Democritus University of Thrace

Department of Electrical and Computer Engineering

Diploma Thesis:

System Dynamics Modeling with Clojure

Christos Pylianidis

Supervisors

Ioannis N. Athansiadis

Katsiri Eleftheria

ksanthi, July 2017

Abstract

Dynamic system simulation is used in an extensive set of fields ranging from environmental modeling to economics and vehicle crash simulations. The number of factors affecting the systems grows exponentially and the systems become increasingly complex. As a result, the dynamic simulators have to be fast and provide results in time in order to be useful. Several techniques and algorithms have been developed in order to reduce the simulation times, with a growing interest in the parallelization of the numerical integration happening during the simulation. As multi-core architectures emerged and platforms utilizing a huge number of processing units became more prevalent, this turn to parallelization has dramatically reduced [1][2][3] the simulation times, but still leaving room for improvement. The vast majority of the solutions provided require an architecture with decent processing capabilities and software which a lot of dynamic system modelers/users don't have access to. In this work, the programming language of choice is Clojure, which among others runs on the JVM and can target home computers and servers. Experiments are made to investigate if Clojure can facilitate the parallelization of a popular numerical integration procedure. Two types of parallelization are examined: parallelism in time and parallelism across the system. The results for parallelism in time are not encouraging but with parallelism across the system a decent speedup is achieved. Along the way, a new system partitioning algorithm is presented as well as a web application hosting the created simulator.

ACKNOWLEDGEMENTS

First of all, I would like to thank Professor Ioannis N. Athanasiadis for his guidance and support throughout this project, as well as for giving me the opportunity to collaborate with him. Also, I would like to thank Professor Eli Katsiri for her assistance and willingness to help me. Of course, I would like to thank my father for his support and patience.

Contents

List of figures

List of tables

Chapter 1

Introduction

This chapter presents the aim of this work and shortly introduces basic concepts of dynamic system simulation and its use.

1.1 Dynamic system simulation

Simulation is a broad term that applies to fields ranging from physics and biology to economics and social sciences. For the broad scientific sense, simulation is the capability of representing system behavior. It is a useful design technique because it can be used to study parameter change effects on system performance and examine system behavior without the actual system. Dynamic system simulation is based on a mathematical representation that describes physical system behavior and dynamics using differential equations [4]. These equations can be ordinary or partial. As mathematical models become more complex and incorporate real-world constraints, equations become non-linear and it is hard to find an analytical solution.

Thus, numerical methods are used to solve the equations. A numerical simulation is done by stepping through an interval and calculating the integral of the derivatives by approximating the area under the derivative curves. Some methods use a fixed step interval, and others use an adaptive step that can shrink or grow automatically to maintain an acceptable error tolerance. Some methods can use different time steps in different parts of the simulation model [5].

Applications of dynamic system simulation include the prediction of environmental factors like the concentration of CO_2 and waters levels in basins [6]. Its industrial uses range from nuclear power, econometric models and drug dose migration through the human body [5]. Also, it is heavily used in computer animation to simulate hair, cloth and liquids.

1.2 Importance of fast simulation

Some of the fields that dynamic system simulation has spread to are more demanding than others. For example, simulations in the automotive and aerospace industries have to be executed as fast as possible and with low error tolerance [4]. The requirements of real-time applications such as flight simulators can make the dynamic system simulation a difficult task. In addition, computer graphics software needs to be paired with physics engines to accelerate such calculations [5]. There are also multi-body systems that need hours and even days for the calculations to be performed [7]. These examples show the need for the simulation to happen fast and with acceptable error rates. For this work, some dynamic systems were created which are smaller in scale than the aforementioned ones and are not related to any scientific field.

1.3 How to speed up

One way to improve the performance of dynamic system simulation is to look for a potentially faster numerical procedure to approximate ordinary differential equation solutions. The forward Euler and Parker-Sochacki [8] methods consist only from additions, subtractions and multiplications making them convenient for high-speed computation. Furthermore, there are methods like Cash–Karp [9] and Dormand–Prince [10] that use adaptive stepping and are considered computationally efficient [11].

Another approach is to investigate if parallel computing can provide any performance gains. By dividing the computational load to more processing units, less time is needed to make the calculations. In this case where there is the need to approximate the solutions of ODEs, which is an initial value problem [12], parallelism can be classified into three main categories: parallelism across the method (or parallelism across time), parallelism across the system (or parallelism across space) and parallelism across the steps [13][14]. These will be explained in paragraph 2.3.

Here, it was decided to go over some parallel techniques in order to achieve the desired speedup, because as dynamic systems grow in size the need for load distribution becomes higher but can also scale well with the increasing number of cores in the processors.

1.4 The technology of implementation matters

Choosing the path of parallelism, in order to achieve the desired performance proper technology should be used. There are several options for the simulator, a distributed system, a cluster or a simple home computer. The final choice depends on the type of parallelism that is going to happen as well as the performance needs of the application. Another important choice that has to be made is the programming language which should be used. It should be fast and ideally have strong concurrency features to allow easier development of complex systems. Depending on the programming language that will be used, there are available some widely known APIs like OpenMP [15] and the Open MPI [16] that facilitate parallel computing and could save development time by providing features that are difficult to implement [17] [18].

In this work, the target hosts are multi-core home computers and servers. The programming language is Clojure and it is going to be used without depending on APIs or another platform for the parallelism part because it is interesting to experiment with it alone and check its capabilities.

1.5 Related work

A lot of research has been made in the past in order to parallelize the process of dynamic simulation and several algorithms have been created. One of the most known is Parareal [19] which uses parallelism in time through the integration procedure. It utilizes temporal discretization and calculates the numerical solution for multiple time steps in parallel. Another algorithm that uses parallelism in time is PITA [20]. PITA is able to solve linear second-order equations which Parareal cannot. Also, it is worth mentioning the PFASST [21] algorithm which uses parallelism in space and time, and relies on the iterative Spectral Deferred Correction (SDC) time-stepping scheme and incorporates Full Approximation Scheme corrections to the coarse propagator.

Shifting to parallelism across the system, research has been made to split hybrid systems into their subsystems using conditional dependencies [22]. In addition, it has been shown that waveform relaxation methods can be used to achieve parallelism across the system [23][24]. More recent findings include SWIFT [3]. SWIFT is an open source simulator for cosmological phenomena, which models the problem into tasks, then into a graph and finally uses graph-based domain decomposition techniques.

1.6 Purpose and structure of thesis

The purpose of this work was to create an equation based dynamic system simulator using Clojure. It was decided to emphasize to the parallelization part and investigate how Clojure can contribute to this aim as well as to explore alternative ways to achieve the desired speedup. The target computers would be home computers and servers, and the system would accept only ordinary differential equations. In addition, the integration procedure would be the Euler method. The research questions formulated were:

1. How can Clojure's features be used to achieve parallelism?
2. Are the created methods effective?
3. How can the best performance through parallelization be achieved, in an integration procedure independent way?

In order to answer the above questions, several parallel methods were created and have been compared to their serial counterparts using a benchmarking library. These methods and their results are going to be presented along with what led to their conception. In more detail, Chapter 2 gives an insight into the choice of Clojure, the integration procedure selection and the test systems that were used. Chapter 3 presents the serial methods that were compared to the parallel ones. Chapter 4 analyzes the variations of the parallel in time methods created. Chapter 5 introduces the parallel in system method created and presents a newly developed algorithm for graph partitioning. Chapter 6 contains the results of the benchmarks for all the methods created. In Chapter 7 a web application is created to host the dynamic system simulator. Finally, Chapter 8 discusses the things which could be changed or added to the existing dynamic system simulator in order to improve it.

Chapter 2

Materials and Methods

This chapter unravels the reasons for the choice of Clojure and the Euler method and presents the input systems which will feed the simulator.

2.1 Clojure

The most compelling reason for using Clojure was its concurrency features. Clojure has several types of references, each one serving a distinct purpose relevant to concurrency. These types can be combined and create functionality that would be too complex to achieve with other languages. Its immutable data structures guarantee that they can be safely shared among the threads without using locks and mutexes. Clojure also comes packed with a software transactional memory system which among others makes race conditions (conditions where the final result depends on which thread finished its task first) impossible to occur. In addition, it is fast as it compiles into JVM bytecode and its performance is on par with Java. This happens without sacrificing flexibility, as it is dynamic and the packed REPL [25] helps make iteration quick. Another plus is that since Clojure runs on the JVM, it can run on most platforms and servers without modifications. Last but not least, every Java class and library can be used from inside of Clojure.

2.2 Euler method

Deciding which integration procedure is going to be used, it had to be taken into account the tradeoff between performance and error tolerance. That’s because there are procedures that are easier to parallelize in time as they are computationally inefficient, but have lower deviation values from the real ones. That means that higher step sizes could be used, resulting in a lower amount of iterations. In the end, the Euler method seemed to be a nice candidate as it is widely used by dynamic system simulation software like Vensim [26] and STELLA [27], and it is the default selected integration procedure in them. Also, the Euler method is proposed from literature relevant to system dynamics modeling [28]. This choice didn’t have a big impact on the development of the parallel methods as they were created with modularity in mind, meaning that the integration procedure could be easily switched without affecting the rest of the simulation system.

2.3 Types of parallelism examined

As mentioned in chapter 1.3, parallelism suitable for this work can be classified into three main categories: parallelism in time, parallelism across the system, and parallelism across the steps.

Starting with the first, parallelism in time exploits the parallelism in numerical scheme that is used to solve equations [29]. Techniques used in parallelism in time include block methods, frontal methods, stabbing methods, Picard-like methods and extrapolation methods [30]. A frequently used approach encountered in parallelism in time, is to express the system with matrices, break them into chunks and sent each chunk to a different processing unit. The main problem in that method is the synchronization happening in every step. To reduce the parts that have to communicate and be synchronized in every step, the problem can be converted to a graph based one, and then use the multilevel technique [31][32] or find hypergraphs [33][34].

Parallelism across the system partitions the system equations into various components and distributes the computations over different processors. The main difference between parallelism in time and parallelism across the system is that the latter creates partitions which are independent from the rest. For the partitioning, a common approach is to find the subsystems that have no dependencies between them. This is possible if the problem is modeled to be task based and proceed with task decomposition [35]. Another route is to express the system in terms of a graph and find potential independent subsystems through graph partitioning techniques [36]. An alternative is to reshape the problem in order to create non autonomous subsystems and proceed with the integration using the concept of conditional dependency [37][22].

Finally, parallelism across the steps includes techniques which solve simultaneously over a large number of steps [14]. This type of parallelism allows a high degree of parallelism with the drawback of a heavy control of the convergence of the numerical solutions towards the exact one [13]. Parallelism across the steps is suitable for architectures that support massive parallelism and have a lot of processing units, which is the reason why it will not be examined in this work.

