments and ϕ-functions This step associates a path to each assign statement in the program. This is done by a simple Depth first traversal of the AST. When dealing with if else statements it is not as simple as in the pseudo code, but it illustrates the principle.

```
1 workList //represents the current path.
2 In a DFT of the AST then :
3 if Out then :
4         if ConditionStm then :
5                 Remove last element from worklist
6
7 if In then :
8         if ConditionalStm then :
9                 add path element to worklist //<conditionalStm, outcome>
10        if AssignStm then :
11                copy path from worklist
12        if PhiExp then :
13                copy path from worklist
```

Step 6: Establish argument domination hierarchy This step establishes for each argument in each ϕ-function, what other arguments dominates it. This is done by looking at the paths of the other arguments and checking whether the argument in question appears there. There are other more efficient ways of doing this, but this is a simple and easy way.

```
1 for all PhiExp phi in phiExpsIntroduced do :
2         for all Arg arg in phi do:
3                 if this is not a reload assignment then :
4                         for each Arg dom in phi where dom != arg do :
5                                 if assignment of arg appears on the
                                     conditional path of dom
6                                         arg.dominators.add(dom);
```

Step 7: Initialize use conditions for arguments Initializes the use condition sets for each argument of each ϕ-function. There is a set of conditional outcomes for each dominating assignment of an argument. Write a CNF producer for use conditions.

The use conditions for an argument arg are : $\bigwedge_{dominators} (\bigvee \neg \text{conditionaloutcomes} \in \{ \text{pathToDominator} \setminus (\text{pathToDominator} \cap \text{pathToArg}) \}$

```
1 for all PhiExp phi in phiExpsIntroduced do :
2         for all Arg arg in phi do :
3                 for all Arg dom in arg.dominators do :
4                         for all condOutcome co in dom.assignment.
                             conditionalDominators do :
5                                 if co not in arg.assignment.
                                     conditionalDominators then
6                                         arg.addUseConditionToSetOf(
                                             oppositeOutcomeOf(co), dom)
```

Step 8: Initialize clear conditions for arguments This step adds clear conditions to each argument of each ϕ-function, if there are any. An argument gets a clear condition for each dominator unless the dominator is a reloader and the dominator is equal to the argument, i.e. a self-dominating argument.

```
1 for all PhiExp phi in phiExpsIntroduced do :
2         for all Arg arg in phi do :
3                 for all Arg dom in arg.dominators do :
4                         co := dom.assignment.conditionalDominators().
                             getLast()
5                         if ! phi.getConditionalOutcomesLeadingToEvaluation
                             () containsIgnoreOutcome co then :
6                                 if dom is reloader and arg is reloader then
                                     :
7                                     if dom.assignment != arg.assignment then
                                         :
8                                         arg.clearConditions.add(co)
9                                     else //do nothing
10                                else
11                                        arg.clearConditions(co)
12                        else //do nothing
```

Now the static analysis is complete and the rest of what happens, happens in the MCALIS compiler.

5.3.5 Optimization

Balance expressions Normally when compiling, arithmetic expressions and other of the binary tree kind, they are rarely balanced and most often have a form as close to a linked list as possible.

Optimizations to the runtime control structure It is very likely that the control structure of the while-statements is too strict and sequential and some of these could be relaxed in some way. I have chosen not to do that, because of the wise rationale embodied by the saying:*if it ain't broke, don't fix it.* Optimizations to the use of ϕ-functions as sequentializing control structure will be described in chapter 7.

5.3.6 Summary

The sequentilization of the code generates logical conditions that dictates the use of a particular assignment of a variable in a given execution of a program. Information about reload assignments in while-statements is stored and will be used by the MCALIS_Compiler to handle these correctly. Exactly how, is described in 5.4.4. The current solution, where ϕ-functions are used at runtime, to make sure that the parallel execution obeys the sequential semantics of a program, is intended to be as clean as possible, not obfuscating any of the basic principles. This has as a consequence, that there may be several optimizations that can be implemented. They are postponed for now, but I will return to it in section 7. [21].

Bonus information gained by the sequentializing analysis Use of uninitialized variables are identified, but this should not come as a surprise, as the Reaching Definition analysis is normally used for that analysis. A more surprising result is, that parallel use of the same variable, if possible, is achieved. This follows from the change that there are no actual variables, only values being tossed around. So if we have:

```
1  main() {
2           var x, y, z;
3           x = 4;
4           y = 0;
5           z = 0;
6           while (x > 0) {
7             y = y + x;
8             x = x - 1;
9           }
10          x = 5;
11          while (x > 0) {
12            z = z + x;
13            x = x - 1;
14          }
15          return z + y;
16 }
```

the two while-statements will actually be performed in parallel. This is a consequence of the analysis as a whole, and not something that is explicitly analyzed or handled.

That variables do not exist explicitly at this point, comes as no surprise as this is what could be expected by the level of abstraction. It could be compared to how we in assembler copy a value

[21] It is important to note that if there were several input and output statements, then they too would need an internal ordering like the one used to control the ϕ-functions. As this could be solved by the same techniques as the ones described in this section, I have only allowed the use of one output statement, and that statement must be the last statement in the program.

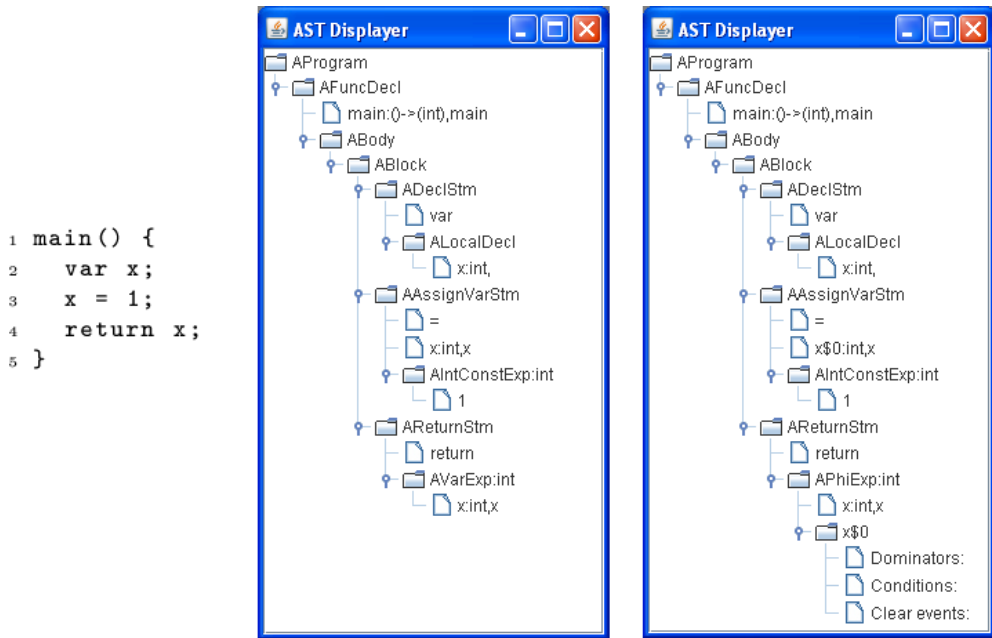
if we are to use it twice. The sequential structure of the stack is what, in my view, leads to the use of stored variables and how the random access memory ensures that this solution is efficient.

5.4 AST to MCAPM

To translate the AST, produced by the frontend of the TIP compiler, into a Meta Cellular Automata Layer Instruction Set (MCALIS) graph, I have implemented an MCALIS Compiler. The MCALIS_Compiler takes an AST as input, and in a few steps an MCALIS graph is created. These steps are:

1. Sequentialize the program, using the technique described in section 5.3.
2. Initiate the basis of the program, i.e. the input and output node and main function. The return statement is transformed into the only output statement in the program.
3. Compile the main function
4. Create reload statements in while-statements
5. Compile ϕ-expressions
6. Create clear statements in while-statements

As the descriptions of the different steps of the CAPM compiling process may be unfamiliar, I introduce a simple example program, which exsimplifies[22] some of the abstract concepts introduced. This ongoing example program is three statements long.



(a) Step -1, the code. (b) Step 0, the AST. (c) Step 1, the sequentialized AST.

Figure 10: Step -1 to 1

The Abstract Syntax tree of the simple example program can be seen in (a) in figure 10.

[22]To simplify by use of example(s).

5.4.1 Step 1. Sequentialization

As step 1 has already be thoroughly explained, I will skip details here. I will keep the ϕ-functions simple, i.e. without the dominators, conditions and clear events, as they have no relevance for the behavior of the simple ongoing example, they would furthermore complicate it quite a bit. The ϕ-function, simply receives and assumes an unconditional value.

5.4.2 Step 2. Initiate basis

The reason why the input and output node are created at this point, is that if they are used, we might as well add the associations required. In the current version of CAPM the output function is only used as the last statement (the return statement) of the main function. But nevertheless, the ϕ-functions need to be able to set the singleton MCALIS_Output as target.

To handle the execution and termination of the program, an MCALIS_StatementCollection representing the main function, is marked, such that the runtime environment can recognize the program as finished when it receives an evaluation complete message. This statement collection only has one outgoing reference to a LeftThenRightSequential (LTR) node. The LTR sequential node, evaluates its left child which will evaluate the outermost block of the main function, except the return statement (last output statement). The last output statement is evaluated by the right child of the LTR sequential node. So the basis of the main function is to create a structure that ensures the evaluation of the body (except the last output statement), followed by the evaluation of the output statement. And last but not least, be able to register a place where a received evaluation complete message means that the program has terminated. The execution of the program is done, simply by letting the termination-monitored StatementCollection start with an Evaluate message with the left child as target.

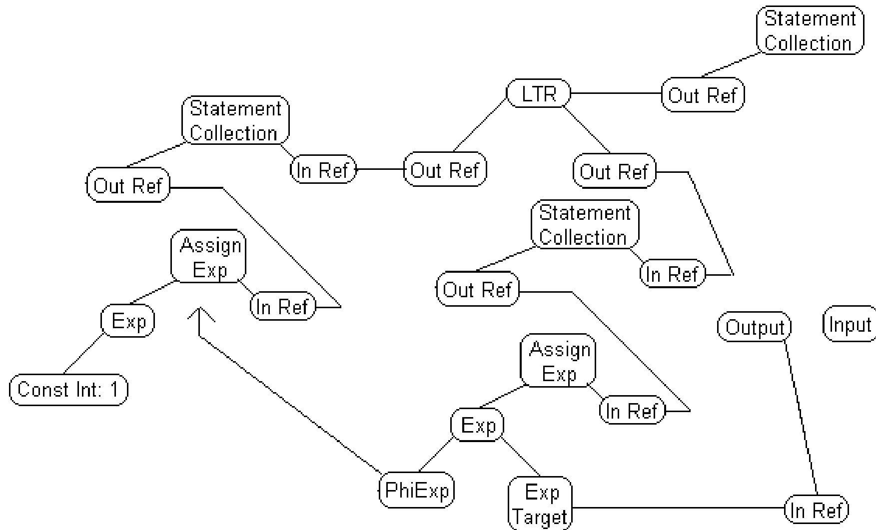


Figure 11: Step 2 and 3, the Meta Cellular Automata Layer Instruction Set Graph

5.4.3 Step 3. Compile main function

Having this as just a step equal to the others, may seem a bit odd. One would expect that this is what the *AST to MCAPM* section should describe, but what actually happens in this step is that

the AST is translated to an MCALIS graph. This is done in much the same way as a compiler outputting assembler would do. It simply performs a depth first traversal of the AST and delegates the translation as required. I.e.

- CompileMainFunction(AFunction function) → CompileBlock(ABlock block)
- CompileBlock(ABlock block) → CompileStatement(PStm stm)
- CompileStatement(PStm stm) → or CompileIfStatement(AIfThenStm ifThenStm) or CompileAssignment(AAssignVarStm assignVarStm)
- and so on and so forth.

In each of these methods, an MCALIS_Node is returned and used by the invoker method to create an association between the logical nodes created by each. A method can invoke several other methods and thus receive several MCALIS_Node objects. An example of this is the CompileIfThenElseStm:

```
1   private MCALIS_Node CompileIfThenElseStm(AIfThenElseStm stm, HashMap<
       String, MCALIS_IntVariable> variables) {
2     MCALIS_AssignmentExpression result = new MCALIS_AssignmentExpression();
3     MCALIS_Node condition = CompileExpression(stm.getCondition(), variables
        );
4     MCALIS_Node thenBranch = CompileStm(stm.getThenBody(), variables);
5     MCALIS_Node elseBranch = CompileStm(stm.getElseBody(), variables);
6     MCALIS_ForkNode target = new MCALIS_ForkNode();
7     stm.setForknode(target);
8     target.SetTrueBranch(thenBranch);
9     target.SetFalseBranch(elseBranch);
10    result.SetEvaluateExpression(condition);
11    result.SetTargetExp(target);
12    return result;
13  }
```

What happens in the above, is the construction of the Cellular Automata version of the If-Then-statement. All the methods translating parts of the AST are implemented such that a Binary Tree Normal Form representation is created. In figure 11 the MCALIS graph for the simple ongoing example is shown.

In the case of the while-statement, the identity assignments that were added by the sequentializing analysis, are ignored. Only the original body is compiled. The identity assignments are handled in step 4.

5.4.4 Step 4. Create Reload statements

To create the reload statements of the while-statements, we need to know all identity/reload assignments in the while loop. All the reload assignments are added to the MCALIS_StatementCollection the while structure evaluates after its body have finished evaluating. This step is not applicable for the simple ongoing example.

5.4.5 Step 5. Compile ϕ expressions

Now it is time to compile all the ϕ-expressions introduced in the program. This is done by the MCALIS_PhiExp class, in the method InitializePhiExp(). For each of the assignments that can define the value of the ϕ-expression, it creates a conditional event driven int variable node

(MCALIS_ConditionalEDIV) which contains the logical structure and ingoing value, as described in figure 5.3.3. Each of these ConditionalEDIV nodes, are afterwards associated with an EventDrivenIntVariable node, which is added to the left subtree of the PhiExp node. The simplified version, without the conditional statements, can be seen in figure 12.

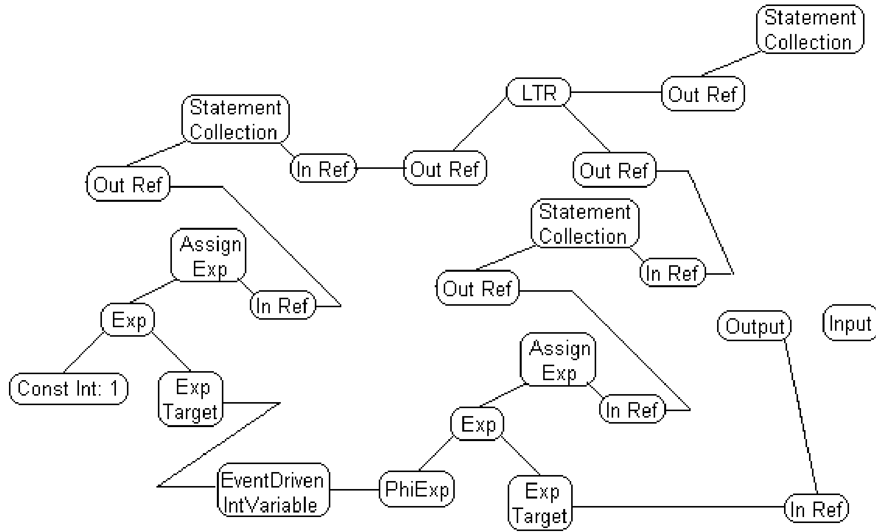


Figure 12: Step 5, the phi expressions is set as target of the assignment that can define its value.

5.4.6 Step 6. Create Clear statements

When we are done introducing the ϕ-expressions, the principle of resetting the ϕ-expressions in while-statements is handled. This simply involves adding clear references from the StatementCollection evaluated between the body and reloads, to all the ϕ-expressions in the body and condition of the while-statement. In addition to this, all ϕ-expressions in the while-statement, including the reload statements, have clear references from the false branch of the condition of the while, added to them. This step is, as step 4, not applicable for the simple ongoing example.

5.5 MCAPM to CAPM

The Cellular Automata Layer Instruction Set compiler (CALIS_Compiler) is much simpler than the meta-counter part.

The CALIS-compiler takes all MCALIS-nodes created by the MCALIS_Compiler. It is at this point the structure of the grid and message passing protocol should be used to optimize the placement of nodes. This also explains, why it should be the MCALIS graph which should be used as abstract distributable compilation.

Currently the message passing protocol is not used, but in a generalized version of CAPM where many different MPPs could exist, it should be used by the CALIS compiler, to create a grid of sufficient size and structure, as message passing protocols can have different requirements to the size of the grid and structure of it. And furthermore place the CALIS nodes in the grid, such that the paths defined by the message passing protocol, are minimized.