Table 2.1: Parallelism in time vs parallelism across the system vs parallelism across the steps

Parallelism in time	Parallelism across the system	Parallelism across the steps
• The involved processing units need synchronization • The degree of parallelism depends on the problem size • The degree of parallelism depends on the numerical procedure used • Not subsystem limited • Easier to balance the workload between the processing units	• The involved processing units can be fully asynchronous • The degree of parallelism depends on the problem size • The degree of parallelism depends on the existence and the number of subsystems found • Not numerical procedure limited • Load balancing can be hard • Scales well even in systems with a few processing units	• The involved processing units need synchronization • The degree of parallelism is independent of the problem size (large scale parallelism even for small problems) • The degree of parallelism depends on the numerical procedure used • Not subsystem limited • Easier to balance the workload between the processing units

• Benefits from architectures with a good amount of processing units		• Benefits from architectures with a good amount of processing units

In this work, to achieve parallelism in time, the equations were partitioned into groups arbitrarily and sent to different threads in order to be calculated in each step (there is also a case where they weren't partitioned). To achieve parallelism across the system, the systems were modeled as graphs and their independent subsystems were detected and then calculated in different threads.

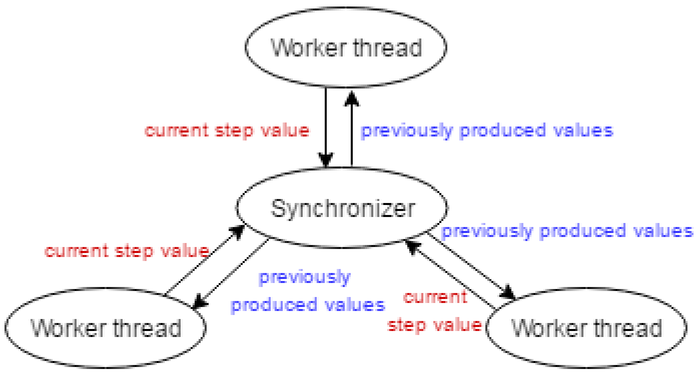


Figure 2.1: Overview of parallelism in time, the threads share their results in every step of the integration.

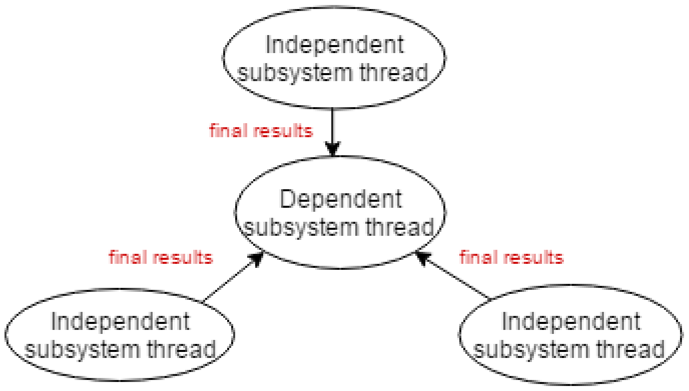

Independent subsystem thread
final results
Dependent subsystem thread
final results
final results
Independent subsystem thread
Independent subsystem thread

Figure 2.2: Overview of parallelism across the system, the threads share their results at the end of their individual integration.

2.4 Benchmark systems

To test the dynamic system simulator results for correctness some small systems with known output were used, although they won't be mentioned here. Instead, here will be shown only the test systems which were used for benchmarking. Before proceeding an important note has to be made. These systems (except for one) don't represent any real system neither have any physical meaning. They were created just to benchmark the developed methods. The reason is that for big systems were the equations were many it was difficult to gather them in one place and pass them to the simulator, as some depended on other unknown functions or had constants with unknown values. An implication of the unknown behavior of these systems is that their equations had to be made linear, otherwise when the iteration counts would grew the accuracy of the computer would be insufficient to depict the results.

Test system 1: This system was created in order to test the methods which used parallelism in time. It consists of 29 equations with arbitrary calculations. From now on it will be abbreviated as ts1.

Test system 2: This system was used to test the methods which used parallelism across the system. It consists of three predator-prey subsystems connected using three equations. So, there is a total of nine equations creating three independent teams of pairs and one dependent team with three equations. Its abbreviation will be ts2. If an equation is represented by a node and an incoming arrow shows dependency from the node that the arrow starts from, the graph of the system is the following.

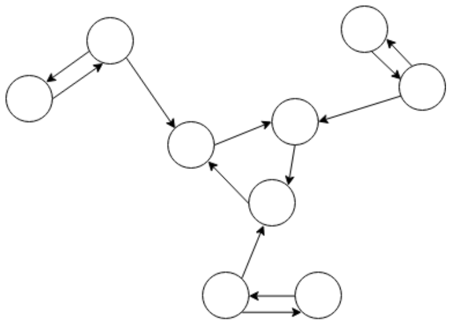


Figure 2.3: Test system 2 graph depiction.

Test system 3: This system consists of 120 equations, forming 10 independent teams of 10 equations each and one dependent team of 20 equations. It was created in order to test the increase of the speedup, in the case where a lot of independent subsystems are found. This will be the system ts3.

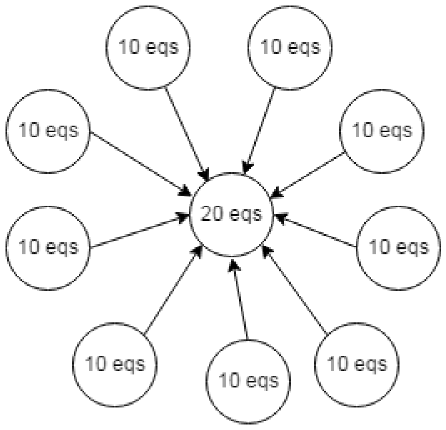


Figure 2.4: Test system 3 graph depiction.

The equations of the above systems can be found in the appendix. It will be noticed that ts1 has the equations in prefix form and inside Clojure functions. One the other hand, the rest of the systems have the equations in infix form and follow a specific notation. This is due to the fact that the mechanism to convert infix equations to prefix and create functions from them was added later on, after the methods which used parallelism in time were created.

2.5 Serial methods

In order to test the performance of the newly created parallel implementations, they had to be compared to their serial counterparts and measure the speedup. Two serial methods were created, one to compare with the methods using parallelism in time and one for the methods using parallelism across the system. These methods differ in the way they take inputs and handle data internally. More on the reason of this duality on the appendix G.

2.6 Parallelism in time

While examining how parallelism in time could be achieved by taking advantage of Clojure’s reference types several techniques were tested. Some of them were expected to have mediocre to bad performance but it was worth to look into it and provide a proper implementation for each situation. Also, experiments were made using different data structures as storage containers. Effort was put into finding a way to lessen the synchronization happening between the threads in every iteration of the integration procedure. In addition, a few minor optimizations were made in some parts of the code in order to have some performance gains.

2.7 Parallelism across the system

Having a bit more experience with Clojure’s reference types and their behavior when used for multithreading from developing the methods for parallelism in time, things were clearer when the time came to create the method for parallelism across the system. Also, a variation of this type of parallelism was tested here using communication through channels. The main challenge was to find a systematic way to partition systems into independent subsystems if they existed. Many techniques were examined like task decomposition, dependence analysis and modeling the systems as graphs and applying widely known algorithms to them. None matched the current case, and as a result a new algorithm was created which satisfied certain criteria (more on paragraphs 5.2-5.3).

Chapter 3

Serial Methods

This chapter presents with more detail the two serial methods created, along with some design choices that led to their specified type of input.

3.1 Serial method for comparison with parallelism in time

Initially, some assumptions were made in order to design this method. The input would be the integration limits, the step size and a variable number of ODEs, describing the dynamic system which would be simulated, and whose form would be unknown to the method. Also, a common way to represent the input equations was needed, taking into consideration that in order to use these equations, the inputs and their order should be known in advance. Therefore, the selected representation for each equation was the vector below

$$[initial\_value \ \ \mathrm{inputs} \ \ \mathrm{function}]$$

where $initial\_value$ was the initial value of the equation, inputs were the functions which provide the arguments this equation needs in the order it accepts them and $\mathrm{function}$ was the actual function calculating the equation. By the way, this is a nice example of the power of functional programming languages like Clojure because they allow easier manipulation of functions, as they handle them as first class citizens, meaning that they can be passed as arguments to other functions, returned by them or stored in a reference like common variables in other languages. Moving on to the serial method implementation, the values were stored in a vector of subvectors. Each subvector kept the values produced for one function. The subvectors were created with size 1, to hold the initial values and grow as time passes by. There was also a map showing from where a function should pull its values and a vector with the positions of the inputs of each function.

3.2 Serial method for comparison with parallelism across the system

The inputs of this method were the integration limits, the step size and a map. This map was holding all the information relevant to the functions. It had keys the names of the functions and as values maps which had keys the initial values, the expressions of the functions and the inputs. In contrast with the previous serial method, here there was no need to know the order of the inputs in which they were specified in the function declaration of the equation, as the library Infix took care of that. As a result the previous notation wasn't used here. The produced values were kept in a transient map where keys were the function names and values transient vectors for faster access and updates. After the integration the above data structures were made persistent before returning the results.

Chapter 4

Parallelism in Time

This chapter presents the methods developed using the parallelism in time approach. The time used to create these methods was also a time of experimentation and as a result it involved a lot of trial and error in order to find the appropriate Clojure constructs.

4.1 Method concept

In order to calculate the value of a differential function for one step the Euler method requires the value of the previous step. This phenomenon also appears in several other integration procedures and is the root of the synchronization happening between the threads when using parallelism. In the methods tested, there was always an entity acting as a synchronizer between the threads involved in the numerical integration. The synchronizer could be the main thread of execution or the data containers (the combination of data structures storing the produced values). Its job was to block the threads when the values they required weren't ready. Eventually, the purpose of these methods was to reduce this synchronization as much as possible and make the computations efficient.

4.2 Problem deconstruction

Proceeding to the experiments, the most important factors affecting the performance of the simulation were considered to be: the data structures storing the produced data, how many equations were sent to each thread, the reference types used and how threads were handled.

For the data structures acting as data containers the viable choices were vectors and maps due to their fast access/lookup times. Lists were left out because their element access complexity was linear. Another Clojure construct which could be used as data container was the promise. The special thing about promises was that they could provide a synchronization mechanism as they block the thread trying to dereference them until a value is ready.

Concerning the number of the equations sent to each thread, the choices were to send one equation to each thread or groups of equations. The latter seemed to be a better choice as the thread invocation time should not exceed the calculation time, which could happen if the workload of a thread was too little. Another aspect relative to the number of groups of equations was how many groups should exist. If the groups were less than the logical threads of the CPU it could mean that the processing power is not fully utilized. On the other hand, if the groups were more than the logical threads, frequent context switching would happen which would mean concurrency and not parallelism in the calculations. Ideally, the groups should be equal to the number of logical threads of the CPU.

The dominant factor affecting the performance was the reference types used (more details on reference types in appendix B). That's because the reference types control the way the concurrency happens as well as the scope of the involved references. The available options were atoms, vars, agents, refs or none of these. Starting with the refs, they are synchronized through transactions like the ones happening in databases. In the case of numerical integration these transactions would have to happen at every step and that would introduce significant overhead. Atoms provide a compare-and-set semantic and vars among others makes the reference globally accessible to the scope of the application, both behaviors being useful for this problem. Agents hold a value and queue actions in order to change it, this change happens in another thread in unknown time. They can be most useful when there are a lot of actions to be done. In the case of numerical integration, an action could be the calculation of an equation. So, a bunch of actions could be queued to a number of agents and they would calculate the results in each step. The problem is that agents return to the caller thread after every calculation. This is inefficient, and a better solution is to keep the calculations of specific equations in specific threads during the integration. Keeping calculations in specific threads would render agents needless, as it would mean only one action in the beginning of the integration, in order to send the equations to the threads and then wait for the integration to be completed. Regarding the reference types, another choice is to not use one. This has implications as the scope of the data containers and coordination of the threads can be a problem.

Proceeding to thread handling, this term includes the way threads are invoked and also if they are placed in a thread pool. The options were to use futures, the action mechanism of the agents, external libraries like the core.async, or Java Threads. Futures provide an easy way to execute code on another thread, and this thread is chosen from a thread pool. The action mechanism of the agents gives two options to enqueue actions, 'send' and 'send-off'. With 'send' a thread from a thread pool is selected. If there isn't an available thread the agent can wait until a thread is freed. With send-off, a new thread is created and after it completes its work is put in a thread pool. The latter two were discarded as it was decided that at this time only Clojure constructs would be used.

Below follows a table which presents the combinations of reference types and data structures used in each method.

Table 4.1: A comparison table showing the combination of data containers (first column) and reference types (first row) taking part in the experiments. The reference types store the corresponding data containers in the methods shown.

	Atom	Var	Agent	Ref
List	-	-	-	-
Vector	Method 1 Method 3 Method 3 variation 1 Method 4 Method 4 variation 2 Method 4 variation 3	Method 4 Method 4 variation 2 Method 4 variation 3	Method 2	-
Map	Method 1 Method 3 Method 4 variation 1	Method 4 variation 1	-	-

Promise	Method 3 Method 3 variation 1 Method 4 Method 4 variation 1 Method 4 variation 2	Method 4 Method 4 variation 1 Method 4 variation 2	Method 2	-

The following figure shows an overview of the simulator for parallelism in time.

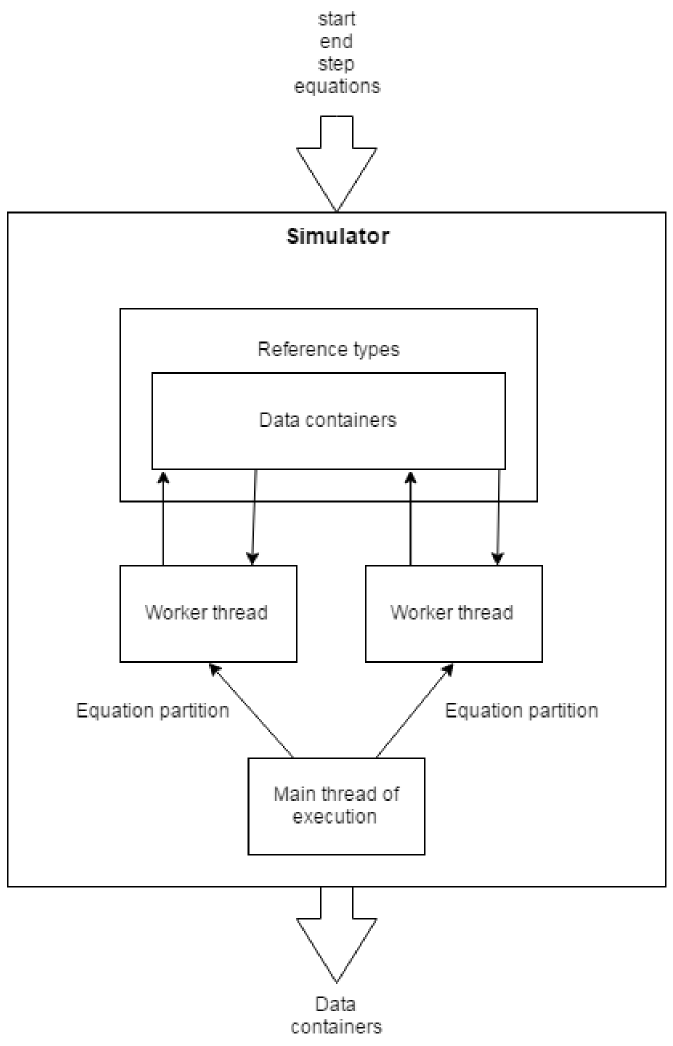


Figure 4.1: Parallelism in time, general view of the simulator. The inputs are the given start, end, step and the equations (including their initial values). The equations are partitioned and the main thread of execution assigns to each worker thread a partition. The worker threads store and read the produced results from the data containers. The reference types contain the data containers and provide different kinds of access to them. After the results are ready, the data containers are passed to the output.

Next, a table presents the distinct characteristics of each method, based on the overview of the simulator, involving the data structures used as data containers, how the equations were partitioned before being sent to the threads for calculation, the reference types which contained the data containers and how threads were created.

Table 4.2: A comparison of the methods using parallelism in time. In the column 'Data containers' are the data structures used to store the final results of the integration, in 'Equation partitioning' it is explained the portion of the input handled by each thread, 'Reference types' contains reference types like agents, vars, atoms and refs used to achieve specific behaviors and finally 'Thread handling' is the way a calculation is sent to another thread or a thread starts.

	Data containers	Equation partitioning	Reference types	Thread handling	Comments
Method 1	map, vectors	The equations are partitioned into chunks using the formula $equations\_num / cores\_num$ and are processed in different threads **in every iteration**	atom	futures	• **In every iteration** chunks are sent to different threads. • In every iteration all the threads wait for the rest to complete their calculations, and then the results are saved in a map which has the equation names as keys, and vectors to save the produced values as values
Method 2	vectors, promises	Each equation is assigned to an agent and each agent dispatches an action in every step	agents	send-off	• Each agent holds a vector where values are saved • agents_num=equations_num • **In every iteration** all agents dispatch actions to threads in a thread pool. Threads complete the actions and save the results to the corresponding agents • Vectors have fixed size=iterations+1 • Promises create the block until value is ready behavior
Method 3	map, vectors, promises	The equations are partitioned into chunks using the formula $equations\_num / cores\_num$ and are processed in different threads	atom	futures	• A map which will hold all the information is created

					• It has the equation names as keys, and vectors to save the produced values as values • The map is an atom • Vectors have fixed size=iterations+1 • Promises create the block until value is ready behavior
Method 3 variation 1	vectors, promises	The equations are partitioned into chunks using the formula $equations\_num / cores\_num$ and are processed in different threads	atoms	futures	• **Same as method 3 but with vectors instead of map** • Each equation saves its values in a vector • Each vector is an atom • equations_num=atom_num=vector_num • Vectors have fixed size=iterations+1 • Promises create the block until value is ready behavior
Method 4	vectors, promises	The equations are partitioned into chunks using the formula $equations\_num / cores\_num$ and are processed in different threads	vars, atoms	futures	• Each equation saves its values in a vector kept in an atom • equations_num=atom_num=vector_num. • Each atom is kept to var • Vectors have fixed size=iterations+1 • Promises create the block until value is ready behavior
Method 4 variation 1	maps, promises	The equations are partitioned into chunks using the formula $equations\_num / cores\_num$ and are processed in different threads	vars, atoms	futures	• Each equation saves its values in a map. • Each map is an atom

					• So, equations_num=atom_num=map_num • Maps have iterations as keys and the produced values as values • Promises create the block until value is ready behavior
Method 4 variation 2	vectors, promises	The equations are partitioned into chunks using the formula $equations\_num/cores\_num$ and are processed in different threads	vars, atoms	futures	• Each equation saves its values in a vector kept in an atom • equations_num=atom_num=vector_num • Each atom is kept to var • **In contrast to method 4 vectors grow as time passes by** • Promises create the block until value is ready behavior
Method 4 variation 3	vectors	The equations are partitioned into chunks using the formula $equations\_num/cores\_num$ and are processed in different threads	vars, atoms	futures	• Each equation saves its values in a vector kept in an atom • So, equations_num=atom_num=vector_num. • Each atom is kept to a var • Vectors have fixed size=iterations+1 • **Same as method 4 but promises were thought to be a delay so they weren't used. Instead, infinite loops took their place until a value is ready**

4.3 The initial failed attempt for solving linear ODEs

In the beginning, the idea to create methods that work only for linear ODEs in the format

$$y = ax + b$$

came up and two implementations were made. The special thing about having only linear equations like the above is that the A and B matrices can be created who contain the a and b parameters of all the functions respectively. Furthermore, this means that the input format mentioned in paragraph 3.1 can be simplified to this

$$[A \text{ B initial\_values start end step}]$$

where A, B, initial_values are vectors corresponding to the A and B matrices and the initial values. This eliminates the need to know the order of the inputs as well as from which function they are calculated. Two implementations were made. Both of them used refs to hold the new values produced in every iteration. Additionally, they distributed the appropriate workload to threads in every iteration. This means that in every iteration a thread had to calculate some values and then return them to the main thread. Their difference lies in the number of threads they deploy in order to make calculations. The first one starts a new thread for every given function. The second distributes the elements of the input matrices to the available cores in order to be more efficient.

The reason for choosing refs was that by the time these methods were written only the final values of the functions were desired. This led to a need for coordinated change in the functions' values. Otherwise, if in a specific step the value of a function changed, the other functions would calculate their results based on the new value of this function which is wrong because they should do it for the value of the previous step. Refs provide this coordinated change through Clojure's STM (software transactional memory). So, the new values had to be calculated in different threads and then be set as the new values of refs.

Apart from the use of refs in order to achieve the previously described way of parallelism, an attempt was made to use another Clojure construct which parallelizes some operations automatically, the reducers [38]. Reducers allow certain kind of operations to be performed in parallel without the intervention of the programmer. They require a reducible collection (a collection that knows how to reduce itself) and a reducing function (what needs to be done during the reduction). In particular, reducers partition the collections, apply a reducing function and then combine each partition using Java's Fork/Join framework. Here, a reducer could be used in the place where the inputs are summed and parameters of a function in order to produce a new value. Addition is an associative operation so regardless of the order of the calculations the final result would be the same.

Although the above were interesting ideas, a programming mistake was made, resulting in the serial execution of the methods, which needed to change a lot in order to work as expected. However, the time was limited and more sophisticated methods, which would also work for non-linear ODEs had to be implemented, so the methods for linear ODEs were left aside.

4.4 Method 1

Even though refs provided the desired coordinated update of values, they also introduced some unwanted overhead as doing transactions in every integration step had a performance toll when the steps were increased.

For this reason a method was made using Clojure’s futures. The idea was to send the input functions, to calculate their new values based on the Euler method, in separate threads at each step and then wait for all the futures to complete in order to proceed with the next step. For this purpose a hash-map was created, and was put inside an atom to provide the essential coordination, with functions as keys and the values produced as values. When all futures had finished, their values replaced the old ones in the hash-map. To make things a bit more efficient the input functions were partitioned into chunks based on the number of logical cores of the system. The partition happened in such a way that the workload was equally distributed to each thread. The reason for doing this was that threads stayed idle from the point of completing a task to taking on another. The time wasted could be reduced if the number of tasks could be reduced. By grouping them the calculation of many functions became one task and theoretically time was saved.