The translation of the MCALIS nodes to CALIS nodes are quite simple, as they basically maps from one class to another. The MCALIS_Node class is responsible for this translation. When all

the input nodes, have been translated to CALIS nodes, they are each assigned to an individual CAL_CellularAutomaton. It is obvious that if we wanted to optimize the placement of the CALIS nodes in the grid, it is in this step, such an optimization would take place.

The last thing to do, with regards to the CALIS nodes, is to synchronize their associations, i.e. translate the associations of the MCALIS node, they were created by, into positions of their parent, left child, right child and additional relative.

The last thing done is to initialize the message passing protocol and synchronize the grid buffer.

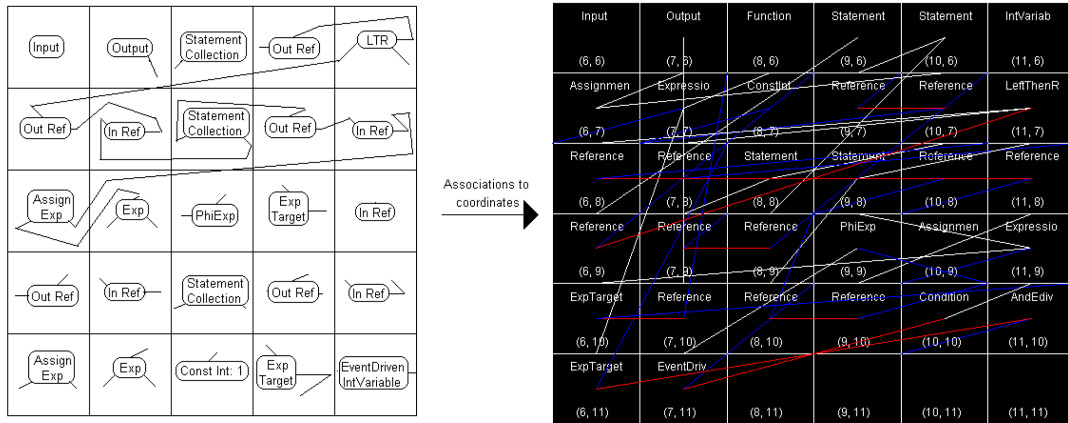


Figure 13: MCAPM to CAPM. CALIS nodes are created from MCALIS nodes, added to the grid and associations updated.

5.6 Message Passing Protocol

The Cellular Automata grid can be viewed as a network graph, where all cells are network nodes that have a network edge to each of their immediate neighbors. The main difference between networking/routing in CAPM and normal networking in meshes or routing in normal computer networks, is that messages are not forwarded in the normal sense, rather they are read and erased by nodes, in such a way that a packet or message is routed from a to b. This is a consequence of the definition of the Cellular Automata processing model, where each cell achieves a new state by looking at their neighbors. The help given by CAPM to achieve observation based routing is that the update of the cells is performed synchronized, so there is no risk of low level concurrency errors, e.g. reading memory that is changed during the read.

Normal routing protocols could be translated into a cellular automata model. Handshaking could be performed between to neighboring cells, but such handshaking is not immediately necessary if the message passing protocol is implemented in a way where it can be shown that a message will travel from a to b, simply by the combined behavior of all the cells running the same protocol algorithm. As an example, consider a cell that contains a message of which it is not the target, then it has to wait for one of its neighbors to read the message before it can safely delete that message and be able to read a new one.[23] It is possible that implementing a normal routing protocol with handshakes, etc., could achieve better performance than the message passing algorithm I have implemented, but I decided early in the project that I would try to develop an MPP that was

[23]This could also be viewed as if the cell tries to send the message to each of its neighbors. The neighbors then reflect the message or keeps it. If one applies this view to understand the problem, then the problem is that at one point exactly one neighbor does not reflect the message sent. This information will be used by the cell sending to delete the message contained by it.

deterministic, had arrival guarantee and had an algorithm that looked and acted like a Cellular Automata rule.

Why a deterministic MPP? The reason why I wanted the MPP to be deterministic, was to avoid handshaking. The hand shaking would be needed if a cell α decided to forward its message to the cell β to the right of it. At the same time this happened, the cell δ decides to forward its message to the cell above, β. What would happen in this case? β would choose one of the messages. If α was chosen, then what would δ do? Even though it could be guaranteed that with increasing probability a message would be forwarded, getting closer or at least not farther from its target, I was worried that patterns of oscillation could arise, and remain hidden because of the obscurity introduced by the randomization. In addition to that concern, this solution to the MPP-problem, just does not *feel* like cellular automata, in the classical sense. That is, in my eyes it should be possible from any given configuration to predict the configuration of the next generation with certainty.

Why guarantee arrival of messages? Many problems in routing and network communication are like a dog chasing its own tail. As an example, consider what happens if we do not guarantee arrival of messages. Then, in addition to a lot of state information, the cell sending a message would need to reach a point where it interprets the lack of reply as evidence of the message being lost. At that point, either the message or the reply was lost. This is a problem known in network protocols, and the solutions are often less than elegant and can increase the needed bandwidth considerably. Deterministic behavior is certainly not the defining attribute in most of those solutions and thus by the previous paragraph, are already standing with half their feet in the grave. In addition to the what-message-was-lost-problem, the timing involved would require that we knew something about the worst case message passing time. This might seem simple, but when one sees the proofs involved in proving worst case time of message passing protocol, a man must ask himself whether or not that is something he wants to put himself through. I did not.[24] That put the remaining feet in the grave.

Why look and act like a Cellular Automata rule? During the work on my Master's thesis, I have tried to keep true to the "spirit" of the Cellular Automata concept, to see where it would take me. Abandoning this approach, when it came to message passing, seemed to me like a bad idea. I have in the above, given a few examples of handshaking. Using handshaking would, in my view, be to evade my own ground rules by technicality. It could on the other hand turn out that message passing would be the appropriate abstraction level to simply think in terms of messages being sent, not read. An argument for that point of view, is that if CAPM was to be implemented by a simpler set of rules in simpler Cellular Automata, maybe even Conway's Game of Life, the messages would actually be sent, not read. I have not researched these solutions, as my primary objective was to get the message passing working and preferably in a CA like fashion.

When implementing a message passing protocol in a graph or grid, some of the attributes it is evaluated by are:

1. What is the worst case time for a message to get from a to b?
2. What is the average time for a message to get from a to b?
3. What is the buffer size in each node or cell?
4. Are all messages guaranteed to arrive?

[24] I excuse the chauvinistic language, substitute him with her, when appropriate.

5. Can duplication of messages occur?
6. Can deadlock occur?
   - Here deadlock refers to message passing, not parallel computation, although they can be viewed as different incarnations of the same fundamental problem; shared resources.
7. Can starvation occur?
   - Here starvation refers to the routing interpretation of starvation. This can occur by routing a message in an oscillating or cyclic pattern.

There are other considerations than the ones listed here, e.g. fault tolerance, bandwidth of protocol communication etc, but the ones listed have been my main considerations, as trying to consider all the aspects of a solution is beyond the scope of this project. I have assumed that copying information works perfectly and does not introduce errors.

A requirement I have decided upon is that the buffer size of a CA is exactly 1, i.e. a CAL_CellularAutomaton can contain 1 message at any given time, excluding the ones contained in the CALIS_Node. I have furthermore decided that, as the CAs are limited in size, the messages must be guaranteed to arrive to their target, so that state information, timers, knowledge of worst case routing time and other things are avoided. I have furthermore assumed that the grid runs one program and as a consequence, starvation may only occur temporary unless a part of the program loops indefinitely.

Figure 14: How to handle message passing in a 2D grid by viewing it as 1D

When implementing an MPP in a grid with more than one dimension, where cell buffersize is 1, cycles are more or less unavoidable, and therefore an MPP will have to be able to handle these cycles, such that not all are waiting for the one in front of him to get out of the way. This problem could be avoided by an MPP that makes the multidimensional grid act as a one-dimensional. E.g. by having all pass messages from the CA above and below, and have slightly modified rules for the CAs on the top and bottom edge of the grid, where they look at their horizontal neighbors instead of the vertical they are missing. A depiction of this solution is shown in figure 14. This solution has provable bounds for performance of message passing, but I have chosen not to use it because the provable bounds are provably quite slow.

The reason why I have chosen to restrict the buffer size to be 1 message/CA, is first of all the KISS[25] principle. What I mean by this, is that I experienced that increasing the buffer size, lead to the deadlock cases being somewhat hidden and certainly more difficult to identify. Another reason for this choice, is that if we were to translate CAPM to a simpler CA model, then it is likely that messages would have to be passed around, as stable moving entities.[26] This could be viewed as physical entities moving around, and my thought was initially that the message passing protocol could be implemented using simple mechanics. Whether or not this has been achieved, I will let the reader decide by interpreting figure 15.

Implementing an MPP in CAPM is a bit different than normal routing and communication, because all the communication is handled by cells, being updated in synchronized buffered fashion, and the cells themselves are the only ones that can "write" in their own message buffer. To illustrate this, let us use a 1D example to get the basics in place.

In figure 16 an example is shown and an overview of the principles CAPM message passing.

[25] Keep It Simple Stupid
[26] Stable in the same sense of the glider retaining/repeating its shape/form.

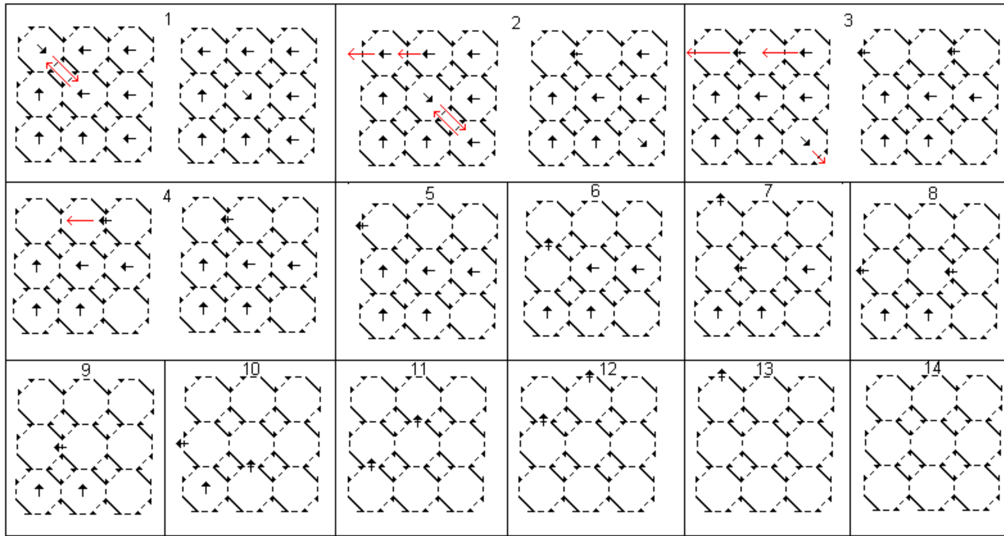


Figure 15: An depiction of a mechanical interpretation of SCAMP, the red lines represents the target pulling the physical message in its direction.

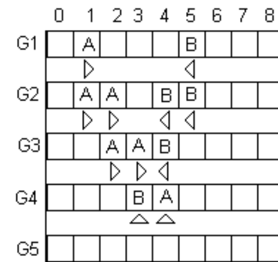


Figure 16: 5 generations illustrating 1D message passing

The states that are looked upon are the old states of the neighbors, see figure 6. When making an MPP where the cells update their own state, some of the behavior have to be "coded between the lines". By this I mean that behavior is reflected by the behavior of a cell in the opposite or counter-situation. This is most obvious when a switch is performed. In the switch there is a risk of losing a message if in G3 either index 3 or 4 does not read the message of the other.

When a message has been read by a neighbor that brings it closer to the target, then this information will have to be realized by the cell who had the message, and will lead to it erasing the message. If there are problems here, then duplication of messages can occur, which would make the processing unsound.

5.6.1 The CAL_CellularAutomaton message passing interface

Before we get into an example of a working CAPM-MPP, a few things must be defined. One of these is how the CAL_CellularAutomaton handles ingoing and outgoing messages while also passing messages from neighbors.

The Update algorithm for the CAL_CellularAutomaton consists of two parts. The first part updates and performs some message passing the internal CALIS_Node, if any. The other part is the grid message passing part. It is important to notice, that messages from the internal CALIS_Node are only sent out on the grid, if the CAL_CellularAutomaton is still empty after grid message passing. This has as a consequence, that if a CALIS_Node has an outgoing message, then it can receive an incoming message, before that outgoing message has made it out on the grid. This is something that must be kept in the back of the head when designing new CALIS_Nodes, to avoid causing unsound state changes.

```
1 CAL_CellularAutomaton::UpdateState() {
```

```
2          CAL_Message deletedMessage = null;
3          if (this.currentNode != null) {
4                  this.currentNode.Update();
5          if (this.currentMessage != null) {
6          if (this.currentMessage.GetTargetAddress().equals(this.myPosition))
               {
7          deletedMessage = this.currentMessage;
8          this.currentMessage = null;
9        }
10     }
11   }
12   CAL_Message newMessage = null;
13   if (this.currentMessage != null) {
14     newMessage = MPP.MessageSwap(this);
15   }
16   if (this.currentMessage == null) {
17     newMessage = MPP.MessagePass(this);
18   }
19   if(newMessage == null){
20     newMessage = this.currentNode != null ? this.currentNode.
          GetOutgoingMessage() : null;
21   }
22   this.currentMessage = newMessage;
23   if (this.currentMessage == deletedMessage) {
24     this.currentMessage = null;
25   }
26 }
```

5.6.2 SCAMP - Simple Cellular Automata Message Passing

SCAMP is based upon the idea that deadlocks and starvation are more or less unavoidable, so it is based upon a concept of the unwinding of deadlocks bring at least one message closer to its target and the others involved not to enter an oscillating pattern. It furthermore requires the grid to be four times as large, keeping the CALIS_Nodes in the bottom right quarter of the grid. This is not at odds with the definition of cellular automata as the grid would be infinitely large and four is not that large a number.

The SCAMP protocol is quite simple and consist of two main behaviors, where the second is split into two parts.

1: MessagePass

MessagePass is the upper line of (c) in figure 17. This is the case where the CA currently contains no message, and therefore can read a message from one of its neighbors. This is done with the priorities: diagonal > horizontal > vertical.

2: MessageSwap

MessageSwap is used when a cell α contains a message m. Depending on what direction m should go, the correct neighbor is looked upon and depending on whether or not the message in the neighbor is equal to m, α either deletes m or not. The second line of (c) in figure 17 illustrates this step. If the m is deleted, then if there is a diagonal message d with α as the next step of its path, d is read and stored by α.

3: TriNodeSwitch

The name TriNodeSwitch should not be confused as to mean, that three cells switch messages. Rather it means that the switch is based on the states of three cells. The last line of (c) in figure 17 indicates how this step is performed. There are basically two cases in the TriNodeSwitch. The

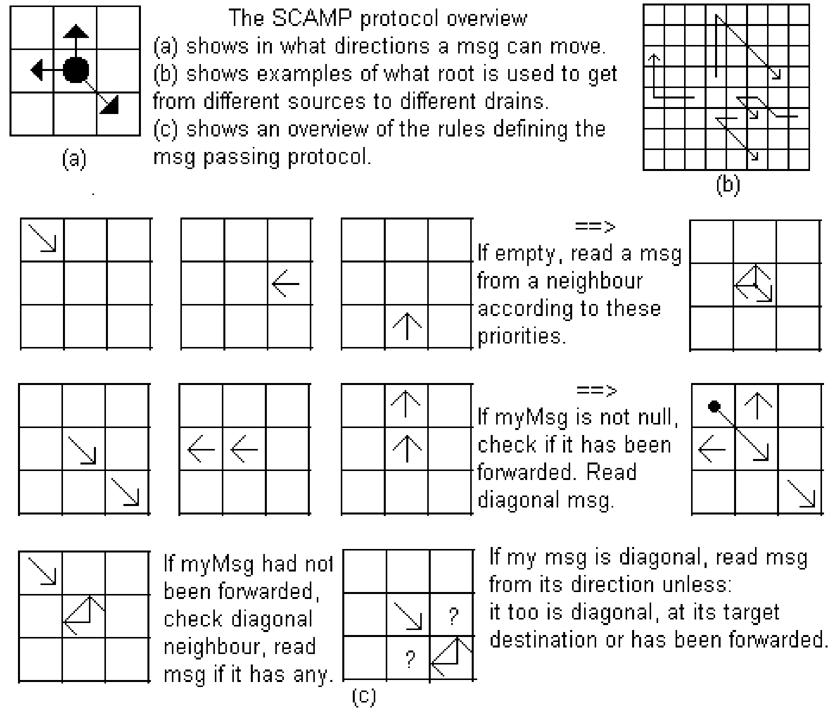


Figure 17: An overview of the principles of the SCAMP protocol.

first case is that the direction of current message m of the current cell α is diagonal, then if the diagonal neighbor contains either a horizontal or vertical message, and these have not been read in the direction they should go, read the message, overriding m. The other case is that the m is not diagonal. Then if the backward diagonal message d has α as next step, then replace m with that message, unless m is equal to d.