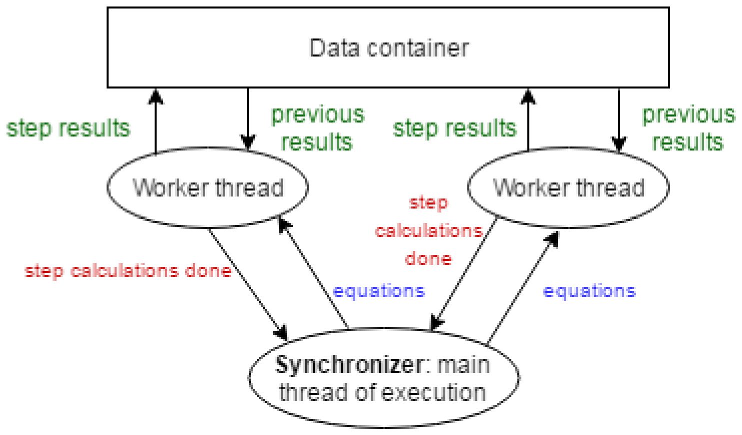


Figure 4.2: Parallelism in time, operation logic of method 1.

4.5 Method 2

Another experiment was made using one of Clojure’s reference types, the agent. In this implementation, the agents also acted as storage containers for the values produced in previous steps. So, each input function was assigned to a different agent and in every iteration the input functions were sent as actions to the corresponding agents in order to update their values, using the Euler method.

The choice of using the agents as storage containers occurred due to a misconception of that time, considering that it was better to not have all the previously calculated values in one place (such as a map or vector). To explain, such a storage place would have to be searched every time in order to find the position where a function’s values are stored. A hash-map with functions as keys and the produced values as values would do the trick but even that would cause some searching overhead, which becomes bigger as the equations grow. Furthermore, the common storage place would have to be updated constantly as all the functions would have their values kept there and this would slow down the whole integration procedure as only one thread could 'write' a value at a time. Eventually, with the current implementation a vector is created which shows to every function where to look for its inputs in the given input functions vector. For example, given this input vector of functions

$$[[1 \text{ h g f }][0 \text{ g g}][2 \text{ h f h}]]$$

the following vector will be created

$$[[2\ 1][1][2\ 0]]$$

which means that the first function ($[1 \text{ h g f }]$, remember function notation from paragraph 3.1) takes its first input from the function in position 2 ($[2 \text{ h f h}]$) in the input functions vector, and the second input from the function in position 1 ($[0 \text{ g g}]$). By doing it that way, the time to find and get the function inputs is constant because the agents are also contained in a vector which is ordered as the input functions vector. That means that the function 'f' in the previous example will take its first input from the second agent in the agent vector, and the second input from the first agent.

Another aspect which had to be taken care of was that the lack of synchronization between the agents caused situations where a thread proceeded to the calculation of a function, but some of its inputs weren't calculated yet. So, this thread had to block until the values were ready, or dispose the action and somehow enqueue it to the front of the corresponding agent queue in order to let the thread free for someone else. The second option didn't seem viable because it would add extra overhead for the enqueueing and action reordering. Moreover, it didn't provide any guarantee that the thread would be used by someone else who needed it more because the same agent could dispatch the same disposed action repeatedly and none else would take turns. As a result, the first option was selected and the threads blocked until all the inputs were ready.

Here, 'send-off' was used for testing purposes in order to assign each input function to a different agent. In case 'send' was used, there would be times where all the threads in the fixed thread pool were occupied and blocked, meaning that the other threads waiting to take turns would be led to starvation. And as the occupied threads would be waiting for values that would be produced from threads that couldn't take turns the integration would halt.

An obstacle deriving from the uncoordinated behavior of agents is how to handle the absence of some values that a function needs. In other words what should be done if some of the function inputs haven't been calculated by the time it needs to calculate a new value. The way to handle this situation depends on the contents and size of the storage containers the agents have. Suppose these containers are vectors. If their size depends on the amount of values produced so far, a function may need a value which hasn't been produced yet. So, it will look for a value at an index that doesn't exist because the vector is smaller, throwing an exception like that. A solution is to make the vector size same as the number of steps and fill it with a value that will never occur during the integration. Then, if a value is needed, check if the value at the index where it is supposed to be is different than the predefined one. If not, perform this check until it's ready. This works but adds boilerplate code. A more elegant way, which also reduces the latency of continuous checks, is to use Clojure's promises. The previous vector is filled with promises and when a value is not ready the thread that made the request will block until the value is ready.

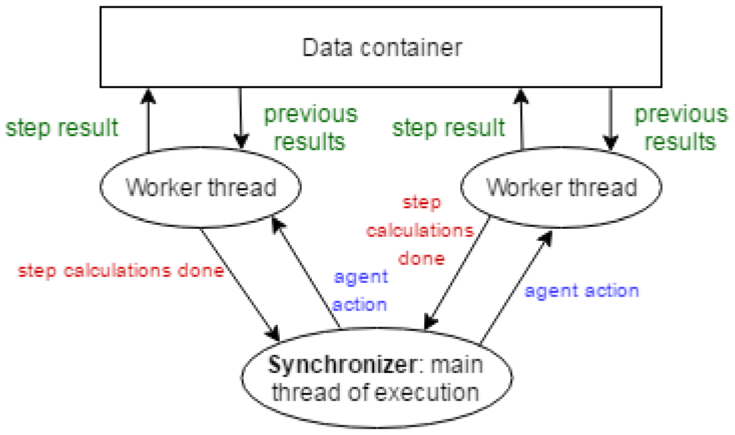


Figure 4.3: Parallelism in time, operation logic of method 2.

4.6 Method 3

Sending functions to be calculated in different threads at every step caused major delays. Both methods, the one using futures and the other with the agents were wasting a lot of time keeping the available threads idle between the tasks. It was time to create and benchmark a method which would have the calculations stay in specific threads. It was decided that there was no point in using agents because calculations would stay local in each thread and there wouldn't be any actions to send to them. In addition, refs didn't make any sense as they inherently brought some noticeable transaction overhead and the same coordination behavior could be achieved in other ways. What was needed was to send the input functions to a number of threads and then wait until all of them had finished their calculations. Futures seemed to be a viable choice for this.

Another topic seeking attention was how to handle the storage of the produced values. Multiple threads should have access to the same resources and should also be able to modify them. Atoms are ideal for situations like this. They provide access to shared resources and support synchronous, uncoordinated operations. In order to change an atom's value, a function is applied to the current value and it is attempted to compare-and-set in the new value. Since another thread may have changed the atom's value in the intervening time, the above procedure may have to be retried, and does so in a spin loop. The atom would hold a hash-map constructed in the same way as the previous methods and filled with promises. The next decision that had to be taken was about the number of threads that should be deployed. The options were to have one thread for every function or to group the functions depending on the number of the logical cores of the system. The second option was clearly more efficient and also a better distribution algorithm was made than the existing one used to distribute functions in the previous methods. To sum up, the functions were distributed to the available threads, each thread calculated the functions that were assigned to it and when a value wasn't available yet it blocked.

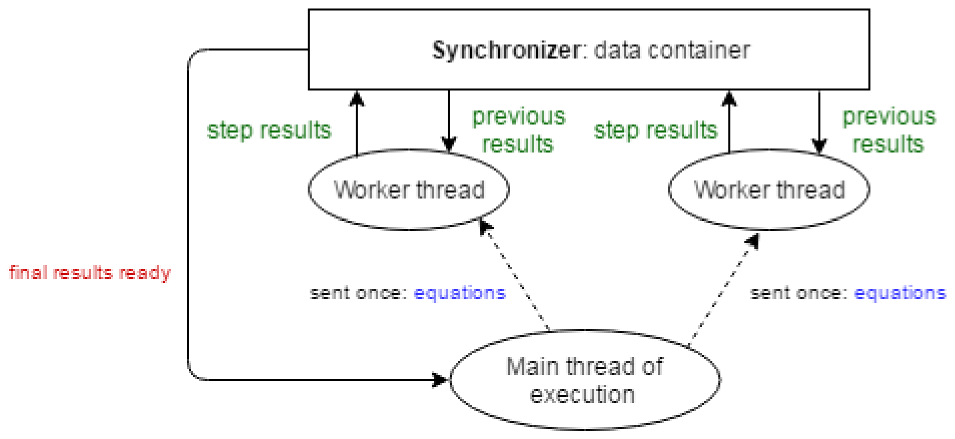


Figure 4.4: Parallelism in time, operation logic of methods 3.

4.7 Method 3 variation 1

An interesting experiment was to check how vectors performed against maps. This method differs from the previous one only on the data structure holding the produced results which became a vector.

4.8 Method 4

Except for method 2, the methods so far stored the produced results of all the threads in the same data structure. That's something undesirable because the same place means one atom to control access to the data structure which means retries. An ideal solution would be to have an atom holding a vector for each function. This would be the only function changing the vector and as a result no retries would be made. The interesting part was to make the other threads know from where to pull the needed values for the calculations as they should search the same place. To achieve this, Clojure's vars were used. The important part here is the global access which vars provides, as knowing the 'name' of the var is enough to take its value. The idea was to create a var holding an atom for each function. In order to find the appropriate var to save or read a value, they were named after the function's 'name' whose values they were holding, adding a preceding underscore. Although the aforementioned trick is poor form for writing Clojure code as it pollutes the namespace with vars and apart from that, conflicts may occur if another var has the same name with one of them, it was used as the only solution for the time being. To sum up there isn't any data structure holding all the values produced but instead each function can find the necessary values according to its name and the names of its arguments. The rest were left same as in method 3.

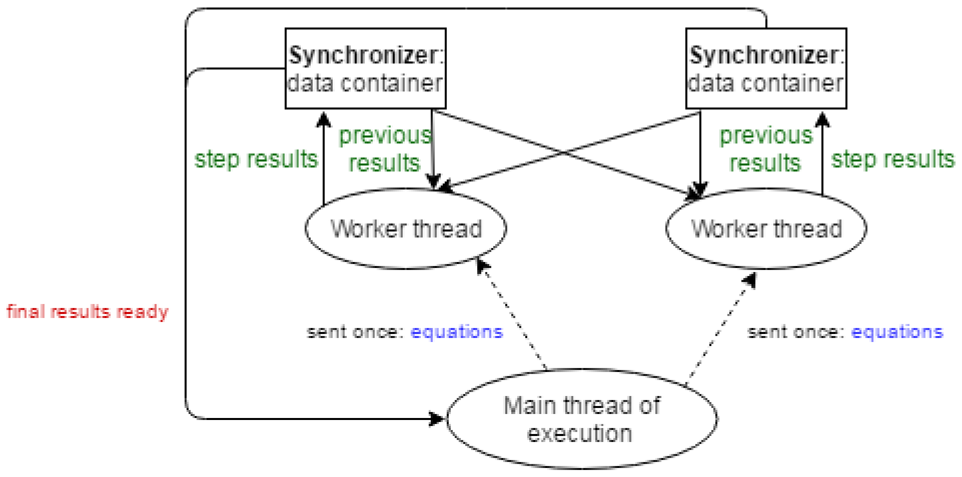


Figure 4.5: Parallelism in time, operation logic of method 4.