Message Direction In the above, I have used the direction a message ought to go, without saying how this direction is known. The message itself, actually knows nothing about direction, it only knows a target coordinate. The direction is computed by the message passing protocol, in the case for SCAMP the algorithm for computing what direction a message would "prefer" to go[27] is quite simple.[28]

- HasArrived
  - This "direction" is returned if the current position equals the target of the message.
- Diagonal
  - This direction is returned if the current position is on the diagonal of the target and at the same time is to the left of the target.
- Horizontal

[27]I use prefer here, because if the direction of a message is horizontal or vertical, it could be forced to move diagonally.
[28]In the appendix, the code for the GetDirection method is shown in A.1.

  - This direction is returned if the current position is above/to the right of the diagonal.

- Vertical
  - This direction is returned if none of the above were returned.

What are the attributes of SCAMP? To see what the attributes of SCAMP are, let us look at the questions posed in the beginning of 5.6:

What is the worst case time for a message to get from a to b? Time complexity in this analysis refers to the number of instructions, i.e. in SBPM assembler instructions and in CAPM logical nodes in the grid. If there is only one message in the grid, then the worst case route is maxLength of horizontal or vertical followed by a maxLength diagonal trip. A worst case route is from the top right element to the bottom left element. This route would be of length $2\sqrt{n} + \sqrt{n} = 3\sqrt{n}$.

If a program, with n SBPM assembler instructions, has been shown to have a worst case time complexity O(f(n)), then the CAPM compilation of this program will, in worst case, need to perform at least O(f(n)) instructions. In addition to this, the ϕ functions introduce g(n) additional CALIS_Nodes. This means that, in worst case, O(f(n*g(n))) messages must be sent. Assuming that only one message per generation comes closer to its target, we are lead to the conclusion that the worst case performance is O(f(n)*g(n)*$\sqrt{n}$). This is of course a very pessimistic estimate, and empirical data in section 6 indicates that CAPM performs a lot better than this estimate. I will return to this question in section 6.3.

What is the average time for a message to get from a to b? The arguments used above for empirical data to estimate the efficiency of SCAMP, also apply to this question. I will return to this question in section 6.3.

What is the buffer size in each node or cell? The buffer size of each cell is exactly 1.

Can deadlock occur? Deadlocks can not occur globally. Deadlocks, seen from the point of a cell or message, are temporary. It is possible for a message to be stuck in the same cell in an unknown time interval, but because every generation brings at least one message closer to its target, the messages, blocking the path of a stuck message, will eventually disappear.

Can starvation occur? If we assume that the program being processed does not contain parts that loop indefinitely, starvation can not occur. The reason why we have to assume this, is more or less the same as in SBPM. If a part of the program loops indefinitely then, in the sequential model, the following statements will never be evaluated. In CAPM it is possible, but not certain, that the message passing of the looping part, will block the communication between other parts of the program, thus starving those parts. To see that starvation does not occur, when the assumption above holds, we have to observe that when a message has reached the diagonal path leading to its target, then it will move one step closer each generation, unless another diagonal message is right in front of it. This argument is inductive so we know that at least one message on the diagonal in question will move forward and therefore the message in question will in time move down its diagonal. By looking at the diagonal as a sub target, the diagonal movement of the horizontal or vertical messages when switching with a diagonal message, does not change the distance from their sub target. The only question left is whether or not there is a point for each message where they are sure that no more diagonal messages will keep them from getting closer to their diagonal. These

points are the upper and left edges of the grid. This is the reason why we have the CALIS_Nodes in the bottom right quarter of the grid. Another point to be made is that the workload of the grid, seen as the number of active messages in the grid, will by structure, oscillate around some equilibrium of (numberOfMessages,n).[29] This happens because as the number of messages being temporarily stuck increases, the number of messages arriving decreases and thus the amount of new messages also decreases. Because of the logical structure of the CALIS_Nodes, as binary trees with an additional reference, most of the sub computations require more than one message to arrive, thus the effect of this load balancing will probabilistically ensure that we do not end up in a situation where each cell contains a message and thereby turns into a kind of sliding-tile puzzle, the effect of which would be a very significant slowdown.[30]

Are all messages guaranteed to arrive? Assuming that all sub computations terminate, then all messages will arrive if no messages are being lost in message passing. The reason for this being the only criteria, is seen in the deadlock and starvation arguments. To prove that no message is lost because a cell "thinks" another will read it and therefore erases it, we will need a case-by-case structural induction proof. As this proof is quite tedious and the number of cases is quite large, a convincing argument will have to suffice. **Do the convincing argument here, using the figure of the protocol.**

Can duplication of messages occur? Duplication can not occur. By investigating the cases of the message passing protocol, it can be shown that it is only when a message is forwarded in its expected direction, that two instances of the same message exist. It can also be shown that this is not a problem, as it is discovered and erased by the appropriate cell. I will not show or prove the non-duplication attribute of SCAMP here, because it is trivial and has many cases. The structure of the proof are the two arguments mentioned.

Summary of SCAMP The SCAMP-protocol, is deterministic, guarantees arrival of exactly one message and it acts like a cellular automaton rule, so it fulfills the defined requirements to the message passing protocol. Unfortunately it lacks, to my knowledge, a provable worst case travel time of the messages, that is anywhere near efficient. I expect that if optimizations were made to the positioning of logical nodes in the grid, then from this optimization a worst case bound could be given. The problem here is that we do not know anything about the topology of the logical node graph, but if we placed the logical nodes in the grid according to some pattern, then it is possible that SCAMP would prove to be somewhat efficient and furthermore the performance would be provable "efficient". Sadly, this positioning algorithm has not been developed during the work on CAPM, so this is something to be investigated in the future. In figure 18 generations 133 to 141 of the simple ongoing example, using SCAMP, is shown. The yellow cells, are the ones who have activated a sub computation, and waiting for evaluation complete messages. The red moving circle, is a Write message and the white is an evaluation complete.

5.6.3 Optimizations

Not knowing what the worst case time for a message to get from a to b or knowing it to be extremely high, is a problem. Finding such a protocol, maybe by relaxing some of the restrictions I put up for myself, could be interesting, as it would result in provable bounds for CAPM. The

[29] The use of equilibrium here is possibly a bit misleading. The effect is more like the evolution term ESS - Evolutionarily Stable Strategy [TSG] page 69.

[30] This is not something unique to SCAMP, this would probably be the case for most CA message passing protocols.

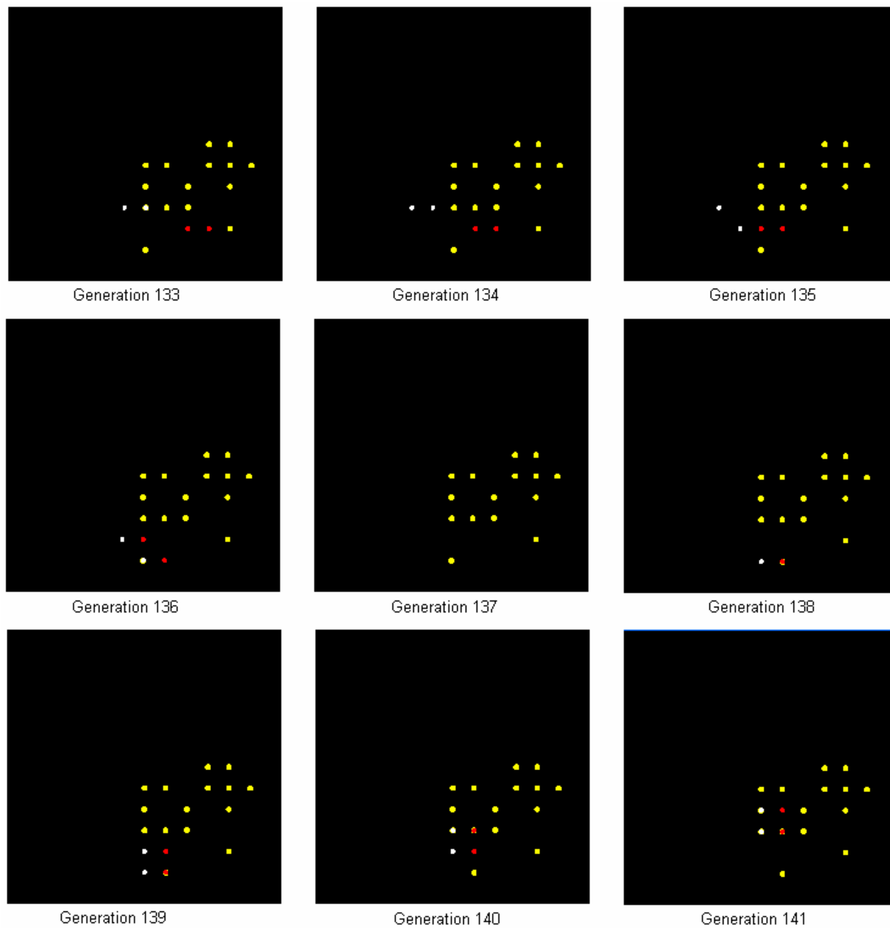


Figure 18: Generation 133 to 141 of the simple ongoing example using SCAMP.

bounds we do know are the ones described in theorem 1: The maximum number of cells that are active in generation g are $\leq (2g+1)^d$. The analysis behind theorem 1 can be generalized to set time bounds for while loops running in parallel, by assuming they exist in different places of the grid, where their activation only interfere in two cases; initial activation and termination.

An obvious optimization of an MPP, is to place the logical cells in the grid, such that the distances between associated logical cells is small, with regards to the paths of the MPP. How this could be done, is currently unknown to me.

Direct Routing I have previously mentioned that the MPP should be implemented in a way, such that if one wanted to ignore the neighborhood definition, this should be possible. This could certainly speed up many programs considerably, as exponential activation (and thereby polylogarithmic time complexities) would be possible. The important thing is that the SCAMP would work on a CAPM architecture, but if we run a CAPM program on an architecture, that supports dynamic lookups, e.g. a GPGPU texture implementation of CAPM, then the program can be run more efficiently by use of DirectRouting. This is like the difference between a classical Turing machine and a Random Access Turing machines, but instead of a polynomial speed-up,

the speed-up is exponential. In section 6 the performances of SCAMP and DirectRouting are compared.

5.7 CAPM design and runtime algorithm

The primary concern in the design of the CAPM Virtual Machine has been to divide the different logical aspects into modules, so that experiments with one aspect can be performed without too much hassle. This has been achieved to some degree, although the current version still lacks some modifications to be as generic as I would like it to be. An example of this is the CAL_Position class. Way too many other classes assumes or knows the position class to be two dimensional. In a generic model, this would have been completely abstracted away by using an interface with the methods needed by the classes using position objects. The only classes that should know the structure of the position objects, are the message passing protocol and the grid. In the current CAPM-VM this could be achieved, by rewriting some of the code, but as it is a prototype this has not be done. In section 5.7.7, I will sketch a possible design of the next version of the CAPM-VM. Now that I have commented the possible lack of generality, I have to say that the structure of the current CAPM-VM have worked quite well. The four basic classes, CAL_Grid, CAL_CellularAutomaton, CAL_MessagePassingProtocol and CALIS_Node, each handles a different aspect of the CAPM-VM and abstracts that aspect away, so the others do not have to worry about it.

5.7.1 CAL_Grid

The CAL_Grid consists of 2 two-dimensional arrays of equal size (maingrid and subgrid) where every cell contains a CAL_CellularAutomaton (CALCA). When the grid is updated, it updates the CALCAs in the maingrid and after these are all in their new state[31], the CALCAs in the subgrid are synchronized with their counterpart in the main grid. This is how the double buffer technique is implemented, and it could have been implemented in more optimal ways. E.g. by letting the CALCA itself contain the old state information. The design choice was made for prototype debugging purposes, with the intend of making it easy to switch the view between the old state and new state of the cellular automata in the grid. This choice have proven to be very useful during the implementation phase, because it is difficult in this programming paradigm to keep track of when and where errors arise.

The CAL_Grid makes the cells in the subgrid available, so they can be used by the update mechanisms in CAL_CellularAutomaton, CAL_Node and MPP. This gives easy access to the state-information of which their actions/state changes depend. The old state information of a CALCA is acquired simply by invoking GetSubGridCellularAutomatonAt with its position given in either two integers or as CAL_Position. The coordinates of the grid have the y-axis "inverted", i.e. topleft = (0,0).

5.7.2 CAL_CellularAutomaton

Parts of the runtime algorithm of CAL_CellularAutomaton have already been shown in section 5.6.1, where the message passing, from the CAL_CellularAutomaton viewpoint was explained. The synchronize method used by the grid in its initialization, simply copies the values and associations from one CAL_CellularAutomaton object to another. As mentioned in 5.7.1 this could have been implemented as buffer values in the class.

[31]Their new state could be equal to their old state.

5.7.3 CALIS_Node

The CALIS_Node is the base class for all logical nodes in CAPM[32]. The CALIS_Node contains the different parts of the update algorithm of the logical nodes. The subclasses of CALIS_Node overrides different reactions to different events. This could be seen in the simple example in section 3.3 where the equals node and the addition node acts differently when having received answers from both of their children. The closest superclass of these two, is the CALIS_BinaryExpression class. That class handles the logical reaction of binary expression operators that require both the right and left side to be finished evaluating. The only thing implemented in the addition and equals[33] is the *int EvaluateResult(left, right)* method. Other classes are much more complex, but there are generally only four methods implemented in the immediate subclasses of CALIS_Node: HandleParentIn(), HandleLeftChildIn(), HandleRightChildIn() and HandleAdditionalIn(). Before we go into examples of how those methods may look, I will in details describe the update algorithm of the CALIS_Node.

The CALIS_Node have for each relative (i.e. parent, left child, right child, and additional association = twin):

- Position ≈ a pointer to memory
- Message received from the relative ≈ A relative-specific inbox
- Message bound for a relative ≈ A relative-specific outbox

In addition to these, the CALIS_Node have an outgoing message and an ingoing message, which are the only messages seen by the "outside" world, i.e. the CAL_CellularAutomaton. The update mechanism consist of basically three steps:

1. If the CALIS_Node contains an ingoing message, handle the message appropriately. This results in the ingoing message being null.
2. Check if the CAL_CellularAutomaton in which this CALIS_Node lives has a message for the CALIS_Node. If it has, set that message as ingoing message.
3. If the CALIS_Node has any messages to be sent, i.e. not all outboxes are empty, then if the outgoing message is null, set the message of an outbox and set as outgoing message. This empties the outbox.

1. Handling the ingoing message
This step consists of two parts. The first part is to identify who sent the message (parent, left child, right child, other ≈ twin). The second part is to handle the message. This means invoking the appropriate handler, e.g. HandleParentIn(message from parent), which the subclasses of CALIS_Node must implement.

2. Checking for new ingoing message CAL_CellularAutomaton
This step handles the ingoing "communication" with the CAL_CellularAutomaton. If a message has as target the coordinate of the CAL_CellularAutomaton in which it is currently contained, this means that the CALIS_Node currently residing in that CAL_CellularAutomaton is the target of the message. The message is set as the current ingoing message of the CALIS_Node.

3. Handling outgoing messages
I have previously mentioned that only when the CAL_CellularAutomaton contains no message, then an outgoing message from the CALIS_Node can enter the grid. This approach is enabled

[32]The term logical node is equal to logical cell entity.
[33]And the other comparison and arithmetic operators

by the CALIS_Node only emptying one of its outboxes at a time. It checks whether the current (if any) message locked, loaded and outbounded have been read by the CAL_CellularAutomaton. If that is the case, the outgoing message is erased in the CALIS_Node, and if any outbox is not empty, then the message of that outbox is set in line to be the next to enter the grid, i.e. set as outgoing message and the outbox is now empty.

This basic update algorithm is common for all subclasses of CALIS_Node and except for a few places, the only methods implemented are the eventhandler methods.

5.7.4 Subclasses of CALIS_Node

The CALIS_Node has several subclasses that combined implements the basic logical functionality of CAPM. The immediate subclasses of CALIS_Node, can be viewed as three different types.

1. Entities or statements
2. References
3. Expressions

1. Statements These subclasses have to do with blocks of code and is the basic building of code structure. They furthermore implements special kinds of sequentialism. An example of a statement, is the CALIS_StatementCollection class (SC), which corresponds to a block. Normally a block has a list of statements and the list defines the order of the statements. Evaluation of the block statement could be viewed as the block evaluating the statements in its list, one after the other. The SC is somewhat different. The order of the statements is unknown and when the SC node is asked to evaluate, it simply asks its left child (an outgoing reference) to evaluate. When the left child returns an EvaluationComplete message, the SC reacts, by sending an EvaluationComplete to whoever initiated the evaluation of it. This means either the parent or the right child.

2. References These subclasses have to do with associations between statements. They furthermore implement a kind of buffer of parallel program counters (PCs) and some of the basic control structures. The CALIS_Reference base class implements the functionality of outgoing and ingoing references. The SC described in the above, can have an outgoing reference as left child and an ingoing reference as right child. The ingoing and outgoing references are objects of the same class, but are marked as either ingoing or outgoing, which defines their behavior. This is not elegant, but it works. Ingoing references act as a kind of buffer, so if many messages are sent to a tree of ingoing references, then the ingoing reference nodes, will make sure that only one message at a time reaches the root of their tree. Currently this is not utilized and beside the example of the number generator in section 3.1.3, it is unknown to me what it could be used for and the risk of starving some request would have to be handled. Because this locking mechanism or single-access is not currently utilized, it could be considered to have multiple other nodes point to an ingoing reference and have the ingoing reference answer using the sender address of messages received. This is however not a choice I have been willing to make before further investigation of CAPM is carried out. The outgoing references act more or less as activators. These nodes form the power grid of the cellular automata grid. By power grid, I mean that the references are the ones igniting the parts or statements of a program.

3. Expressions These subclasses have to do with the basic functionality needed in a processing model. This functionality covers arithmetic, compare and loop functionality. The most basic kind of expression is the binary expression, inherited into several classes. The expression subclasses also contain the phi-expression node and the looping structure. So by now, it should be obvious that

things are somewhat intermingled, and if CAPM should be extended to support more functionality than it does now, the development of other CALIS_Node subclasses would certainly benefit from a more elegant and clean design. The expression class evaluates its left child, when receiving an evaluate "from above". The left child is assumed to return an EvaluationComplete message with a result value in it. When that message is received by the expression node, it reacts, by sending an evaluate message to its right child with the recently received value in it. Most of the time the right child will be an expTarget node. If this sounds familiar, it probably does so because this is actually just an assignment. And again the need for a thorough work trough of the class hierarchy and naming of classes becomes apparent.

Exceptions Something not mentioned in the descriptions of the CAPM-VM implementation is how exceptions are handled. There are currently not any exceptions thrown in the program[34] as it, in my eyes, would complicate the development needlessly. There are however several simple ways in which exceptions could be implemented. One is to simply (using DivZero as example) to check whether the exception case is going to occur and react upon that knowledge and send a message to the output node of type Error and value error type or address of the CALIS_Node in which it occurred. Another approach is to build the exception checks as new types of logical nodes. This could be viewed as a kind of error predicate, that for example the division operator asks before computing a result. The error predicate could be inserted between the right child of the division node and the division node itself, such that if the value of the right child were zero, the error node would react by sending an error message to an output node. It would not send a message to the division node, and as the division node requires a message to evaluate a result, the error would not occur in the inner workings of the division node, removing the need for the programmer or engineer, implementing the division node, to concern about DivZero. It is possible that not all errors could be handled in the ways described above, but I think that the solutions I have provided could go a long way.

5.7.5 CAL_Message

The CAL_Message is more or less a struct. A message consists of:

- TargetAddress
- SenderAddress
- MessageType
  - Types are: Evaluate, EvaluationComplete, AssignValue and Clear. Unused types: ReadValue and Exception
- Value
  - How to interpret the value, depends on the message type.
- InvokerID
  - Can be used to route in reference binary trees. It is possible that the invokerID is not necessary.

[34]DivZero exception will be thrown by JVM if a zero division occurs, but no CAPM-VM exceptions are thrown.

The rest of the attributes of the CAL_Message class are for debugging purposes. No real functionality exist in the CAL_Message class, it is intended to be a stateless data packet.

The CAL_Message class could be optimized considerably by reducing the size of it, as the many messages produced can become quite a strain on memory. This is not done and again the reason is prototyping and debugging.

5.7.6 CAL_MessagePassingProtocol

The abstract class CAL_MessagePassingProtocol, is the class to be extended, if a new message passing protocol is to be implemented. When extending this class, the subclass is required to implemented two methods:

1. CAL_Message MessagePass(CAL_CellularAutomaton ca)
2. CAL_Message MessageSwap(CAL_CellularAutomaton ca)

This two methods have been described in the section 5.6, but to give an overview of what is needed to implement a new message passing protocol, I will try to give the essentials of each of the methods.

MessagePass This method handles the case where the input ca currently contains no message. What is returned, is the new message that the ca should contain. The MessagePass method checks if any of the neighbors of ca has a message and if one or more do, it chooses the one with highest priority. What I mean by priority here, is that the MessagePass should choose one of the messages according to what direction supersedes another. In the case of SCAMP the diagonal direction supersedes both horizontal and vertical.

MessageSwap This method handles the case where the input ca contains a message. The first thing to do in this method is to check whether a message have been forwarded and thus should be erased. The second thing to do depends on the outcome of the first. If the message had been forwarded, then we can simply act as MessagePass. If not then it is possible, depending on the message passing protocol, that a switch should be performed with one of the neighbors of ca.

General points The primary concern, when designing a message passing protocol in CAPM, is to make sure that if deadlocks occur, then they are temporary. Two more things to be sure of, is that non of the neighborhood configurations are ambiguous and can result in message duplication. This happens if two neighbors believe it is their responsibility to take care of a message. The other is non-reflective cases of message swaps, that can result in message loss. This happens if one of the involved parties in a swap, does not read the message of some other cell that expected it.

It is, in my experience, a difficult programming task to get a message passing protocol to work and one has to be careful to reflect the different configuration cases. If the design of the message passing protocol should be generalized in a next version of CAPM, then it would be a great idea to have the message passing protocol in a kind of script language defining cases and behavior and an MPP-checker, that checks all different cases and identifies errors.

This checker would, in a brute force implementation, require exponential work testing that no message duplication or loss occurs.[35] Even though it is exponential, it should be possible, if the variables involved are kept small, that this would enable generic message passing protocols, which could be tailor-made for the program and at the same time guaranteed to be safe. Whether or not it is possible to detect loops and starvation in this fashion, is currently unknown to me.

[35] $O(f(numberOfMsgDirections^{numberOfNeighbors}))$ where f is some function checking for loss or duplication.

5.7.7 The design of the next version of CAPM

If CAPM was to be redesigned, the most important aspects to improve, would be a generic design of the associations between the classes: CAL_Grid, CAL_MessagePassingProtocol and CAL_CellularAutomaton. The ideal version of CAPM would be to have the MCALIS graph as the distributable file and when running a program, the compiler would take a file describing the message passing protocol and grid, and compile these as well. To achieve this generic association interface, it is essential to redesign the CAL_Position class, such that it too is generic and can be compiled along with the grid and MPP.

This would create an extremely generic processing model that, by abstracting the structure of the grid and message passing away, could run a program very efficiently because the machine running a program can supply a grid and a message passing protocol perfectly fitted to the architecture of the machine. It would also be possible to transform the program into a sequential evaluation if needed.

5.8 The CAPM-VM GUI

During the development of CAPM, I have implemented a GUI which I have used to debug the compiler and VM. This GUI makes a lot of information available and furthermore gives a graphic representation of what happens during the evaluation of a compiled program. As the GUI have been used primarily for development purposes, not much effort have been put into user friendliness or the efficiency of the graphics. To avoid using to much precious time to implement the GUI, I have used the GUI builder Jigloo[?]. Another consequence of the GUI having low priority, some bugs are still present in the program and can be somewhat unstable. It can however be used to instigate CAPM programs and compare running times to that of the TIP-VM.

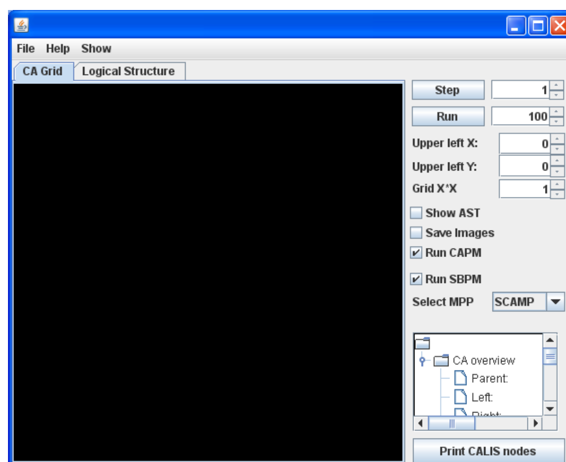


Figure 19: The CAPM-VM GUI start-up screen.

5.8.1 The basics

The CAPM-VM GUI is started without arguments. The GUI is shown in figure 19. The big black area is the CAPM Configuration Viewer (CAPM-CV) where the CAPM configuration is shown.

To compile a program, use File -> Open file. The file chooser is filtered to only show ".tip"-files, so be sure to use this file extension for your CATIP program. Compiling a file is as simple as that.

Running a CAPM compilation is equally simple, to start the VM simply select file and press start. The CAPM-VM is now ready to begin evaluating. This can be done by either using step, where the number of steps (=generations) you want to evaluate is inputted to the right of the *Step* button.[36] If you want to see the changing configurations of the program while it is running, press the *Run* button. To the right of the run button, you can set the time between frames that the CAPM-VM should try to achieve.

5.8.2 Advanced features

In addition to see the program run, the CAPM-VM GUI can be used to gain information about the compiled program and the runtime behavior of it. In figure 20 a screenshot from the evaluation of the simple ongoing example is show. The screenshot shows the grid information (selected by Show -> Show/Hide gridlines) and by left clicking I have selected the Expression node and zoomed in on it. The white and blue lines coming out of the node, shows the associations of the selected node, the blue being left and right child, the white being the parent. If it had had an additional association, then that would have been shown as a red line.

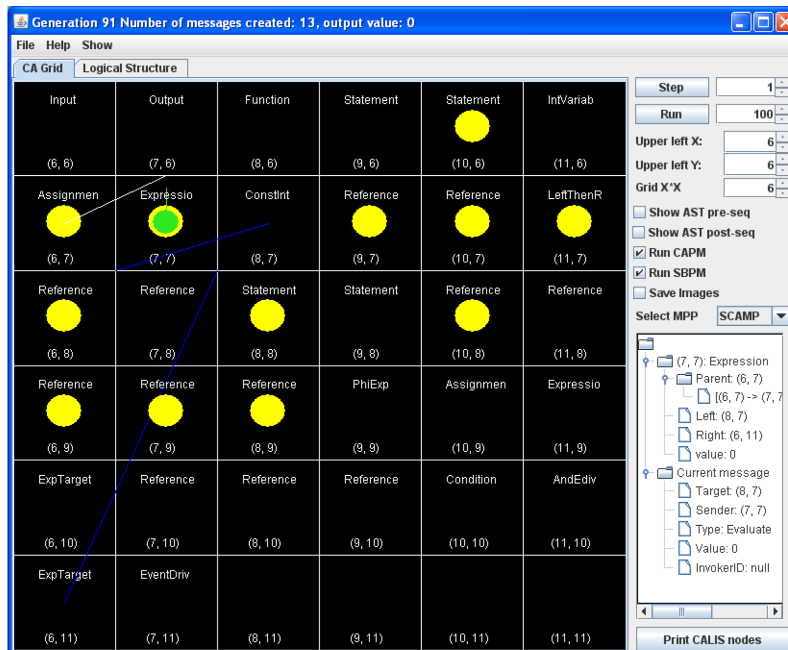


Figure 20: Grid tab.

In the bottom right corner of the screenshot, information about the CALIS node is shown. This involves coordinates of its relatives, messages currently contained by each of its inboxes and outboxes, and the current message contained by the cell in which it resides.

If the *Logical Structure* tab is selected, the logical node that was selected in the *CA Grid* tab, is also selected in *A* in figure 21. Here an AST-like representation of the compiled program can be used to navigate around the structure, this is more or less the MCALIS graph. The order of the children in the MCALIS tree are, top down, parent, left child, right child and a possible additional

[36] Step does not show what happens during the steps, but if you click the CAPM-CV, it shows the state of the time you clicked.

association. The additional association can be seen below the expanded LeftThenRightSeqNode. There are currently some small bugs in the tree viewing, some times the relatives are listed twice or more. So to avoid confusion, simply right click the relative you are interested in, then the tree will select that node for you at the root level. In B, the compiled Code is shown.

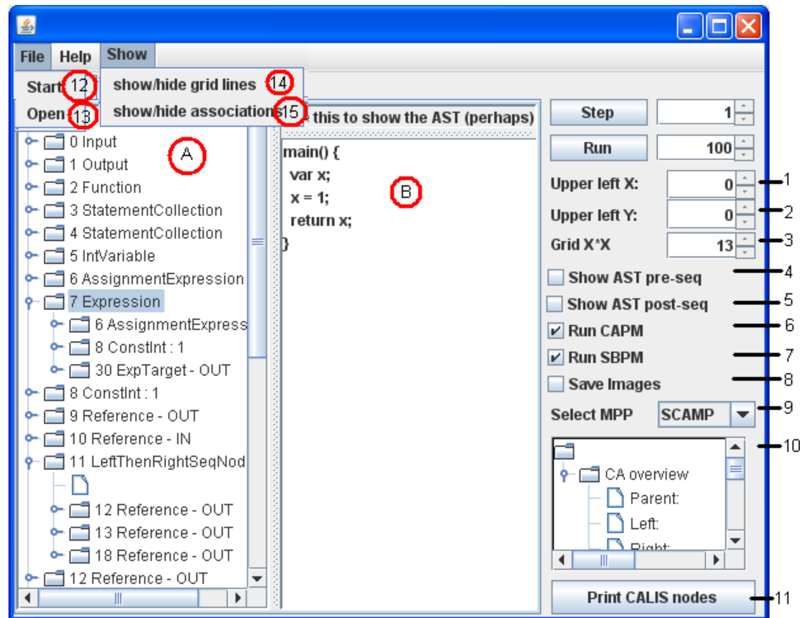


Figure 21: Logical Structure tab.

To give a quick overview of the GUI, I will list the marked elements of figure 21 and after these have been defined, I give a quick use case inspired listing.

Overview of elements marked in figure 21

1. Set the left x coordinate. If this results in going to far to the right, it will not move right.
2. Set the upper y coordinate. If this results in going to far to the bottom, it will not move downwards.
3. The zoom ratio. Defines the number of cells shown, vertically and horizontally.
4. Show the AST before the program is sequentialized. This option have to be set before choosing a file to compile. Notice that it takes a while to generate the AST.
5. Show the AST after the program is sequentialized. This option have to be set before choosing a file to compile. Notice that it takes a while to generate the AST.
6. Run the program in the CAPM-VM. This option have to be selected before choosing a file to compile. Compiling information is printed in the console, so if this is of interest, be sure to run the jar file from a console.
7. Run the program in the TIP-VM. This option have to be selected before choosing a file to compile. The number of stack changes performed by the TIP-VM, is printed in console, so if you want this information, be sure to run the jar file from a console.

8. Save the images. If this is selected, then an image of each iteration is saved in *root-directory/CAPMImages/*. The images are put in a folder with name $\lfloor(generation/1000)\rfloor$ and have names according to their generation.

9. Select the Message Passing Protocol. Currently there are only SCAMP or DirectRouting. The message passing protocol have to be chosen before choosing a file to compile.

10. The overview of the chosen CA.

11. Prints information about the CALIS nodes currently having messages in their inboxes. This has been used for debugging message passing protocols.

12. Start the VM.

13. Choose a file you want to compile.

14. Shows or hides the grid and the text description of cells. Notice that if the zoom value is too large, no textual information will be shown.

15. Shows or hides all associations between CALIS nodes. This can be used to view how well the sorting of cells are, which has relevance for the performance of the message passing protocol. (Except if Direct Routing is used.)

Some simple use cases

- Help: There is currently no help in the CAPM-VM GUI.
- Select cell: Left click on the cell. (de-select by clicking it again)
- Select CALIS node: Left click on the cell in which it resides or select it in the logical structure.
- Zoom: Use scroll button or Grid X*X. If a cell is selected, then it will be kept at center.
- Navigate grid: Right click the cell you want as new center. I.e. right click the grid in the direction you want to go. Alternatively use 1 and 2.
- Navigate through the logical structure, either use the associations on the CA Grid or use the tree in the logical structure tab.

A word of caution I the program being compiled is large, then showing the Abstract Syntax Trees can require a lot of memory. If the program ends up with an empty grid, then it does so because of memory troubles. This can be avoided, to some degree, by increasing the memory size of the JVM. Use the following line, when starting the program:

```
java -Xms256M -Xmx512M -jar capm_1.0.jar
```

6 Testing CAPM

I have chosen Boolean Matrix Multiplication as benchmark problem. This is done for a couple of different reasons. Boolean matrix multiplication is a problem where the amount of work on a random access sequential model (RAM-SBPM) and a parallel circuit or parallel random access model (PaCiPRAM) can be equal within a constant factor. The work needed to multiply an n x n matrix by itself is exactly $O(n^3)$. That the work is never less than the asymptotic time, means that the there is no randomness in the testing, so hopefully no unknown factors should influence the test results.

The time complexities of computing the boolean matrix multiplication on a RAM-SBPM and a PaCiPRAM are $O(n^3)$ and $(\frac{n^3}{\log n})$, which is a significant difference in running time, which should be noticeably both when the benchmark of CAPM is compared to parallel bounds and when it is compared to sequential bounds.