4.9 Method 4 variation 1

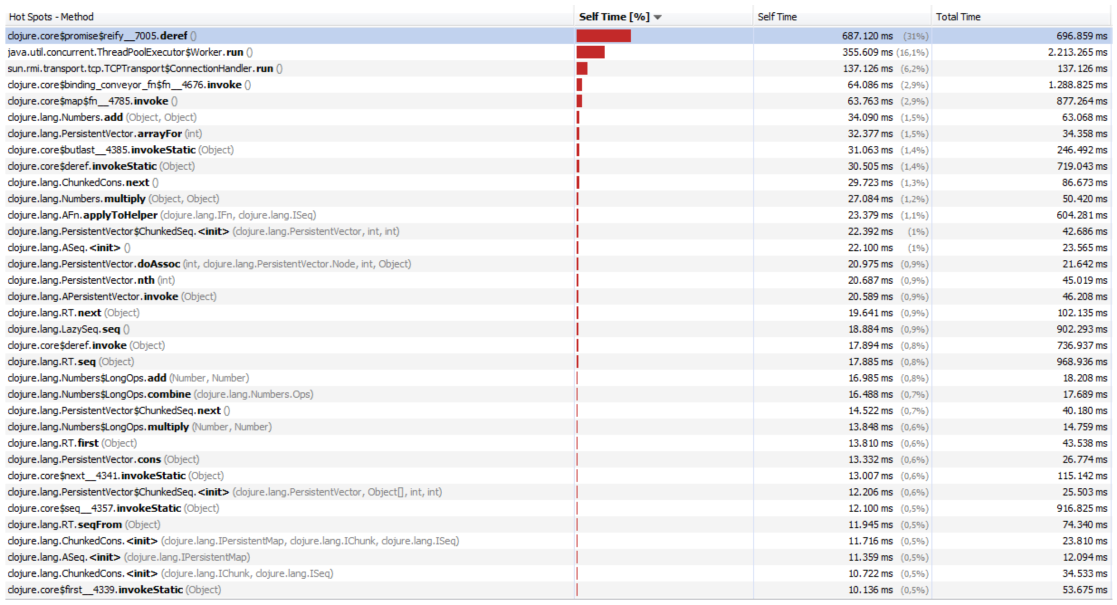
The previous method was altered by substituting vectors with maps in order to see how much the performance would be affected this time. Instead of having vectors with as many elements as the iterations plus one, there would be maps with fixed size of key-value pairs same as the vectors. Each map would be initialized to have the iteration numbers as keys and promises as values and then values would be delivered to the appropriate pairs.

4.10 Method 4 variation 2

While thinking of ways to improve the performance of method 4, a suspicion arose that having fixed sized vectors that had to be recreated every time a value was delivered to them was a slowdown factor. So, another experiment based on the previous variation was made but now the size of the vectors would increase as values were produced instead of being fixed at $iteration\_num+1$. In every iteration a check was made to ensure that after the conjoin of the new element the vector size would be at maximum $iteration\_num+1$. Also, at every given time vectors had two more elements ahead of value production of their function that they were associated to. This was because another thread whose function takes less time to be calculated, might need a value which hadn't been produced yet and it shouldn't search for an index that doesn't exist.

4.11 Method 4 variation 3

By profiling the previous methods it was found that major delays were caused by the frequent thread context switching and heavy use of the deref function when asking for values of atoms, vars and promises.



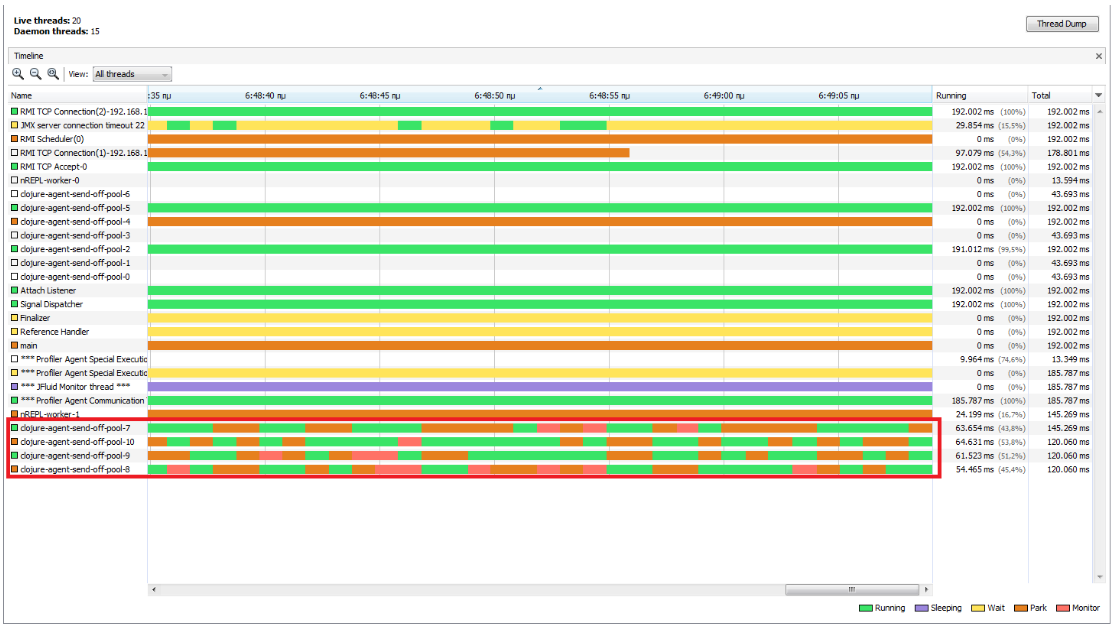
Hot Spots - Method	Self Time [%]	Self Time	Total Time
clojure.core$promise$reify__7005.**deref** ()		687.120 ms (31%)	696.859 ms
java.util.concurrent.ThreadPoolExecutor$Worker.**run** ()		355.609 ms (16,1%)	2.213.265 ms
sun.rmi.transport.tcp.TCPTransport$ConnectionHandler.**run** ()		137.126 ms (6,2%)	137.126 ms
clojure.core$binding_conveyor_fn$fn__4676.**invoke** ()		64.086 ms (2,9%)	1.288.825 ms
clojure.core$map$fn__4785.**invoke** ()		63.763 ms (2,9%)	877.264 ms
clojure.lang.Numbers.**add** (Object, Object)		34.090 ms (1,5%)	63.068 ms
clojure.lang.PersistentVector.**arrayFor** (int)		32.377 ms (1,5%)	34.358 ms
clojure.core$butlast__4385.**invokeStatic** (Object)		31.063 ms (1,4%)	246.492 ms
clojure.core$deref.**invokeStatic** (Object)		30.505 ms (1,4%)	719.043 ms
clojure.lang.ChunkedCons.**next** ()		29.723 ms (1,3%)	86.673 ms
clojure.lang.Numbers.**multiply** (Object, Object)		27.084 ms (1,2%)	50.420 ms
clojure.lang.AFn.**applyToHelper** (clojure.lang.IFn, clojure.lang.ISeq)		23.379 ms (1,1%)	604.281 ms
clojure.lang.PersistentVector$ChunkedSeq.**<init>** (clojure.lang.PersistentVector, int, int)		22.392 ms (1%)	42.686 ms
clojure.lang.ASeq.**<init>** ()		22.100 ms (1%)	23.565 ms
clojure.lang.PersistentVector.**doAssoc** (int, clojure.lang.PersistentVector.Node, int, Object)		20.975 ms (0,9%)	21.642 ms
clojure.lang.PersistentVector.**nth** (int)		20.687 ms (0,9%)	45.019 ms
clojure.lang.APersistentVector.**invoke** (Object)		20.589 ms (0,9%)	46.208 ms
clojure.lang.RT.**next** (Object)		19.641 ms (0,9%)	102.135 ms
clojure.lang.LazySeq.**seq** ()		18.884 ms (0,9%)	902.293 ms
clojure.core$deref.**invoke** (Object)		17.894 ms (0,8%)	736.937 ms
clojure.lang.RT.**seq** (Object)		17.885 ms (0,8%)	968.936 ms
clojure.lang.Numbers$LongOps.**add** (Number, Number)		16.985 ms (0,8%)	18.208 ms
clojure.lang.Numbers$LongOps.**combine** (clojure.lang.Numbers.Ops)		16.488 ms (0,7%)	17.689 ms
clojure.lang.PersistentVector$ChunkedSeq.**next** ()		14.522 ms (0,7%)	40.180 ms
clojure.lang.Numbers$LongOps.**multiply** (Number, Number)		13.848 ms (0,6%)	14.759 ms
clojure.lang.RT.**first** (Object)		13.810 ms (0,6%)	43.538 ms
clojure.lang.PersistentVector.**cons** (Object)		13.332 ms (0,6%)	26.774 ms
clojure.core$next__4341.**invokeStatic** (Object)		13.007 ms (0,6%)	115.142 ms
clojure.lang.PersistentVector$ChunkedSeq.**<init>** (clojure.lang.PersistentVector, Object[], int, int)		12.206 ms (0,6%)	25.503 ms
clojure.core$seq__4357.**invokeStatic** (Object)		12.100 ms (0,5%)	916.825 ms
clojure.lang.RT.**seqFrom** (Object)		11.945 ms (0,5%)	74.340 ms
clojure.lang.ChunkedCons.**<init>** (clojure.lang.IPersistentMap, clojure.lang.IChunk, clojure.lang.ISeq)		11.716 ms (0,5%)	23.810 ms
clojure.lang.ASeq.**<init>** (clojure.lang.IPersistentMap)		11.359 ms (0,5%)	12.094 ms
clojure.lang.ChunkedCons.**<init>** (clojure.lang.IChunk, clojure.lang.ISeq)		10.722 ms (0,5%)	34.533 ms
clojure.core$first__4339.**invokeStatic** (Object)		10.136 ms (0,5%)	53.675 ms

Figure 4.6: A screenshot from the profiler of Java VisualVm, running the method 4 variation 2. Self time is the time spent inside the body of a method without the time of inner method invocations. Total time is the time spent inside a method during the profiling session. As it seems, the dereferences take a respectable amount of time and the deref method is the most time-consuming from all.

Referring to the previous methods, the dereferences that had to be made because of promises in every iteration were as many as the inputs of the function plus one to acquire the previous value of the function. This was costly and an experiment was made based on the second variation without the use of promises. Promises were used as blocking mechanism when values weren't ready. The same behavior can be achieved by using loops, inside of which the availability of a value is checked. If the value at the index of a vector is the one that it was initialized with, the loop is repeated. Else, the necessary value is available and calculations can continue.

4.12 Discussion

The parallel in time methods performed worse than the serial one. All of them were more or less working in a synchronized manner. Waiting for all the futures to return at every step and blocking threads until other values were ready are different sides of the same coin, synchronization. By profiling the previous methods it was proved that blocking the threads when waiting for a value was a major cause of delay. This can be seen in the following image



Live threads: 20
Daemon threads: 15

Name	Running		Total
RMI TCP Connection(2)-192.168.1	192.002 ms	(100%)	192.002 ms
JMX server connection timeout 22	29.854 ms	(15,5%)	192.002 ms
RMI Scheduler(0)	0 ms	(0%)	192.002 ms
RMI TCP Connection(1)-192.168.1	97.079 ms	(54,3%)	178.801 ms
RMI TCP Accept-0	192.002 ms	(100%)	192.002 ms
nREPL-worker-0	0 ms	(0%)	13.594 ms
clojure-agent-send-off-pool-6	0 ms	(0%)	43.693 ms
clojure-agent-send-off-pool-5	192.002 ms	(100%)	192.002 ms
clojure-agent-send-off-pool-4	0 ms	(0%)	192.002 ms
clojure-agent-send-off-pool-3	0 ms	(0%)	43.693 ms
clojure-agent-send-off-pool-2	191.012 ms	(99,5%)	192.002 ms
clojure-agent-send-off-pool-1	0 ms	(0%)	43.693 ms
clojure-agent-send-off-pool-0	0 ms	(0%)	43.693 ms
Attach Listener	192.002 ms	(100%)	192.002 ms
Signal Dispatcher	192.002 ms	(100%)	192.002 ms
Finalizer	0 ms	(0%)	192.002 ms
Reference Handler	0 ms	(0%)	192.002 ms
main	0 ms	(0%)	192.002 ms
*** Profiler Agent Special Executic	9.964 ms	(74,6%)	13.349 ms
*** Profiler Agent Special Executic	0 ms	(0%)	185.787 ms
*** JFluid Monitor thread ***	0 ms	(0%)	185.787 ms
*** Profiler Agent Communication	185.787 ms	(100%)	185.787 ms
nREPL-worker-1	24.199 ms	(16,7%)	145.269 ms
clojure-agent-send-off-pool-7	63.654 ms	(43,8%)	145.269 ms
clojure-agent-send-off-pool-10	64.631 ms	(53,8%)	120.060 ms
clojure-agent-send-off-pool-9	61.523 ms	(51,2%)	120.060 ms
clojure-agent-send-off-pool-8	54.465 ms	(45,4%)	120.060 ms

Figure 4.7: A screenshot from the profiler of Java VisualVm, running the method 4 variation2. Inside the red circle are the threads running the method. The colored bars depict states of threads during time, which passes from left to right. The green parts mean actual running time, the orange parts mean that the thread is parked (here the parking is done when a thread waits for another value from another thread because of the promise functionality), and the red part shows periods in which the profiling happens. Ideally, the bars would be green and maybe have some red parts.

The orange parts mean that the threads are idle, waiting for tasks. This happens because of the synchronization when waiting for other values to be produced. So it became obvious that finding a way to group functions which could run independently from others (parallelism across the system) was essential to get a performance boost.

Chapter 5

Parallelism across the System

In this chapter a new algorithm for system partitioning is presented as well as the created methods. Also, it is explained why some algorithms didn't fit in this specific case of finding independent teams.

5.1 Method concept

The goal here was to find a way to partition the input systems into independent components and proceed with the numerical integration individually in each one of them. It was decided to use the serial integration in each of these threads as it was the fastest candidate as seen from the previous tests. These threads would send their results to a thread responsible for the integration of the dependent team which would wait for them to finish in order to start, or would work along with them.

5.2 Searching for a system partitioning algorithm

To parallelize the integration, the first thought was to try and find a way to group the input functions depending on their parameters, and make teams which would be able to complete the whole integration procedure without depending on the rest functions. Each independent team would be put in a separate thread and the rest of the functions which couldn't form teams would also run apart, blocking their threads until all their parameters were ready. Topological sorting [39] seemed to fit the problem in the beginning. It is an algorithm used primarily for task scheduling and sorts the tasks based on their importance of completion.

To use algorithms like topological sorting, the systems had to be modeled as graphs. A node represented a function, and the incoming edges denoted that this function depended on the functions (nodes) where these edges started.

An obstacle was that many directed cycles were created and topological sorting worked for directed acyclic graphs. On second thought, the dependencies were in time, meaning that the edges showed dependence from the previous step. So, in reality there weren't any cycles and the depiction of the graph had to change. For every node, another one was added representing the function's value in the previous step. For example the graph depicting the following system (with the notation presented in paragraph 3.1)

$$[[0\ \mathrm{D}\ A][0\ A\ \mathrm{C}\ \mathrm{D}\ B][0\ B\ C][0\ A\ \mathrm{C}\ D]]$$

is this

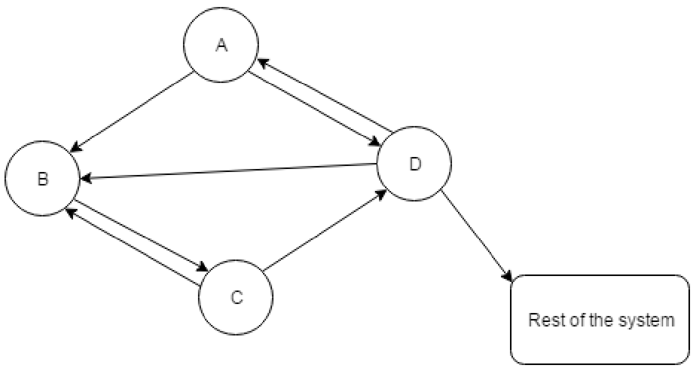


Figure 5.1: Example of a graph with a cycle, depicting a subsystem.

and as it can be seen cycles exist. The revised graph is

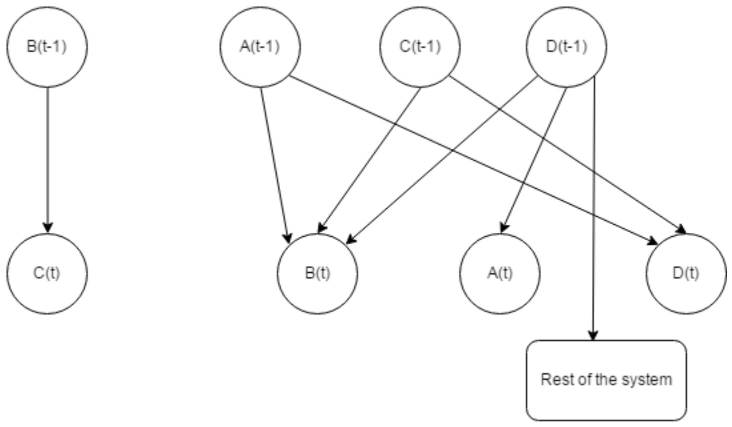


Figure 5.2: A bigraph of the system in figure 5.1, if dependencies through time are taken into account.

The second depiction made clear that there would always be two levels of nodes. Applying topological sorting to this graph would always make two teams of tasks, one for each level. It was realized that topological sorting didn't fit the current problem because at every step all functions have the same priority. Topological sorting orders stuff based on their priority in time, not clearly on their dependencies. In the Euler method there is no time priority because when referring to grouping the talk is about the same step.

Many other algorithms and techniques were examined in order to find a way to create independent groups of functions and attempts were made to find similarities with problems whose solution is already known. A problem that appeared to be similar was the 'Closure problem' [40]. In graph theory and combinatorial optimization, a closure of a directed graph is a set of vertices with no outgoing edges. There are algorithms to find the maximum and minimum closures. The important thing here is that in a closure there are no outgoing edges. Presumably, the area left out of the closure will have only outgoing edges. So, the dependencies of the nodes outside the closure are also nodes outside the closure. This means that they form a group which can run asynchronously. For example, in the picture below the closure consists of the nodes 6,7,4,8. The rest of them 1, 2, 3, 5 can form a group.

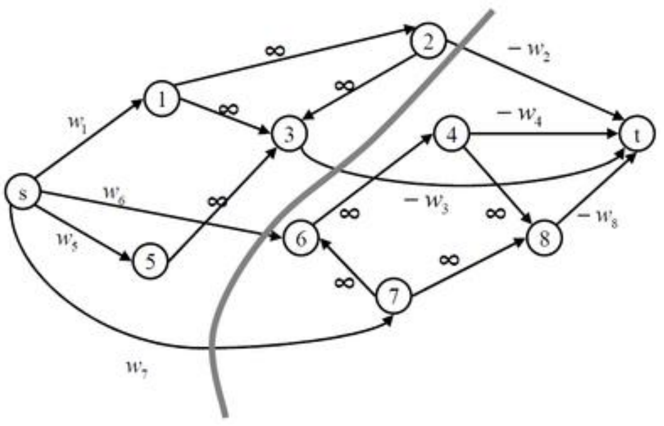


Figure 5.3: Reduction from closure to maximum flow (taken from 'https://upload.wikimedia.org/wikipedia/en/thumb/d/dd/Closure.png/420px-Closure.png').

The problem is that the existing algorithms find the maximum or minimum closure. There is no middle ground solution or a way to dynamically adjust how many nodes should be grouped. For example, using the maximum closure two or more independent teams might be grouped together and put on the same thread, while putting them in different threads would be preferable. On the other hand, using the minimum closure the opportunity to put extra nodes in a small team would be wasted. The picture below shows the latter example, a minimum closure will contain the nodes 3, 4, 5 and the independent group will consist of the nodes 1 and 2.

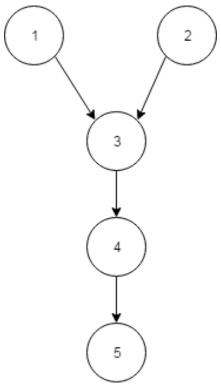


Figure 5.4: Example of a system showing the possibility of adding nodes to independent teams recursively. An incoming edge means that the node depends from the node on the other side of the edge. The nodes (1, 2) can form teams, and then (3,4,5) can be added to it if needed.

The opportunity to create a larger group containing the node 3 or even node 4 is lost. This team expansion could play an important role if the number of cores was limited because fewer threads could run simultaneously and fewer teams should be created. So, it is important to be able to control this parameter.

At some point the research led to the strongly connected components [41]. A graph is said to be strongly connected if every vertex is reachable from every other vertex. It is possible that in a graph there are subgraphs which are strongly connected. These subgraphs form independent teams if they don't have any ingoing edges, but most of the time that's not the case. In the following image the strongly connected components are highlighted. The upper left one consisting of the nodes a, b and e forms an independent team because there are only outgoing edges, which means that the dependencies of this group are found only inside the group.

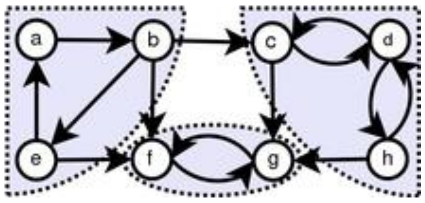


Figure 5.5: Graph with strongly connected components marked (taken from 'https://upload.wikimedia.org/wikipedia/commons/thumb/5/5c/Scc.png/220px-Scc.png').

Additionally, every strongly connected component with no ingoing edges is a team but a team that is not strongly connected will not be detected by the algorithm responsible for finding these components. As a result finding the strongly connected components was rejected.

Several other mathematical concepts and techniques used in different fields were examined. Among them was the dependence analysis [42] used in compiler theory. With this method, execution-order constraints are created in order to determine whether or not it is safe to reorder or parallelize statements. There are two classes of dependencies, control dependencies and data dependencies. The dependencies of the functions seemed to fit the second type and more specifically a subcategory of data dependencies called flow dependencies. Flow dependencies are created when a statement needs a value, which another statement changes, in order to proceed. Respectively, functions need parameters produced by other functions in order to proceed. Although similarities exist there wasn't found a practical way to apply this kind of analysis so it was abandoned.

Other terms that might be related to this problem, but couldn't find a way to link them to it, were found in different fields like combinatorics and graph theory. Some of them were the task decomposition [35] and the Eulerian matroids [43]. Task decomposition refers to the procedure of finding the individual tasks composing a bigger system and it is used in order to discover which of these subtasks can be run in parallel. The Eulerian matroids are structures whose elements can be partitioned into collections of disjoint circuits.