The reason why I use Boolean matrices, instead of just numerical matrices, is that boolean matrices can be used to solve other problems, such as directed or undirected reachability in graphs.

Even though TIP does not explicitly support boolean algebra, this can be done by a few simply tricks. If we want to compute what a1b1 is in:

```
a1b1 = (a1 and b1) or (a2 and b2) or (a3 and b3);
```

we convert the right side into an equivalent arithmetic expression:

```
/*
The principle used is that the result is 1, unless all of the clauses are 0.
If one or more clauses evaluate to 1, then the 1 - 1 in the parentheses,
ensures that the entire multiplication expression is zero and thus
the result is 1 - 0 = 1.
*/
//The values of the variables are assumed to be 0 or 1.
a1b1 = 1 - (1 - a1*b1) * (1 - a2*b2) * (1 - a3*b3);
```

The tests of CAPM vs. parallel and CAPM vs. SBPM and sequential are focused on time complexities and memory usage and related is ignored, but is covered in section 6.5. As TIP does not have boolean algebra, the SBPM is not able to perform lazy evaluation of the boolean functions.

6.1 CAPM vs. Parallel

Comparing CAPM with other parallel processing model, can be done in several ways. One is to simply compare the number of generations used by CAPM to solve the benchmark to the optimal number of iterations of a parallel algorithm. In section 3.4 I concluded that CAPM can not achieve exponential activation[37] so a comparison between CAPM using SCAMP and an optimal parallel time bound, would show sublinear to polynomial vs. logarithmic time. This is a no-contest comparison and not really that interesting. What is interesting, when comparing to parallel models, is the performance of the logical structure of CAPM. This comparison can be performed by using the message passing protocol DirectRouting.

DirectRouting emulates optimal delivery time of messages in CAPM, i.e. if a message in a CAL_CellularAutomaton at position *a* should go to position *b*, then it does so in one step unless the CAL_CellularAutomaton at *b* can not receive the message.[38]

[37] It is possible that some ingenious structure of the neighborhood and message passing could achieve close to exponential activation, but as I am not sure, I assume that CAPM can not.

[38] Remember that a message is not sent, it is received and deleted at the sender when observed to be received.

Common attributes for the tests measuring the parallel attributes of CAPM:

- Simple balancing of arithmetic expressions have been performed. The balancing is not optimal, and in the case of n = 8, it gives a difference of 10 generations.
- Uses direct routing.

6.1.1 Parallel test 1: Hardcoded Boolean Matrix Multiplication

This first test of the optimal performance of CAPM, avoids too much impact of ϕ expressions, and simply tests the raw power of the basic structure of CAPM. The test measures the number of generations needed to multiply the matrix with itself and does so for increasing size of the matrix.

```
1  //Parallel test 1 code template:
2  main() {
3    var  v00, vr00, v01, vr01, v10, vr10, v11, vr11;
4    v00 = 1;
5    v01 = 1;
6    v10 = 1;
7    v11 = 1;
8    vr00 = v00;
9    vr01 = v01;
10   vr10 = v10;
11   vr11 = v11;
12   v00 = 1 - (1 - vr00*vr00) * (1 - vr01*vr10);
13   v01 = 1 - (1 - vr00*vr01) * (1 - vr01*vr11);
14   v10 = 1 - (1 - vr10*vr00) * (1 - vr11*vr10);
15   v11 = 1 - (1 - vr10*vr01) * (1 - vr11*vr11);
16
17   return 1;
18 }
```

The number of generations used by CAPM to perform the multiplication of an n by n matrix can be seen in graph in figure 22. The graph shows a big overhead, which is expected as there is some startup and shutdown work performed by CAPM, regardless of what program is being run.

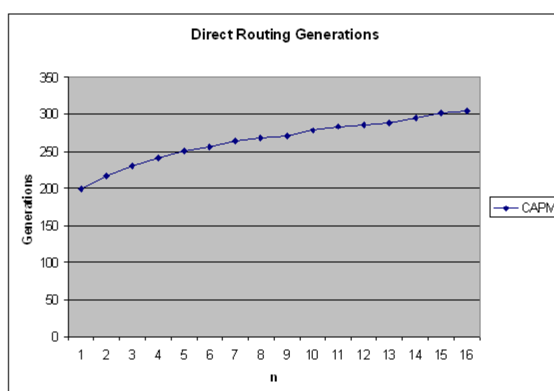


Figure 22: The CAPM test results.

It is a bit difficult evaluating what the increase in time complexity as a function of n is. In figure 23 I have tried comparing the test results to various logarithmic functions, and the CAPM

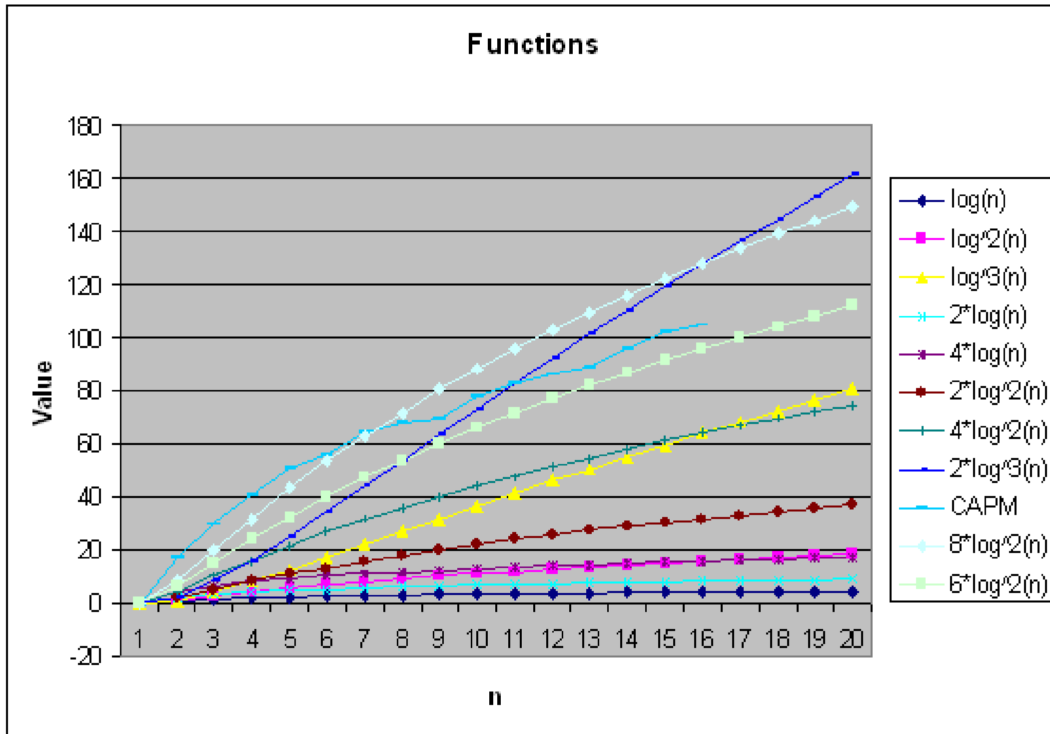


Figure 23: The CAPM data on this graph have had 200 subtracted, as this seem to be the constant overhead.

results, where the overhead is subtracted, appears to be placed somewhere in between 6*$\log^2 n$ and 8*$\log^2 n$, but of course it is very difficult to say whether or not this would be true for n being 50 or 100. But unfortunately the CAPM compiler chokes around n = 15.

By taking a look at what actually happens in CAPM when performing the matrix multiplication we should be able to come up with a qualified estimate, that should support the conclusion that the increase is logarithmic or polylogarithmic.

The program starts by activating all the assignments. There are 3*n*n assignments, so this is $O(\log 3 * n * n)$. As CAPM can only send one message from one CALCA at a time[39], the activation of the next level is slowed down by a factor of two. So we have $O(2 * \log 3 * n * n)$. When activating an assignment there are four steps from the outgoing reference, to the evaluation of the expression. These steps are outgoing to ingoing to root to assignment to expression. So the last logical node being activated will be activated after approximately $O(2 * \log 3 * n * n + 4)$ generations. Now the expression will be activated, meaning an additional $O(2 * \log n + 3)$[40] generations multiplied by two because the answer propagates back.

The constants in the above are approximate values and not really important. What is important is that nowhere in the estimate does time complexity exceed $O(\log n)$. We now see that the estimate of a polylogarithmic time, was too pessimistic and caused by all the different constant overheads, that would have been less dominant if the test had gone up to 50 or 100.

[39]There is also some slowdown in the communication between the CALIS node and the CALCA, but this is ignored in this estimation. The slowdown is probably around 4 or 5.

[40]the '3' comes from the boolean algebra.

6.1.2 Parallel test 2: Hardcoded Boolean Matrix to a power

To test the impact of the while-statements, I will square matrices of sizes 4 by 4, 8 by 8 and 12 by 12, 1 to 10 times. The results of this test can be seen in figure 24. By looking at the averages, it can be seen that the difference in generations, between squaring 4 by 4 matrix and a 12 by 12 matrix, is very small, the 12 by 12 using approximately 20 % more generations than the 4 by 4. Another thing that can be seen, is that the time/iteration, drops almost 40 % from computing only one iteration to performing five. In figure 25 the impact of the while loop is shown, and it seems that the while loop slows down the performance of the matrix multiplication of a factor slightly greater than two. This is in my view acceptable, as it is a consequence of the reload statements. If the need for reload statements could be removed, then the effect would be that the program ran twice as fast.

```
1  //Parallel test 2 code template
2  main() {
3    var counter, v00, vr00, v01, vr01, v10, vr10, v11, vr11;
4    v00 = 1;
5    v01 = 1;
6    v10 = 1;
7    v11 = 1;
8    counter = 1;
9    while( counter > 0) {
10     vr00 = v00;
11     vr01 = v01;
12     vr10 = v10;
13     vr11 = v11;
14     v00 = 1 - (1 - vr00*vr00) * (1 - vr01*vr10);
15     v01 = 1 - (1 - vr00*vr01) * (1 - vr01*vr11);
16     v10 = 1 - (1 - vr10*vr00) * (1 - vr11*vr10);
17     v11 = 1 - (1 - vr10*vr01) * (1 - vr11*vr11);
18     counter = counter - 1;
19   }
20
21   return 1 ;
22 }
```

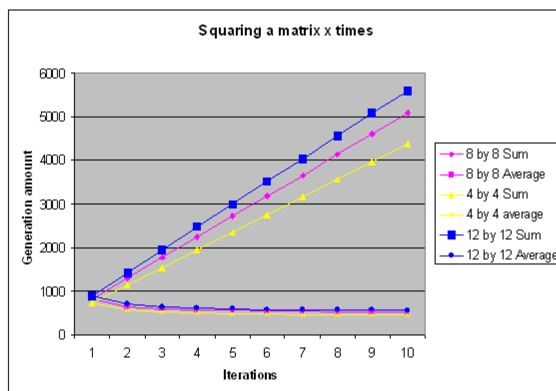


Figure 24: Iterations of while loop using direct routing.

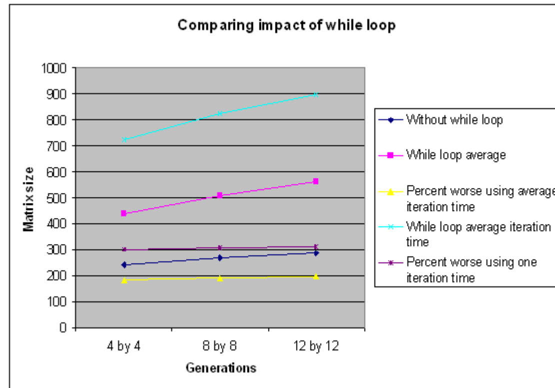


Figure 25: Comparing the matrix multiplication with and without a while loops.

6.1.3 CAPM vs. Parallel conclusion

CAPM can be asymptotic as efficient as other parallel models, if the DirectRouting is allowed. The overheads involved in the structural attributes of CAPM, will however increase the time by a factor which is somewhat significant. The reason why this result is interesting anyway, is that if a GPGPU version of CAPM was implemented, problems as this one could be solved efficiently, by use of the DirectRouting protocol. Another positive thing in this aspect is, that because the neighborhood restriction is abstracted away to the routing protocol and the rest of the CAPM code adheres to the cellular automata principle, the translation to a GPGPU version, of the current implementation of CATIP and CAPM, should be fairly simple, in section 7 a conceptual solution to the translation to the GPGPU version of CAPM is provided.

6.2 CAPM vs. Sequential

Comparing CAPM using SCAMP to the TIP-VM should indicate whether or not neighborhood restricted CAPM has anything to offer as a practical model. This comparative tests between CAPM and SBPM will be more extensive than the comparison with the parallel models, as CAPM is intended to be an alternative to SBPM.

The first test is the same as in 6.1 with one difference. The sum of the matrix entries is returned.

6.2.1 Sequential test 1: Hardcoded Boolean Matrix Multiplication with output

An interesting thing discovered during this test, was that balancing the expressions could actually make the program slower. Though the difference does not seem significant, for example with n = 6 this difference is 2907 vs. 2888, it is nevertheless interesting and gives motivation to implement an optimization of the placement of CALIS nodes.

This decrease in generations is not surprising though, as the positioning of the cells are not optimized and therefore balancing can increase the distance between associated CALIS nodes and furthermore congestion is more likely to occur. The reason why balancing can (and is likely to) increase the distance between nodes in the same expression is because the CALIS nodes are put in the grid according to their creation and thus by default, two CALIS nodes from the same expression are likely to be close to each other. The reason why the risk of congestion increases is that the faster messages are produced, the less time the counter effect has to "react".

The test results of matrix size going from 1 by 1 to 15 by 15 is depicted in figure 26. Even

though the difference between the results from CAPM and SBPM are not that impressive, at the 15 by 15 matrix, CAPM is almost twice as fast (or half as slow) as SBPM.

```
1  //Sequential test 1 code template:
2  main() {
3    var  v00, vr00, v01, vr01, v10, vr10, v11, vr11;
4    v00 = 1;
5    v01 = 1;
6    v10 = 1;
7    v11 = 1;
8    vr00 = 1 - (1 - v00*v00) * (1 - v01*v10);
9    vr01 = 1 - (1 - v00*v01) * (1 - v01*v11);
10   vr10 = 1 - (1 - v10*v00) * (1 - v11*v10);
11   vr11 = 1 - (1 - v10*v01) * (1 - v11*v11);
12
13   return 0 + vr00 + vr01 + vr10 + vr11;
14 }
```

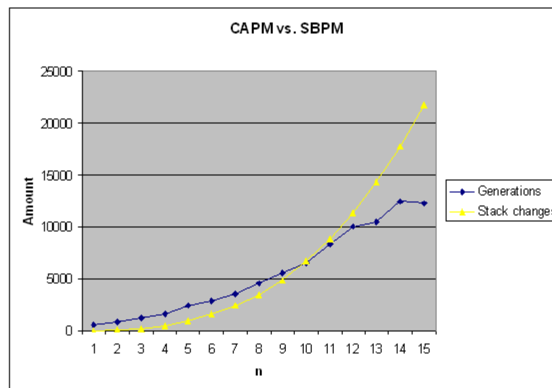


Figure 26: CAPM-VM vs. TIP-VM.

6.2.2 Sequential test 2: Hardcoded Boolean Matrix to a power

This second test of CAPM vs. SBPM investigates the impact of while-statements in the code. To do this, I square a boolean matrix i times and compare the number of generations used by CAPM the number of stack changes occurred in SBPM. What is interesting in this test, is the impact of the ϕ expressions, reload statements etc. involved in while loops in CAPM.

```
1  //Sequential test 2 code template:
2  main() {
3    var  counter, v00, vr00, v01, vr01, v10, vr10, v11, vr11;
4    v00 = 1;
5    v01 = 1;
6    v10 = 1;
7    v11 = 1;
8    counter = 1; //= i
9    while( counter > 0) {
10     vr00 = v00;
11     vr01 = v01;
```

```
12      vr10 = v10;
13      vr11 = v11;
14      v00 = 1 - (1 - vr00*vr00) * (1 - vr01*vr10);
15      v01 = 1 - (1 - vr00*vr01) * (1 - vr01*vr11);
16      v10 = 1 - (1 - vr10*vr00) * (1 - vr11*vr10);
17      v11 = 1 - (1 - vr10*vr01) * (1 - vr11*vr11);
18      counter = counter - 1;
19    }
20
21    return 1 ;
22  }
```

This test shows, as the equivalent in the parallel tests, that the structure of the while loop in CAPM is not as optimal as one could wish for. Comparing the number of generations in sequential test 1 and 2 reveals quite a big difference.