5.3 The partitioning algorithm developed

An interesting find of having a directed graph where each node represents a function and an ingoing edge denotes a dependency from the starting node of the edge, is that independent teams only have outgoing edges. Assuming that movement between nodes is allowed depending on the direction of the edges, the previous observation means that moving from one node to another, there would be some nodes that are unreachable. That is the base of the algorithm.

For start, the reachability of all the nodes is determined. That’s trivial using a graph exploration algorithm like depth first search. The next step is to compare the reachabilities. Nodes with the same reachability are put on the same teams. Now, some of these teams are independent and others not. To identify which belong to each category, a check is applied. If the dependencies of all the nodes of a team are nodes of that team, then the team is independent. Otherwise it’s not.

To further increase the independent team detection capabilities of the algorithm an enhancement was made. So, for each neighbor of each node of each independent team, check if the dependent team it belongs to can be added to the independent team. This can happen if the dependent team has no other outer dependencies than the independent team. If it can’t, check only for this node. If something was added to the team start checking from the beginning the neighbor nodes.

Further information about the evolution of the algorithm and what led to the above enhancement can be found in the appendix C.

5.4 Problem deconstruction

Proceeding to the experiments, it was realized that the problem should be approached from a different perspective than in parallelism in time as a lot of parameters changed opening new possibilities. The reference types offered by Clojure were no longer needed as there was no need for thread coordination or access to the same data structures. Also, equation partitioning was not a matter of concern (like in parallelism in time) as the equations would be partitioned into independent teams using the algorithm described in paragraph 5.3 and each thread would handle one or more of them. In addition, the fact that threads could run independently opened the possibility for thread communication between the independent threads and the dependent one.

Having said the above, the most important factors affecting the performance of the simulation were: the data structures storing the produced data, the thread communication and thread handling.

For the data structures acting as data containers the viable choices were again vectors and maps due to their fast access/lookup times. Lists were left out because their element access complexity was linear. This time promises were not an option as there was no need for synchronization between the threads.

For the thread communication the options were to create a mechanism for interprocess communication or use an external library. A custom made solution would be too time consuming to develop with uncertain results and limited reliability. As a result, the library core.async and its channels were used.

Revising thread handling, this term includes the way threads are invoked and also if they are placed in a thread pool. The options were to use futures, the action mechanism of the agents, external libraries like the core.async, or Java Threads. Unlike parallelism in time, it was decided to use the core.async library because it would be imported anyway for the thread communication and it would be interesting to check how its ‘go’ threads performed. These threads are cheap to create, have the ability to park and also are part of a provided thread pool by the library.

The table below refers to the flow of computations and specifically to the order of the independent and dependent integrations taking place.

Table 5.1: A comparison of the flow of computations between the created methods.

	First the threads calculating independent teams finish their task and then the dependent team calculation begins	The threads calculating independent teams and the one calculating the dependent team run simultaneously	The threads calculating teams, further chunk the teams and send them to other threads for calculation
First the threads calculating independent teams finish their task and then the dependent team calculation begins	Method 1	-	-
The threads calculating independent teams and the one calculating the dependent team run simultaneously	-	Method 2 Method 2 variation 1	_
The threads calculating teams, further chunk the teams and send them to other threads for calculation	Method 3 Method 3 variation 1	_	Method 3 Method 3 variation 1

The following figure presents an overview of the simulator for parallelism across the system. The difference with the overview of the simulator in parallelism in time is the absence of the reference types.

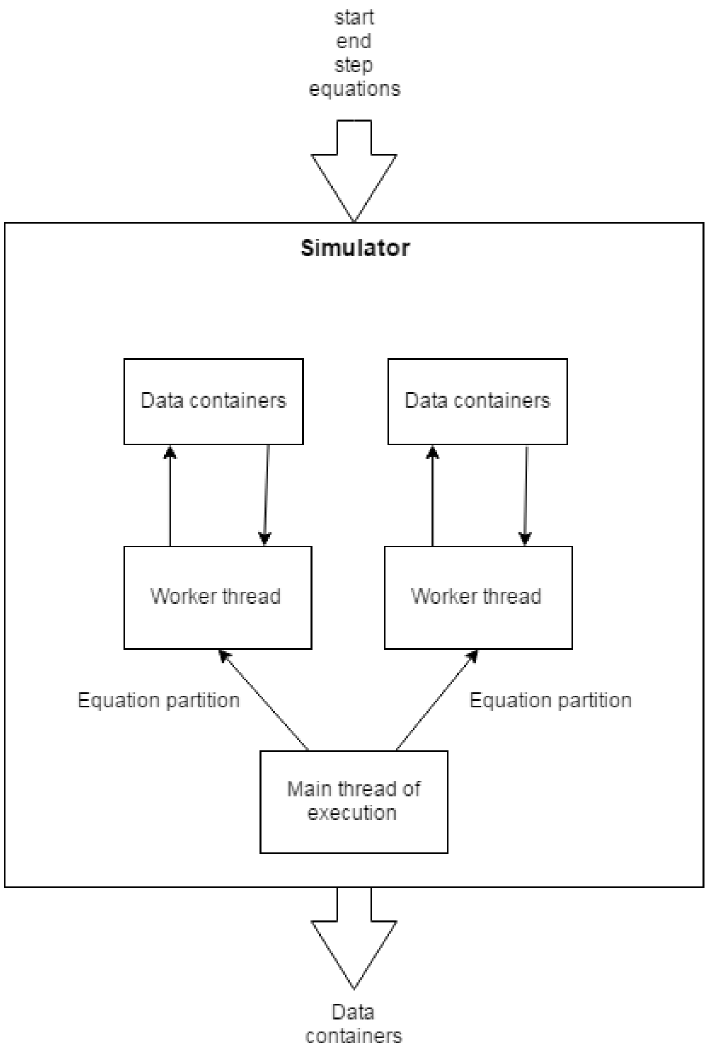


Figure 5.6: Parallelism across the system, overview of the simulator.

The table on the next page briefly presents the methods created for parallelism across the system and their distinct characteristics.

Table 5.2: A comparison of the methods using parallelism across the system. In the column 'Data containers' are the data structures used to store the results of the integration, 'Thread communication' contains the way the threads communicate and finally 'Thread handling' is the way a calculation is sent to another thread or a thread starts.

	Data containers	Thread communication	Thread handling	Comments
Method 1	map, vectors	No communication	core.async/go	• First the threads calculating independent teams finish their task and then the dependent team calculation begins
Method 2	map, vectors	core.async/chan	core.async/go	• Channels • One message in every iteration • Each message contains the results of one iteration
Method 2 variation 1	map, vectors	core.async/chan	core.async/go	• Channels • One message **every fixed number** of iterations • Each message contains all the values produced by an independent team or a group of them
Method 3	map, vectors	No communication	core.async/go	• First the threads calculating independent teams finish their task and then the dependent team calculation begins

				• The functions are sent to other threads in each iteration in order to be calculated
Method 3 variation 1	map, vectors	No communication	core.async/go	• First the threads calculating independent teams finish their task and then the dependent team calculation begins • **Only the functions from the dependent team** are sent to other threads in each iteration in order to be calculated

5.5 Method 1

In this method, the function grouping was done according to the algorithm and the system-map (a map having all the information needed about the functions for the integration) was broken into several other maps named subsystem-maps. The next step was to wait for all the threads calculating independent teams to finish their task and pass the final results to the thread calculating the dependent teams. It's worth repeating here that the serial method was used in each thread to do the integration.

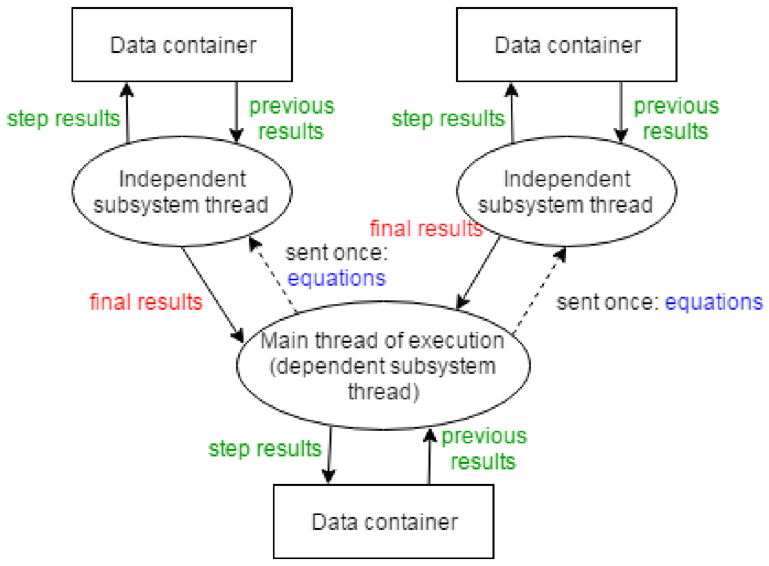
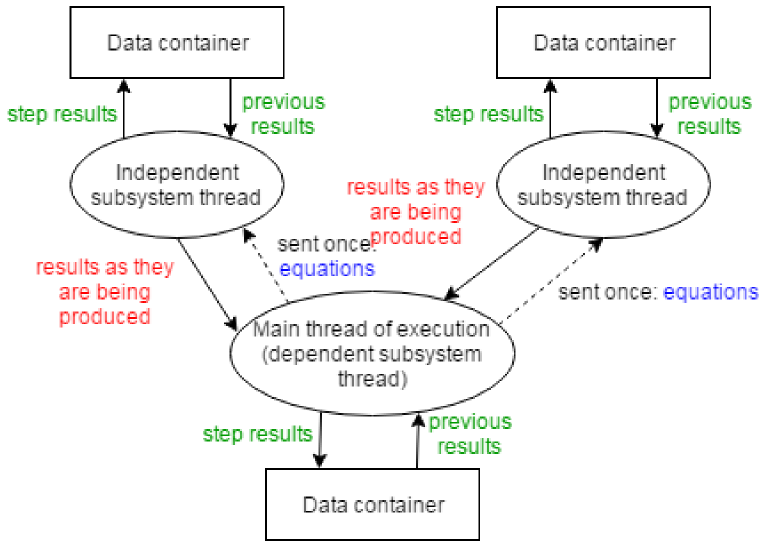


Figure 5.7: Parallelism across the system, operation logic of method 1.

5.6 Method 2

Here, after the function grouping was done, it was decided to send the independent results to the thread calculating the dependent results right after they were ready. It was thought that by doing it in that way a lot of time would be saved because the dependent thread wouldn't have to wait for all others to finish and stay idle for a long period of time. The communication between the threads was done using channels from the core.async library of Clojure. The channels were made buffered and each one had size equal to the total amount of iterations. If they weren't buffered the senders would block until the recipient takes their message. Also, the choice for that specific buffer size was made due to the situation of many senders who do calculations fast and send fast their messages, and a receiver who process them slowly. So, the channels should have enough size to accommodate all the messages for each one of the senders. These channels were then merged, and the new channel delivered its contents to the thread responsible for the dependent integration.



Data container
Data container
step results
previous results
step results
previous results
Independent subsystem thread
results as they are being produced
Independent subsystem thread
sent once: equations
results as they are being produced
sent once: equations
Main thread of execution (dependent subsystem thread)
step results
previous results
Data container

Figure 5.8: Parallelism across the system, operation logic of method 2.