	4 by 4	8 by 8	12 by 12
Without while statement	2486	4566	10075
With while statement	6889	14534	31618

Table 10: Comparing impact of while-statement

Putting the matrix multiplication into a while-statement increases the number of generations needed to terminate by a factor of 3 or 4. This increase actually makes sense, as every iteration of the while-statement requires that all ϕ-function-states are cleared and after that, they are reloaded. This means that the amount of work can increase by approximately a factor of 2 or 3 per iteration of the loop. When the while condition evaluates to false, then all the states of the reload statements will be cleared and this will take approximately the same time as one iteration of the loop would[41], so in the case of only doing one iteration, the expected increase of generations would be at least a factor 3. If the number of iterations increases, then the impact of the extra work of terminating the loop is decreased. This can be seen in the graphs in figure 27 where the generations/iteration is decreased by approximately a third.

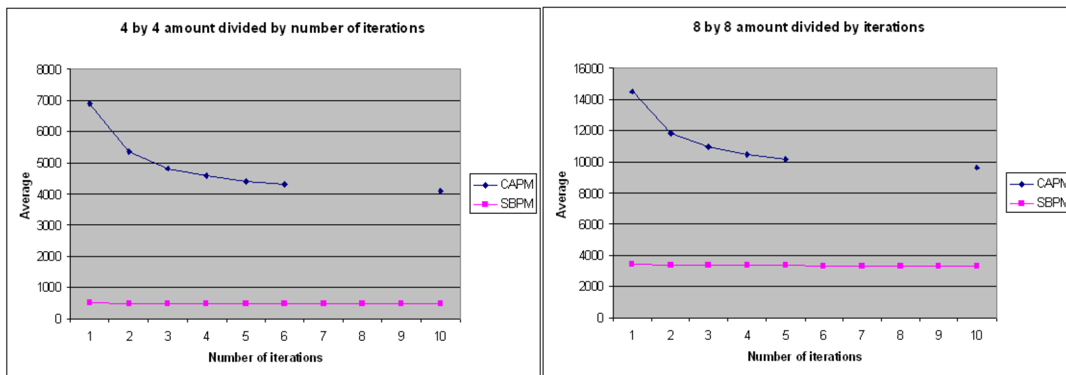


Figure 27: More iterations lessens the impact of terminating the loop.

[41]This is caused by the problem being matrix multiplication. This is not a general attribute of while loops.

6.2.3 CAPM vs. SBPM conclusion

The sequential test 1 showed that CAPM can, when scaled, outperform SBPM and that this happened around the matrix size being 11 x 11. This is a very positive result, as the current version of CAPM using SCAMP is not optimized at all. It is not unlikely that CAPM using SCAMP could be optimized to outperform SBPM already with matrix size being 5 by 5 or 6 by 6.

The sequential test 2, showed that the overhead of the while loops were very similar to the results of the second parallel test, i.e. that the slowdown is around 3 when one iteration of the loop is performed, and when the termination overhead is filtered out, then the slowdown is around 2. This result should boost the confidence in using while loops, but at the same time it should also be noted that when using a while loop, be sure that it is going to run more than twice, as the overhead would otherwise slow the code down by a factor of 3 instead of 2.

6.3 Investigating SCAMP

The difference between the performance of DirectRouting and SCAMP is of a no contest magnitude. This is of course caused by the polynomial activation restriction of SCAMP vs. the exponential activation of DirectRouting.

In section 3.1.5 the questions of worst case and average travel time were posed. These are very difficult questions, so empirical evidence would suffice, at least to give a ballpark figure.

Three examples: The examples used to get the data are: $Matrix^4$ by using while loop. The sizes are 4 by 4, 6 by 6, 8 by 8 and 10 by 10. Travel time is measured from the generation a message is created to the generation where it is received by the target.

Matrix	4 by 4	6 by 6	8 by 8	10 by 10
Size of grid	109 X 109	179 X 179	259 X 259	349 X 349
#CALCA	11881	32041	67081	121801
#CALIS nodes $\leq$	2971	8011	16771	30451
Generations	10681	18869	24126	37139
#MessagesCreated	15756	44670	96485	177660
Average Travel Time $\approx$	28	46	72	110
40 (%) Travel Time (TT) $\leq$	10	9	9	9
60 (%) TT $\leq$	15	19	14	19
80 (%) TT $\leq$	55	86	120	168
Empty grid worst case TT	164	269	389	524
Average vs worst case %	17 %	17 %	18.5 %	21 %
Observed Worst case	162	844	1551	3735
Observed Worst case %	98.7 %	314 %	500 %	747 %
Empty grid worst case $\leq$ TT	0 %	1.4 %	5.6 %	3.4 %

Table 11: Overview of the test statistics of SCAMP. Two iterations of loop squaring the matrix, taking it to a power of 4.

The average travel time is calculated by: $\frac{\#MessagesCreated}{\sum_{i=0..max} i*AmountWithTravelTime(i)}$

The main reason for the messages arriving very late, seem, at least in the 8 by 8 case, to be a combination of congestion and the sender being positioned such that the path to the target is long. An example is (132, 254) -> (214, 163). This message goes up, until it is at the diagonal of the target coordinate. That coordinate is (132, 81). The distance to the "turning point" is 254 - 81 = 173 in the vertical direction. After the "turning point" it travels 82 steps in the diagonal.

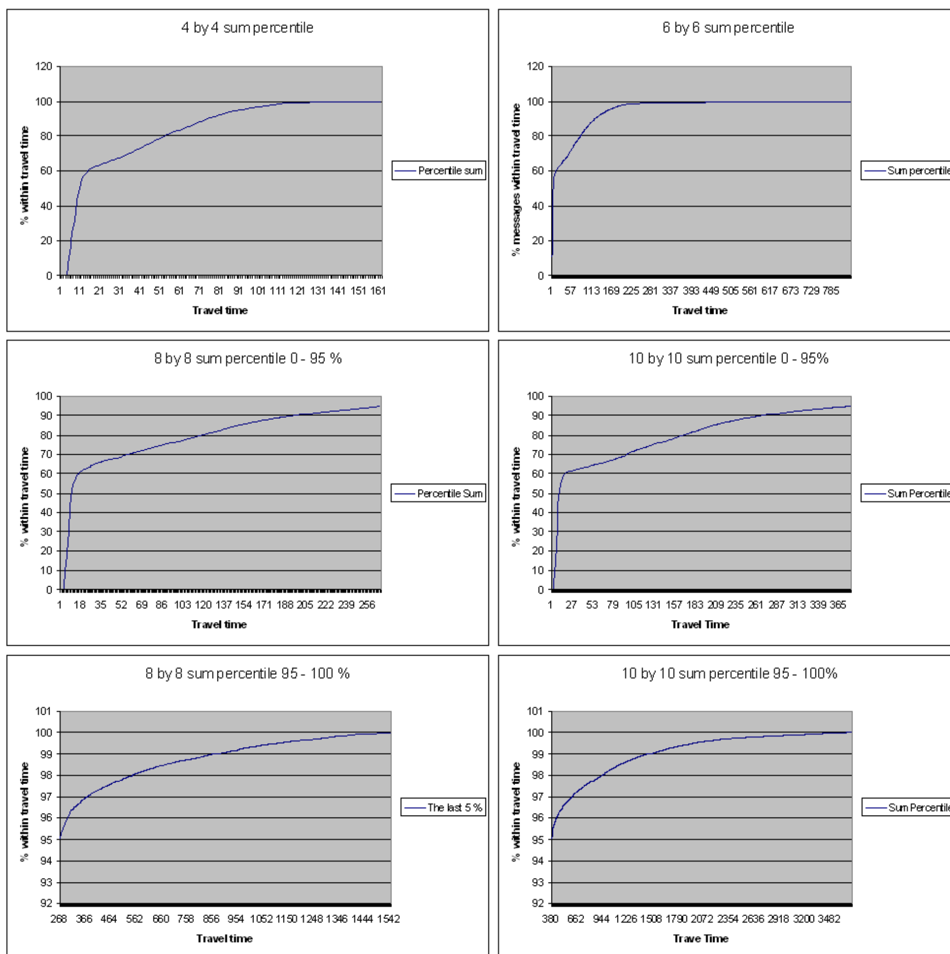


Figure 28: The distribution sum of travel times of messages. Time measured is the difference between the generation a message is created and the generation it becomes the ingoing message of the target.

This gives a total travel time of 255 and that distance is assuming that no congestion-displacement occurs. The travel time of this particular message was 275, only 20 from the optimal.

There are however messages being "stuck in traffic" for a very long time, especially when the size of the matrix increases. I expect that a better placement of logical nodes, would relieve the cells in the upper middle of the lower right quarter of the grid, which seem to be the ones most prone to congestion.

After the tests of SCAMP, it is rather clear that some optimizations of the placements of logical nodes is needed if better and more reliable performance is to be achieved.

6.3.1 Worst case travel time estimate

The tests show that the worst case travel time increase in percentage, as the amount of work increases. This can be seen in the "observed worst case % row". Of course, this worst case is based on the grid being empty, such that no collisions occur. But for the moment, let us assume that it

holds. That the observed worst case percentile increases is not good, as this could indicate that if scaled, then there is the risk that the slowdown of the worst cases outweigh the speedup of the parallelism. However, if we look at figure 26 this does not seem to be the case and the percentage of nodes with travel times worse than the empty grid worst case seem to be stable around 4 to 5 %, but more data would be needed not to make the call prematurely. Optimizations of the positioning of CALIS nodes, would reduce this problem considerably, so these results should not make us lose hope.

6.3.2 Average travel time estimate

The average travel times, indicates that this too increases as the amount of work increases. It does not increase at as rapid a pace as the worst case travel time and most of the messages use very little travel time, i.e. below 20 generations.

6.3.3 SCAMP conclusion

That the travel times increase, both worst case and averages, could be related to the test itself. The test is an exponential activation algorithm, and it is likely that the sudden increase of messages in a short time frame, is what causes massive congestions. It would be preferable to have a message passing protocol that could handle these congestions more efficiently, than what is the case for SCAMP, but then again it is possible that a better positioning of the cells would be a more likely place to find a solution to the congestion problems encountered in these tests. Figure 29 indicates that a lot could be achieved by better positioning of the cells. The edges between the different cells appear to be very entangled.

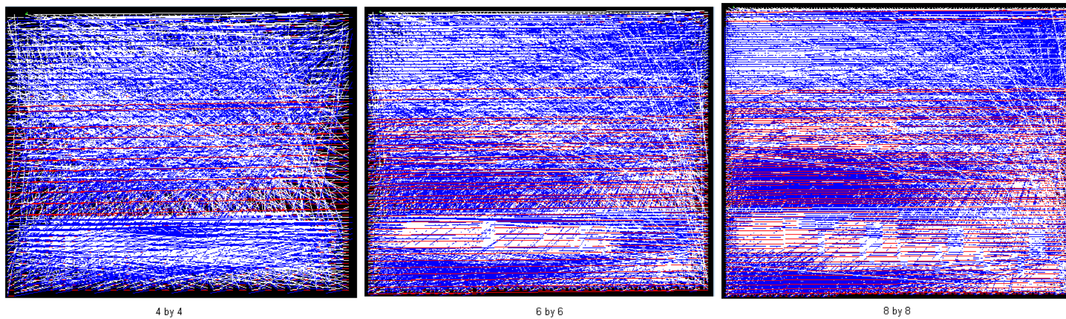


Figure 29: All associations between cells.

With all the bad news handled, it is uplifting to see CAPM using SCAMP beat SBPM, showing that the Cellular Automata Processing Model as a general model, is not hopeless; it has potential.

6.4 Investigating impact of ϕ-functions

It has been shown that the ϕ-functions in while loops, or more accurate, the resetting of ϕ-functions, has a noticeable impact on the performance of the program.

6.4.1 Is the impact of the ϕ-functions acceptable?

In figure 25 the overhead introduced by the ϕ appears to be a factor of 2 or 3, depending on the number of iterations of the while loop. As it is unknown what the performance of other techniques could have achieved, it is difficult to say whether or not this is an acceptable slowdown. That

the factor is two, might seem insignificant, but when using SCAMP, this factor is multiplied by the travel time, which makes the slowdown much more significant. But again, without more data and without a plausible alternative, it is difficult to conclude whether the overhead is acceptable. In addition to this, I do not think it is in the structure of the ϕ-functions that we find the most powerful optimizations. Rather it is the placement of CALIS nodes, that would be the best place to look for faster CAPM computations.

6.5 Memory usage

The memory usage of the CAPM-VM is quite large, but the memory usage I will investigate here, is the number of logical cells needed/used to represent a compiled program in CAPM. The code used to test the memory usage of CAPM, is hardcoded boolean matrix multiplication with and without while loops.

```
1  main() {
2    var  v00, vr00, v01, vr01, v10, vr10, v11, vr11;
3    vr00 = 1;
4    vr01 = 1;
5    vr10 = 1;
6    vr11 = 1;
7    counter = 1;                              //Only with while
8    while(counter > 0) {                      //Only with while
9          v00 = 1 - (1 - vr00*vr00) * (1 - vr01*vr10);
10         v01 = 1 - (1 - vr00*vr01) * (1 - vr01*vr11);
11         v10 = 1 - (1 - vr10*vr00) * (1 - vr11*vr10);
12         v11 = 1 - (1 - vr10*vr01) * (1 - vr11*vr11);
13         counter = counter - 1;}                   //Only with while
14
15   return 0  + v00 + v01 + v10 + v11;
16 }
```

The graph in figure 30 shows the number of CALIS nodes and the total number of CALCA. The reason why the number of CALCA grows faster is, that SCAMP is being used as MPP and hence the grid is at least four times larger than the number of CALIS nodes. It is expected that the number of cells will increase with n^3, because of the workload of the matrix multiplication algorithm. What is interesting is whether the number of CALIS nodes is close to linear if the polynomial increase of work is taken into consideration. This is shown in figure 31.

The two upper lines indicate the number of CALIS nodes per entrance in the matrix. This grows linear. The two bottom lines show how many CALIS nodes are used for each operation in the matrix multiplication[42]. The number of CALIS nodes over the number of operations, converges towards a constant factor. This constant factor is around 15 when there is no while loop and around 27 in the case where there is a while loop.

The reason why the first set of lines are linear, is because when dividing with n^2, we are left with the constant that the second set of lines converge towards.

6.5.1 Conclusion of Memory test

The memory test shows again that the impact of the while loop is a factor around two. What it also shows, is that the structure of CAPM, at least for a problem as structured as the boolean matrix multiplication, only increase the number of CALIS nodes, i.e. the memory usage, by the

[42] It is not taken into consideration that it is boolean matrix multiplication.

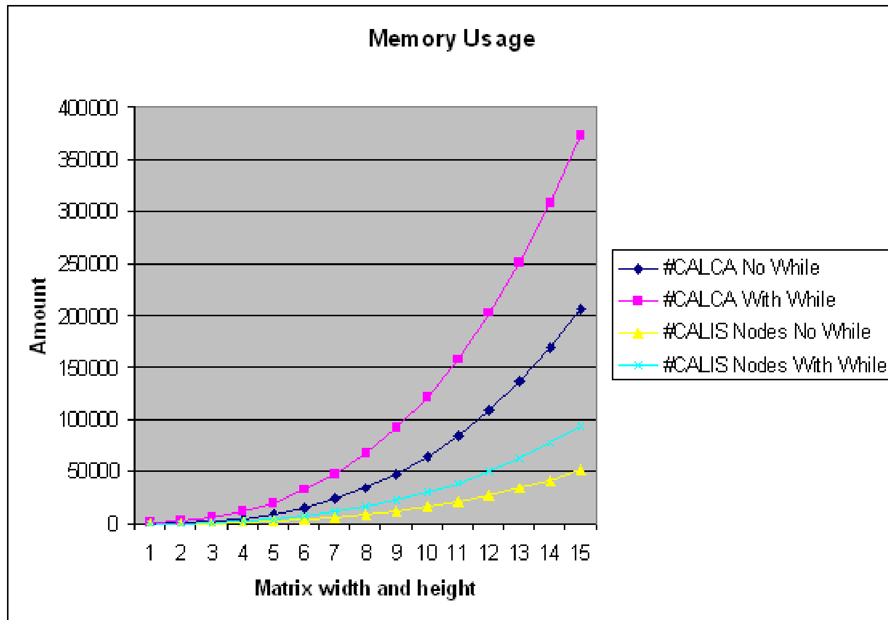


Figure 30: The memory usage of the matrix multiplication with and without while loop.

circuit work amount multiplied by a constant factor. Whether this is the case for more complex programs, remains to be shown.

6.6 Evaluating tests

Considering how much optimization can be performed on different parts of CAPM I am optimistic with regards to the performance of CAPM. There are still a lot of optimization techniques that need to be applied, before the cons of using CAPM instead of SBPM weigh up the pros. The primary optimizations are:

- Reducing the number of ϕ-expressions.
- Placing the logical nodes such that message passing latency is minimized.
- Balancing expressions.
- Balancing the program tree, i.e. the deepest trees are referenced from the top of the statement collection trees, such that those parts that take longer, are initiated first.
- Better message passing protocols.

The tests in 6.1 showed that CAPM can be changed to use Random Access Memory, without doing more than changing the message passing protocol. In addition to the change being easy, the overhead of CAPM supporting two very different types of memory structure, is, in my opinion, acceptable. The tests in 6.2 showed that more optimizations need to be done for CAPM to be a serious competitor to SBPM. There were not any tests of normal code, and the example was inherently parallel, so it was expected that when the numbers increased sufficiently then CAPM would assume a convincing lead. What the performance of CAPM processing "normal" code is,

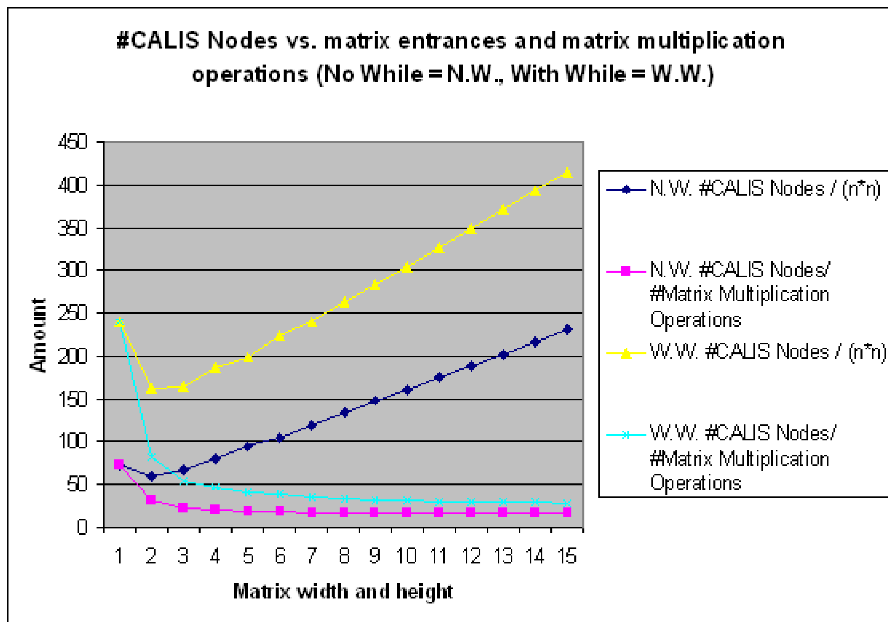


Figure 31: Memory usage vs. number of entrances in matrix and memory usage vs. number of matrix multiplication operations.

remains to be shown, but to be relevant, pointers and functions should be a part of CATIP and the CAPM-VM.

6.7 Test conclusion

6.7.1 Missing tests

There are some tests that I would like to have performed, but because of time restrictions I have been forced to prioritize what tests were the most relevant compared to the time they would take to perform.

Performance of SCAMP in grid with optimized positioning of CALIS nodes This would have been a very interesting test, as it would show how significant the difference in the performance of SCAMP on a non-optimized grid and on an optimized grid. If this test should be performed, it would require either a manual positioning of CALIS nodes or that I implemented a positioning optimizer prototype. Both of these would take quite a long time to do and instead I decided to focus on performing more tests on the unsorted grids and by comparison with the DirectRouting try to infer the information that would otherwise be gained from this test.

Compiler test I have not tested the performance of the CAPM compiler. The reason for this, is that I have not done anything to optimize the compiler, it has been a proof of concept implementation. But during the testing I have learned some things about its performance. When the number of variables increases, there is a great increase in processing time and memory use. This is expected as several of the analyses have not been optimized and especially the implementation of the phi-paranalysis is quite inefficient. In my master's thesis, one of the things I wanted to show

was that it was *possible* to compile an imperative programming language to CAPM, and this have been shown. The efficiency of the compiler, will have to wait for future work on CAPM.

6.7.2 What did the tests show?

The tests are of course based on the current version of CAPM where no dynamic production of new logical cells is implemented. If these were implemented, I doubt that the translation to the parallel paradigm would turn out to be as easy as that of the non-dynamic version of CAPM. This is caused by the observer based principle of the DirectRouting protocol, where the circuit is closed, no empty cells are in the graph. A solution to this could be to create a two-level message passing protocol, where SCAMP was used to creating new logical cells and direct routing used to communicate between existing cells.

With the negative things out of the way, the tests showed that CATIP and CAPM can be run on two different models, and perform quite well on both.

The first tests focused on the parallel model, where the DirectRouting message passing protocol was used to show that the logical structure of the current version of CAPM, could achieve performances close to that of the parallel processing architectures.

The second set of tests showed that the static neighborhood model, as inefficient as SCAMP is implemented nevertheless had performances close to the performance of the random access memory stack based processing model.

The third set of tests, showed that SCAMP, had average travel times much lower than the empty grid worst case travel time. It also showed, however, that if the positioning of CALIS nodes is not taken care of, then there is a risk that the average travel time will get closer to, or maybe worse than, the empty grid worst case time.

The impact of the ϕ-functions appeared to be of some constant factor. I am not sure that this is true, and would advise more tests with lots of conditional statements intermingled with each other, to see what overhead there potentially is. Examples could be designed to create a very big increase in size of the program, when the ϕ-functions are introduced, but I am confident that if a large sample of normal programs were available to test the overhead of the ϕ-functions, then the overhead would be acceptable or in the worst cases avoidable by introducing new variables, such that not too many control flow paths lead to one use of a variable.

The fourth set of tests showed that the impact on memory usage compared to circuit work, was of a constant factor. This was expected, but nevertheless nice to confirm.

All in all, the tests motivates further research in Cellular Automata as a General Processing Model.

7 Future work - Developing CAPM

This chapter contains a number of progresses to be made to mature CAPM. This involves two main points: Extensions and optimizations. After these have been explained, some projects building on the CAPM model will be described.

7.1 Extensions

For CAPM to support a full-blown programming language, some features will need to be implemented. The most important of these, are pointers and functions. Common for these two extensions, is the need for a runtime functionality that allows new CALIS_Nodes to be created at runtime. This functionality involves a new type of message, that indicates that it is a creation of a new logical node and what type of node should be created. Such a message should be routed to a place where it can find an empty CALCA in which it can reside.

7.1.1 Creating new logical nodes at runtime

As there is no control center in CAPM, the allocation of new memory becomes quite complicated, at least if we want the time bound of such allocations to be only a factor slower than the message passing protocol, instead of a power. One way to do this, is to let all nodes know how many empty cells are to the left, right, above and below of them. This information can of course not be up to date, as the information would slowly propagate through the cells. A modification of this solution would be to have the empty-cell values depend on how many "creation" messages have been sent in a given direction. This would of course have to be subtracted from the creation message counter if a cell have returned and gone in another direction.

Another way to solve this problem, would be to devise a scheme for the nodes creating new nodes, such that the pattern and the direction of the creation messages, ensure that without too much overhead, an empty cell is found. A completely different approach could be another dimension of the grid, where only creation messages are passed and let the cellular automata in this dimension efficiently pass the creation messages to an empty cell.

The ideas above are concepts which could be investigated, used as inspiration or discarded if one wanted to implement the dynamic aspects of CAPM.

7.1.2 Adding pointers to CAPM

It is currently unknown how pointers will act in CAPM. One complication of adding pointers, is that in CAPM there are no variables. There are only assignments and targets. The question is whether or not the current concept of variables, using ϕ-functions can be generalized so also pointers are supported. If we for a moment ignore that question, the structure of a pointer in CAPM is quite obvious. It is an entity node, that can supply one of two values. Either it supplies itself or it supplies what it contains. This is basically a root node, with ingoing references as right subtree and a left child that contains a value or another pointer. Returning to the question of runtime structure of the pointer, the value of the pointer, i.e. what it contains, can be viewed as the assignment structure used when variables are used in the current version of CAPM.

7.1.3 Adding functions to CAPM

Early in Master's Thesis i hoped that I would be able to implement function support in CAPM. I therefore tried to envision how function calling in general and recursive functions in particular could be handled. The problem with functions in CAPM, is that we do not have a "program area" like in the SBPM, from where the body of a function can be copied. Instead functions in CAPM

should clone themselves in a way where the associations between parameters and assignments are handled correct.

Inlining all function calls

To avoid the sequentialism of only having the structure of a function represented at one place, I propose to perform aggressive inlining. I would expect that all functions in CAPM are inlined by the compiler, at least the times it is possible. In cases of function pointers this picture gets a bit obscure. Recursive functions should be also be inlined, but the internal self reference[43] will need to have a clone reference to the inlined function, such that, at runtime, if the inner function call is invoked, then it can clone the parent function. This aggressive inlining could lead to the size of the programming growing exponentially during compilation, so either the program should not be coded in ways where such problems arise or some functions could be represented in a central manner.

Function calls buffered args In figure 32 the way parameters are passed is depicted. What needs to be done, is to copy the method structure, in a way such that the inner function calls of the method are associated with the correct variables. Another problem when implementing the function is where the return values should go. Is it returned with the evaluation complete of the function node or does it act as an assignment expression? There are many unanswered questions, but the main point is to create a runtime routine, where the recursive function calls, can make the function clone itself and do it in a way such that the parameters are associated to the correct assignments(/ϕ-functions).

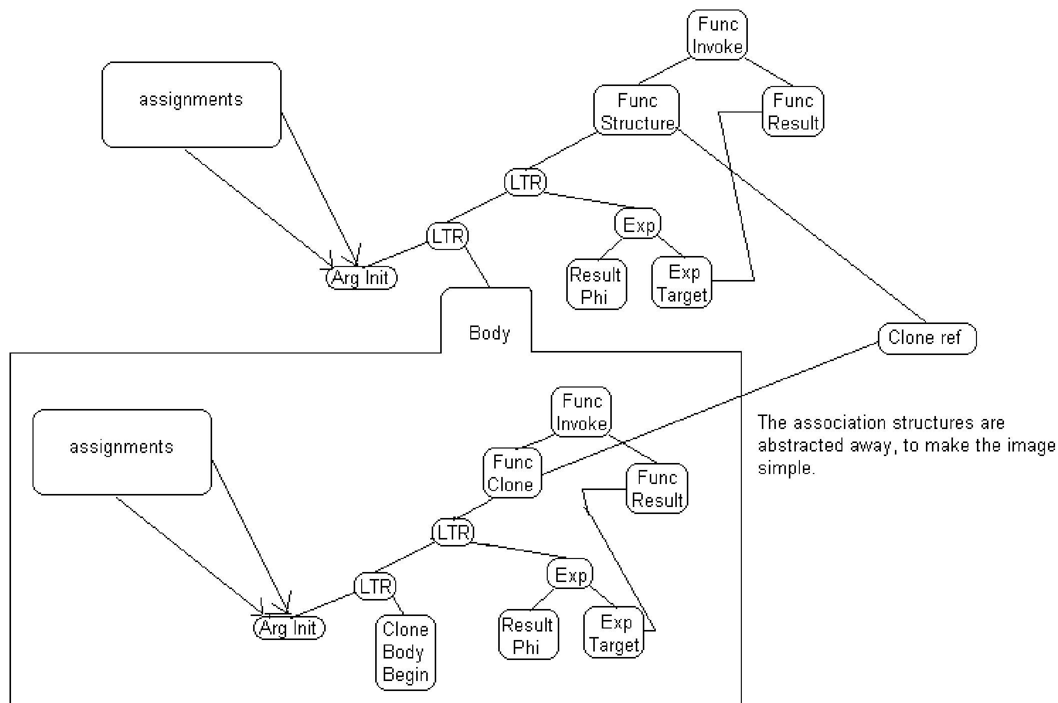


Figure 32: A conceptual depiction of the CAPM functions using cloning. The init args should ensure that the values of the args are defined by the parent function, not the newly cloned function.

[43]If two functions are recursive by invoking each other, then that relationship will have to be discovered, such that the inlining does not go on forever.

7.1.4 Language extensions

I have previously mentioned parallel foreach statements and binary trees as fundamental collection type. There are undoubtedly many more language control structures and standard library components to be identified, but as this have not been the focus of my master's thesis, the suggestions above will suffice.

7.2 Optimizations

7.2.1 Optimizations of the ϕ-functions

The primary optimization of the ϕ-functions is to reduce the amount of memory they use. An easy way to reduce the memory usage of the ϕ-functions, is to collapse several ϕ-functions into a single ϕ-function with multiple targets and replace the functions that have been collapsed with EventDrivenIntVariable-nodes. This would apply to equal ϕ-functions sharing evaluation path. Merging ϕ-functions would certainly optimize the matrix multiplication example and very likely reduce the impact of ϕ-functions in general. The analysis could be handled by transforming the static analysis Available Expressions[STATIC] into an Available ϕ-function analysis.

Another optimization to the ϕ-functions is to replace ϕ-functions where no conditions or clear events are present with a simple EventDrivenIntVariable-node.

A last optimization, is to place identity assignments in the program, such that the number of assignments that can have defined the value of a variable at a given point, is minimized.

7.2.2 Optimizations of placement of logical nodes

In the test section it was shown that a place where optimizations could have a great impact, was to place the CALIS nodes in the grid, such that the distances between the all nodes are minimized. It was furthermore shown that that up to around 70 percent of the messages arrived within an acceptable time bound without no optimization of the placement of nodes. This could indicate that an analysis could focus on the remaining 20-30 percent.

7.2.3 Parallelizing static analysis

Another main area in CAPM to optimize, is to extend the sequential analysis, with parallelizing analyses. Extensive research of this area has been performed over the years and application of the techniques developed, would undoubtedly improve the performance of CAPM.

Classic Loop unfolding Loop unfolding is a classic technique, used to pack as much work into each iteration of a loop as possible. This technique takes into consideration how many processors are available, which in CAPM could be used to optimize to some maximum memory bound.

CAPM Loop unfolding In CAPM the loop unfolding technique could be extended by developing a control structure where the next iteration of a loop could begin evaluating before the current is finished. This could increase the performance of loops with alternating control flow paths inside.

Other well-known parallel optimizing techniques Classic threading techniques could work well in CAPM, although some care should be put into avoiding the communication of one thread starving the communication of another. This could be done by keeping them in separate areas in a way such that the only overlapping communication is inter-thread communication. In figure 33 the concept of CAPM threading is shown. The concept depicted, can result in starvation but how to avoid it, is currently unknown to me.

Another parallel technique is recursive functions, the prerequisites of which have been mentioned. The efficiency of recursive functions, depends strongly on the efficiency of the allocation of new logical nodes, so to use the recursive functions for efficiency purposes, the allocation would need to be efficient.

Figure 33: A conceptual depiction of threading in CAPM.

7.2.4 Branch prediction

It is possible that CAPM could achieve better performances if branch prediction was introduced. If the condition of an IfThenElse-statement takes a long time to process, then it is possible that the evaluation of the condition could be performed in parallel with both the then- and else-body. If no side-effects are present in either body, then this ought not to introduce errors, as a simple modification of the ϕ-functions would ensure that the correct assignment was used. I have not more than scratched the surface of this optimization, but it would be something that should at least be investigated if anyone did more research into CAPM.

7.3 Spin-off projects

7.3.1 Implement CAPM on more suitable architectures

Cellular Automata Modeling Environment and Library (CAMEL) is a framework that can be used to create cellular automata models. It could be interesting to implement CAPM in such a CA environment, as CAMEL could be an easy way to use parallel computing to process CAPM.

Another approach would be to implement, as I have mentioned in the beginning, on the GPU. In the beginning of the work on my master's thesis, I hoped that I would have the time to implement a GPGPU version of CAPM. Because of this, I have a conceptual design of the implementation of CAPM in textures. This is depicted in figure 34 (A), where the additional association is not considered, because that concept was not present at the time this drawing was done. The concept depicted, is to implement a 11-texture solution, where all the textures are of equal size and a cellular automata is all the pixels in a coordinate, as depicted in figure 34 (B).

Figure 34: A conceptual depiction of CAPM on the GPU.

I will not describe the entire solution here, but I have a 13 page description of how the concept in figure 34 works. As I mentioned in the prologue, I have in 2006 had the course GPGPU, where I and two others implemented the graph reachability problem (GPGPU-GAP) in textures

and processed it via fragment programs. The GPGPU-GAP was quite simple, but apart for a few extra functionalities in a GPGPU-CAPM, it is only the complexity that is different, not the fundamental principles used. This fact convinces me that a translation to GPGPU-CAPM, should not be that difficult.

It might appear that the development of the GPGPU solution, has been a waste of time as it has not been implemented, but the conceptual drawing of it actually helped me implement CAPM, as it gives a very clear picture of the different abstractions in CAPM.

The GPGPU model could be optimized by using direct routing for structures that have been created and use SCAMP when new logical nodes are created. These two message passing protocols would have to be implemented in two different textures and some coordination between them would have to be implemented. It would however increase the processing speed considerably.

7.3.2 Deconstruct the CAPM runtime algorithm

In section 3.1.3 I wrote that if CAPM could be implemented with satisfactory performance by abstracting away many sub problems, then a deconstructionstic approach could be followed. By this I mean that a translation of the CAPM algorithm, to much simpler Cellular Automata, could speed up the performance and furthermore, by implementing it in much simpler circuits, some of the goals mentioned in the motivation, i.e. many different aspects of cost efficiency, could be achieved.

The ultimate goal would be to reach a two-state cellular automata, which by configuration, would perform the complex rules performed in CAPM. It is however possible, that the size of each logical cell in such an implementation would be too large for the practical model to be practical.

Another approach to deconstructing the rules in CAPM, would be to implement it as a 3D cellular automata, where the third dimension, was used to achieve the logical operations and the abstract behavior of CAPM only exist at the surface. A conceptual drawing of this, can be seen at the front page of this Master's Thesis.

7.3.3 Cyber Foraging

Cyber foraging is a term[44] that describes an opportunistic approach to borrowing processing powers of nearby processing devices, of very varied quality. The idea is that if your mobile device discovers any other mobile devices in the area you are in, it will try to delegate work to these devices, speeding up its processing capacity and decreasing power consumption. This of course works two ways, so if your device is not doing anything, then other devices can use it to process their work.

This of course has several requirements, which I will not go into here, to both the devices and the programs being processed. I believe that CAPM can utilize this concept efficiently, as the processing of CAPM is highly parallelizable, although communication between the different parts requires some bandwidth to be used. The idea is basically to split the grid into four or more parts and then communicate the state of a part of the grid to another device. Communication between the devices running the different parts of the grid, is needed to ensure the movement of messages, but if it can be asynchronous and efficiently implemented I would not expect this requirement to pose to much of a problem.

Another problem is how reliable other devices are and how often they are lost without returning a result. To avoid too many problems here, the grid configuration could be saved just before delegating parts of it, and if connection to the device is lost, we have to start from that configuration again. So depending on the reliability of other devices, a synchronization should occur every k

[44] coined by 2001 by M. Satyanarayanan from Carnigie Mellon (CMU) in his "Visions and challenges" for pervasive computing.

generations, creating a saved configuration that can be loaded if one of the devices involved is lost during the computation.

This concept is depicted in figure 35.

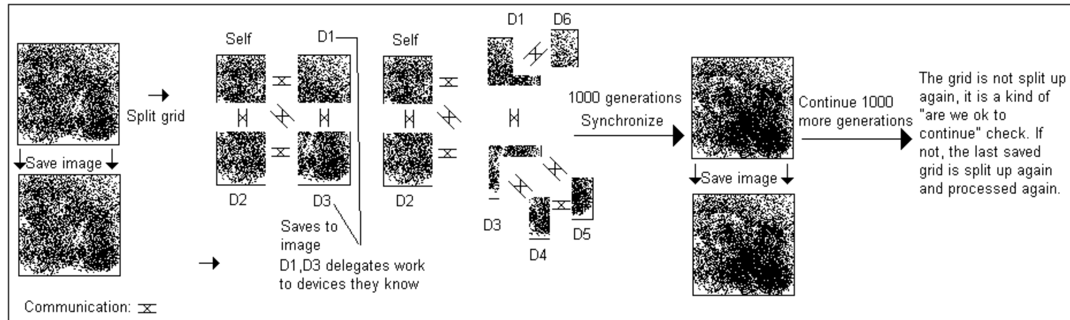


Figure 35: A conceptual depiction of the CAPM - Cyber Foraging.

The CAPM cyber foraging, assumes that the CAPM programs are processed by an internal processing architecture, that efficiently processes CAPM grids, but do not have a CAPM-processor that in one step can update the entire grid, rather it uses x iterations to perform the update.

This leads to the conclusion, that if CAPM is at least as efficient as SPBM or the CAPM program could be produced so that areas to delegate are easy to identify, then Cyber Foraging be used in an easy way to achieve better performance. I have not considered the security issues, but I expect that they could be handled by obfuscating associations, in a way inspired by NAT, so that the structure of the program being run is hidden. I am however not experienced enough with security, to come with a solution to the problem.

This concept also translates to grid computing and other parallel processing models.

8 Conclusion

8.1 Returning to the initial questions

To see whether or not anything was learned in this project, let me pose my initial thesis as questions:

1. Can a cellular automata processing model be an alternative to stack based processing models?
2. Can the performance of a cellular automata processing model be efficient?
3. Can the performance of a cellular automata processing model be scalable?
4. Is it possible to create a high level parallel programming language that enables the programmer to easily utilize the powers of a cellular automata processing model?

and answer each of them:

1. Alternative Yes - The processing model, at least with the abstractions in CAPM, show that the cellular automata processing model can be an alternative to stack based processing models.

2. Efficiency Yes - In the tests it was shown that for matrix multiplication, the cellular automata processing model can outperform the stack based processing model. It would however be prudent to investigate this further, when functions and pointers are supported by CAPM and different optimizations have been implemented, e.g. better positioning of CALIS nodes. With these precautions out of the way, I would say that the work in my master's thesis provides quite strong indications for CAPM being potentially very efficient.

3. Scalability Yes - It remains to be shown that CAPM is truly scalable, but the memory tests gave a strong indication of CAPM using linear or less cells to perform a certain circuit work amount. It also remains to be shown if the scalability of message passing would need the suspension of parts of the rules for cellular automata, by use of wormhole routing and other similar techniques or it could be efficiently scaled without such "tricks". The precautions from 2. also applies here.

4. Accessibility Yes - It is not only possible to utilize the powers of CAPM with a high level parallel programming language, it is possible with a "normal" imperative language.

8.2 Returning to the motivation

In section 2.3.2 I listed some different questions intended to motivate both the reader and myself. My work have only covered a few of the questions posed. This was expected as most of the questions were long term implications of the cellular automata processing model. The questions posed were:

1. What could a practical implementation of Cellular Automata as a processing principle look like?
2. What are the strengths and weaknesses?
3. What kinds of higher level abstraction tools can be used efficiently (in any sense of the word) in CAPM and what high level abstraction concepts can CAPM be used for.
4. What kinds of higher level abstraction tools can not be used efficiently (in some sense) in CAPM?

5. What computer architectures are possible when memory cells update themselves and reading only is done in static neighborhoods. Could it be possible to create other kinds of processing units?
6. What are the economic consequences of such an architecture?
7. What are the attributes of power consumption for CAPM?
8. What types of algorithms are going to be essential to CAPM and what known problems and phenomena do they relate to?
9. Could concepts in CAPM be used for the robotics industry, in massive agent environments?

Even though not more than a few questions were directly answered during the work on my master's thesis, I will provide answers to all but one questions. It is important to note that the answers not acquired from the work on CAPM are merely based on hunches and intuition.

1. Practical Implementation A practical implementation could look like CAPM. CAPM could be viewed as one way of implementing a cellular automata processing model and just within this model a lot of different models could emerge. The defining attributes of CAPM are:

- Sequentialism by ϕ-functions.
- Encapsulated logical nodes associated in a binary tree graph.
- Control flow handled by evaluation wave.
- A guaranteed arrival message passing protocol.

What other practical implementations could arise, I dare not predict.

2. Strengths and weaknesses The primary strength of CAPM is that it can handle a great workload, without too great a slowdown. The primary weakness of CAPM is that it can not do any single thing fast, it can only do a lot of things "slow" at the same time.

3. High level abstraction tools This is a very difficult question to answer. I hinted at garbage collection in section 2.3.2, but after experiencing the different twists (e.g. ϕ-functions) that occurred during the implementation phase, I am very reluctant to try to provide an answer to this question. One thing I would expect, is that the object oriented programming paradigm would translate to CAPM in a way where both "models" would gain something. Exactly what, I do not know.

4. Efficiency of high level abstraction tools I will not try to answer this.

5. Impact on computer architecture I do not know enough about hardware to give any qualified answers to what the impacts of CAPM might be, but I would think that CAPM could be used to create a computer without a CPU and have an architecture where the memory constantly updated itself. The future work section, furthermore contained the cyber foraging example, which was an example of the easy way in which the processing of CAPM could be delegated.

6. Economy I mentioned some circumstantial evidence in the motivation section, for the economic prospects of a cellular automata processing model. I still believe that a cellular automata model would be much cheaper than the CPU and RAM approach. I am however not sure that this is the case for the current version of CAPM or if CAPM should be translated to a more primitive model, for the economic benefits to "kick in".

7. Energy Again I refer to the motivation section, where some different arguments for the energy economy of CAPM being better than SBPM.

8. Essential algorithms The defining attributes described in *1. Practical implementation*, covers the essential algorithms of CAPM.

9. Massive agent environment The physical interpretation of SCAMP, could indicate that the message passing protocols could be viewed as the messages are independent agents, moving around. Whether or not this could be used anywhere else, I will leave unanswered.

8.3 Conclusion

The tasks I set out to solve, have been solved to some degree. In CAPM it is possible to write "normal" code and compile it to a cellular automata processing model. Unfortunately this "normal" code does not support pointers and functions.

I think that one of the reasons why the development of a General Cellular Automata Processing Model have been almost non-existing, is that people have looked at the model and perceived it to be a parallel processing model and jumped to the conclusion, that the performance should be compared to the parallel random access architectures. I have come to the conclusion, that it is a mistake to be focused on the parallel nature, when in my view the interesting comparison is this:

> Can a powerful central processor using random access memory be avoided, by having active static access associative memory?

What the cellular automata processing model can be used for, is not, as commonly perceived, to achieve a great parallel performance, but rather avoid random access memory and retain the performance of the random access by using a parallel architecture.

In the tests of my implementation of CAPM, which is very simple, naive and in need for a great deal of optimization, it turned out that even under such conditions it was possible for CAPM to achieve performances not only close to the stack based random access processing model, but when scaled actually became more efficient.

This leads to the following conclusion:

> A high level programming language without pointers and functions can be compiled to a cellular automata processing model and the cellular automata processing model can potentially be an alternative to stack based computing that uses random access memory. It is furthermore plausible that this also holds true in the case of the high level programming language supporting pointers and functions. The essential question, which have been answered to some degree of this master's thesis, has turned out to be whether or not active memory with a static neighborhood is less, equal or more powerful than a central processor with random access. In the case of matrix multiplication, active memory with a static neighborhood has been shown to be more powerful than the central processor with random access architecture.

9 Epilogue

When I started working on my Master's thesis, I had a rather clear idea of what it would turn out to be, what it exactly was that I wanted to do. But the path from idea to realization is both long and uphill. The prototype I have ended up with, is very similar to the image I had in my mind, when the work had barely begun. The main difference is that many of the foggy areas of the image now stand clear. I had the feeling in the beginning that the idea I had was a good idea, I just did not know what it was. In the beginning I thought that the main alternative the cellular automata model provided was the alternative to the central processing unit architecture. What I now find is the primary alternative, is the alternative to random access memory by using the parallelism inherent to the cellular automata processing model.

In the beginning of the project I spent a lot of time working out proof of concept sketches to convince myself that I would not end up with nothing. Some of these proof of concepts were how to implement functions, how to implement pointers and how to implement the GPGPU part. Unfortunately there was not enough time to implement all these parts and just getting variables to work properly, turned out to be a much more difficult task than I had anticipated. Parts of this initial work, have been documented in the future work part of this document, but as many parts of CAPM have changed between the time I developed the abstract solutions to spawning/copying of nodes, function calls and pointer allocation and the time when I finally had the current prototype working, I have decided not to describe them in depth.

I could probably have spent my time more wisely, using less time on things that became future work and more on the parts actually implemented in CAPM, but I felt that it was important to have a clear image in my mind of how the entire TIP language could be implemented in CAPM, such that I did not abstract any important problems away, which could show themselves later on as impassable roadblocks.

That my suboptimal implementation of CAPM using SCAMP, can beat a random access SBPM (RASBPM), shows that the cellular automata processing model has great potential and should be considered a serious candidate for the next generation of computer architectures[45].

I have learned a lot about computer processing during the work on my master's thesis. Many questions arose during the work and many things I took for granted, e.g. how to use variables, turned out to be more associated with an architecture than being "the only way to do it". I have heard people say, that you can not master one language, before you master two or more. This I believe also applies for understanding computation. And even though I would not say I master either RASBPM or CAPM, I will however say that have gained deeper understanding of both.

[45]Even my implementation was not able to completely restrain the powers of the cellular automata processing model.

A Example Appendix

A.1 The GetDirection method used by SCAMP

```
1 public CAL_CRQSO_Enum GetMessageDirection(CAL_Position myPosition,
    CAL_Position target) {
2   CAL_CRQSO_Enum checkDirection;
3   if (myPosition.Equals(target)) {
4     return CAL_CRQSO_Enum.HAS_ARRIVED;
5   }
6   if (myPosition.GetPosX() - target.GetPosX() == myPosition.GetPosY()
7       - target.GetPosY()
8       && myPosition.GetPosX() < target.GetPosX()
9       && myPosition.GetPosY() < target.GetPosY()) {
10    // Its not diagonal if it diagonally below its point.
11    checkDirection = CAL_CRQSO_Enum.DIAGONAL;
12  }
13  // Check horizontal:
14  else if (myPosition.GetPosY() - target.GetPosY() <= myPosition.GetPosX()
15      - target.GetPosX() &&  myPosition.GetPosY() < target.GetPosY()) {
16      checkDirection = CAL_CRQSO_Enum.HORIZONTAL;
17  }
18  // Check vertical:
19  else {
20      checkDirection = CAL_CRQSO_Enum.VERTICAL;
21  }
22  return checkDirection;
23 }
```

B Bibliography

In the beginning of the work on my master's thesis, I found that not much literature existed on the subject I investigated. I furthermore realized that many of the parallel processing models I initially thought would be a source of inspiration for me, all contained different attributes, e.g. parallel random access ram, that made them unsuited for translation to CAPM. For this reason I do not have lots of pages from the same book or source, rather it is bits and pieces from lots of different sources.

I have divided my literature into two sections, primary sources and secondary sources.

The primary sources are the literature that I have investigated thoroughly, although without much result, and these should be viewed as the curriculum of this master's thesis.

The secondary sources are primarily documenting references.

B.1 Primary Sources

References

[WIKICA] Unknown `http://en.wikipedia.org/wiki/Cellular_automaton` A wiki on Cellular Automata

[PAPA] Christos H. Papadimitriou *Computational Complexity* ISBN 0-201-53082-1, Copyright 1994, Reprinted with correction August 1995, Chapter 15

[PAAA] F. Thomson Leighton *Parallel Algorithms and architectures: Arrays - Trees - Hypercubes* ISBN 1-55860-117-1, Copyright 1992, Chapters 1, 2.1,2.5, 3.1, 3.4, 3.6

[PARA] David Gelernter, Alexandru Nicolau and David Padua *Languages and Compilers for Parallel Computing* ISBN 0-262-57080-7, Copyright 1990, Chapters 1-8.

[STATIC] Michael I. Schwartzbach `http://www.brics.dk/~mis/static.pdf` Lecture notes on the Daimi Course Static Analysis. All but lampda-calculus.

B.2 Secondary sources

References

[BITREE] Sang-Kyu Lee, Member, IEEE, and Hyeong-Ah Choi, Member IEEE *IEEE TRANSACTIONS ON PARALLEL AND DISTRIBUTED SYSTEMS, VOL. 7, NO. 5, MAY 1996 Embedding of Complete Binary Trees into Meshes with RowColumn Routing*

[ACMO] Unknown `http://en.wikipedia.org/wiki/Actor_model` A wiki on Actor Model

[IMPR] Unknown `http://en.wikipedia.org/wiki/Imperative_programming` A wiki on emperative programming

[IMFU] Unknown `http://en.wikipedia.org/wiki/Functional_programming` A wiki on functional programming

[THCO] Unknown `http://en.wikipedia.org/wiki/Theory_of_computation` A wiki on the theory of computation

[CONC] Unknown http://en.wikipedia.org/wiki/Concurrency_%28computer_science%29 A wiki on concurrency

[MACO] Unknown http://en.wikipedia.org/wiki/Matthew_Cook A wiki on Matthew Cook

[GLID] Unknown http://www.alesdar.org/oldSite/IS/images/glider.gif Just a random google result

[CGLTM] Paul Rendell http://www.cs.ualberta.ca/~bulitko/F02/papers/tm_words.pdf

[VNBN] Unknown http://en.wikipedia.org/wiki/Von_Neumann_architecture The Von Neumann architecture, used in most computers today. 3 The Von Neumann Bottleneck describes the bottleneck of the architecture.

[AFC] Marc Cohen http://faculty.washington.edu/smcohen/320/4causes.htm

[GPUT] GPU-Tech http://www.gpucomputing.eu/download/en_presskit.pdf

[TSG] Richard Dawkins *The Selfish Gene* ISBN 0-19-929115-2, 30th anniversary edition, first published 1976.

[BROMILLE] Peter Bro Miltersen http://www.daimi.au.dk/~bromille/

[ION] Unknown http://en.wikipedia.org/wiki/Ion_engines A wiki on the Ion Thruster

[BIRO] Unknown http://en.wikipedia.org/wiki/Bipropellant_rocket A wiki on the Bipropellant rocket

[JIGLOO] JIGLOO] Cloudgarden http://www.cloudgarden.com/jigloo/ The homepage of JIGLOO.

[MFOUC] Michel Foucault http://en.wikiquote.org/wiki/Michel$_$Foucault