5.7 Method 2 variation 1

Next, doubts arose about the robustness of the previous implementation and also about the sent frequency. As a result, the method was changed and the sent frequency was made adjustable by the programmer, who has to decide the number of sends each thread will do during the integration. It could also be made dynamically adjustable depending on the iteration total but that wasn't investigated further. Except for the number of sends, code changes were made in the function that orchestrates the data merging in the dependent integration. These changes were done with bringing down the processing burden of the incoming messages to a minimum in mind. In the previous implementation, the messages were maps containing only the newly produced values. The values were extracted one by one and put in the right place of the map holding the produced values of the dependent integration. With the changes, the messages are the whole map of the produced values and so it can be merged right away with the map of the dependent integration.

5.8 Method 3

Another thing tested in order to make calculations even more distributed, was to use multiple threads per independent subsystem to calculate the functions in each integration step. That wasn't something new as it was tested in parallelism in time, but what changed since then was the way the functions were called, which was much more expensive than before (due to the import of Infix) and so gains were possible. The initial guess was that a speedup could only be achieved if there were enough logical threads; otherwise the context switching between the threads would overwhelm any possible gains. Also, the number of the equations sent should be high enough to overcome the overhead of the thread creation. These may be lightweight go threads but there is still a creation cost. Having said the above, method 1 was changed as necessary to do the aforementioned stuff and thus this method was created.

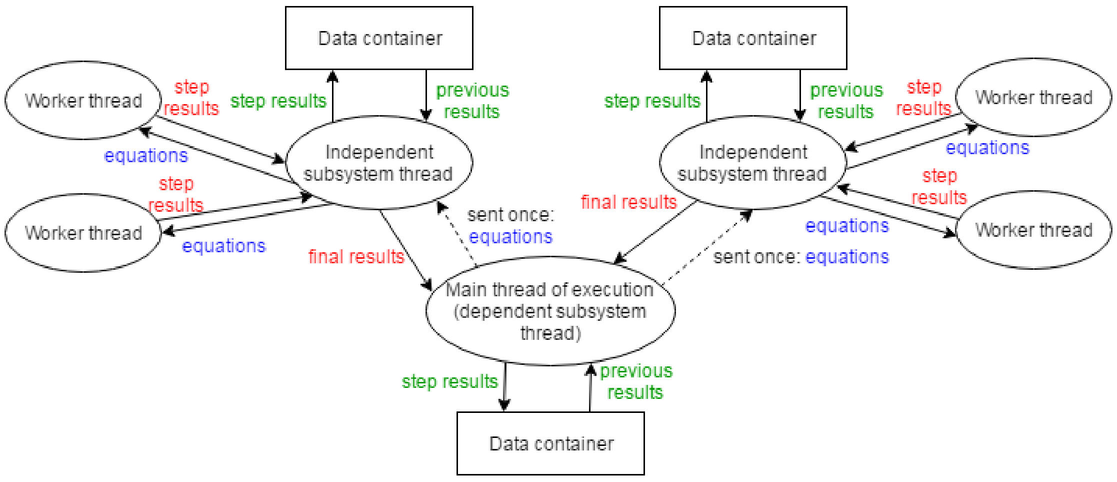


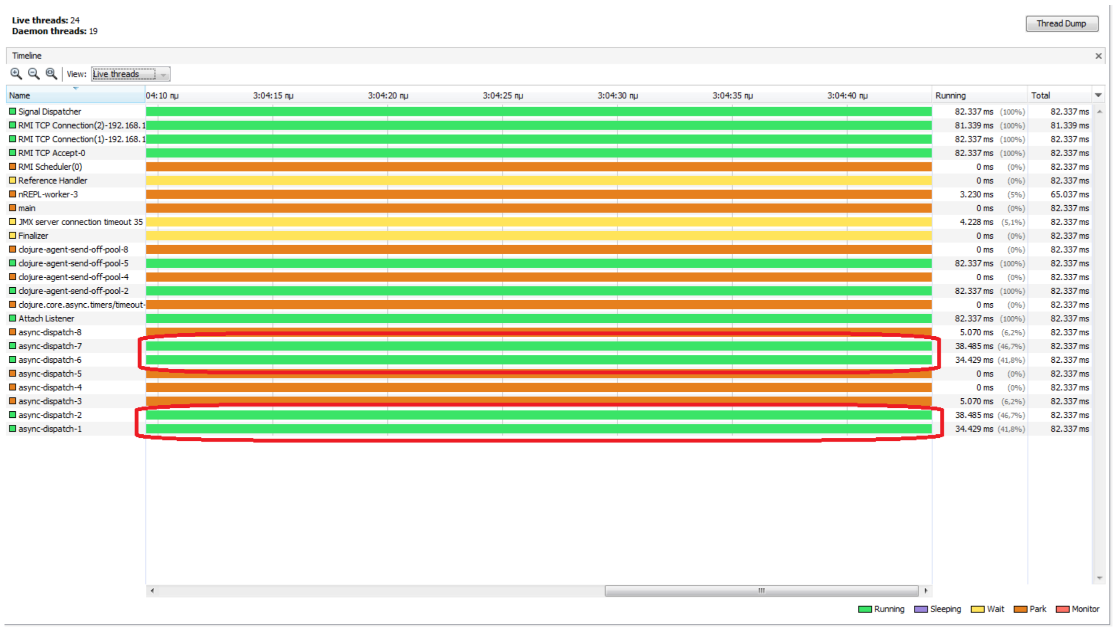
Figure 5.9: Parallelism across the system, operation logic of method 3.

5.9 Method 3 variation 1

This is a variation of method 3. Their difference is that here only the dependent functions were sent in each iteration in other threads in order to be calculated. This was done due to curiosity in order to check if it would be more efficient than the basic method.

5.10 Discussion

At this point all the parallel methods in this chapter were providing satisfying results. With parallelism across the system the inherited synchronization due to the Euler method was eliminated as no thread was waiting for another. This can be seen in the picture below.



Live threads: 24
Daemon threads: 19

Name	Running		Total
🟩 Signal Dispatcher	82.337 ms	(100%)	82.337 ms
🟩 RMI TCP Connection(2)-192.168.1	81.339 ms	(100%)	81.339 ms
🟩 RMI TCP Connection(1)-192.168.1	82.337 ms	(100%)	82.337 ms
🟩 RMI TCP Accept-0	82.337 ms	(100%)	82.337 ms
🟧 RMI Scheduler(0)	0 ms	(0%)	82.337 ms
🟨 Reference Handler	0 ms	(0%)	82.337 ms
🟧 nREPL-worker-3	3.230 ms	(5%)	65.037 ms
🟧 main	0 ms	(0%)	82.337 ms
🟨 JMX server connection timeout 35	4.228 ms	(5,1%)	82.337 ms
🟨 Finalizer	0 ms	(0%)	82.337 ms
🟧 clojure-agent-send-off-pool-8	0 ms	(0%)	82.337 ms
🟩 clojure-agent-send-off-pool-5	82.337 ms	(100%)	82.337 ms
🟧 clojure-agent-send-off-pool-4	0 ms	(0%)	82.337 ms
🟩 clojure-agent-send-off-pool-2	82.337 ms	(100%)	82.337 ms
🟧 clojure.core.async.timers/timeout-	0 ms	(0%)	82.337 ms
🟩 Attach Listener	82.337 ms	(100%)	82.337 ms
🟧 async-dispatch-8	5.070 ms	(6,2%)	82.337 ms
🟩 async-dispatch-7	38.485 ms	(46,7%)	82.337 ms
🟩 async-dispatch-6	34.429 ms	(41,8%)	82.337 ms
🟧 async-dispatch-5	0 ms	(0%)	82.337 ms
🟧 async-dispatch-4	0 ms	(0%)	82.337 ms
🟧 async-dispatch-3	5.070 ms	(6,2%)	82.337 ms
🟩 async-dispatch-2	38.485 ms	(46,7%)	82.337 ms
🟩 async-dispatch-1	34.429 ms	(41,8%)	82.337 ms

Figure 5.10: screenshot from the profiler of Java VisualVm, running the method 1. Inside the red circle are the threads running the method. The colored bars depict states of threads during time, which passes from left to right. The green parts mean actual running time, the orange parts mean that the thread is parked (here the parking is done when a thread waits for another value from another thread because of the promise

functionality), and the red part shows periods in which the profiling happens. Here, there aren’t any orange parts, which is natural as the threads don’t have to wait for values from other threads and can finish their work independently from others.

In contrast to the same picture of the paragraph 4.12, here no orange parts appear. The meaning of this is that all threads are busy until they finish their task, so there is no need to park. To clarify a bit more, 8 async-dispatch threads can be seen numbered from 1 to 8. This is due to a thread pool that the library core.async creates with a default size of 8. Here, only four of them were used, for efficiency reasons, as the testing machine had only four cores.

Chapter 6

Results

This chapter presents the results of the serial methods, the methods using parallelism in time and the ones using parallelism across the system. A short explanation is given after the tables of the parallel methods to justify their performance.

6.1 Benchmarking tool

While benchmarking an application running on the JVM someone should have in mind the effects of the JIT compiler and the garbage collector. Trying to measure computation time using the 'time' function provided by Clojure will yield different results depending on how well the JIT optimized the code and the state of the garbage collector. To avoid such pitfalls the performance of the created serial and parallel methods was measured with the Criterium [44] library. Criterium provides statistical processing of multiple evaluations of an expression, a warm-up period to allow JIT to optimize its code and purging of the garbage collector to isolate timings from garbage collector state prior to testing. The times mentioned later are the execution time means of the evaluations.

6.2 Results for serial methods

Because the methods were created under very different circumstances they will be examined individually. Something else to notice is the different iteration counts in which the methods were tested to. This is because the overhead in the methods using parallelism across the system (see appendix G) made them slow and for bigger systems it would take too much time to complete. However, this doesn't affect anything as it was emphasized earlier that the performance of methods using a different kind of parallelism should not be compared due to the introduced overhead of the methods using parallelism across the system.

6.2.1 Results for serial method for parallelism in time

This method was tested with the system ts1.

Table 6.1: Results for the serial method for parallelism in time (ts1)

iterations	1.000	10.000	100.000
execution time	33,51 ms	265,09 ms	3.722,50 ms

6.2.2 Results for serial method for parallelism across the system

This method was tested with the systems ts2 and ts3. For the first system the results were

Table 6.2: Results for the serial method for parallelism across the system (ts2)

iterations	100	1.000	10.000
execution time	3,17 ms	31,66 ms	302,86 ms

and for the second

Table 6.3: Results for the serial method for parallelism across the system (ts3)

iterations	100	1.000	10.000
execution time	114,01 ms	1.193,63 ms	11.114,80 ms

6.3 Results for parallelism in time

For the results of the methods using parallelism in time a summary table is provided and then they are examined individually. This table also contains the corresponding serial method to show the speedup/slowdown compared to it. After that, the results a depicted in a line chart. All the execution times for parallelism in time refer to the test system ts1. The CPU of the target machine had 4 logical threads.

Table 6.4: Cumulative results of the methods using parallelism in time. The serial method results are also included.

Iterations	1.000	10.000	100.000
Serial method	33,51 ms	265,09 ms	3.722,50 ms
Method 3 variation 1	36,69 ms	447,65 ms	5.866,25 ms
Method 4 variation 2	42,91 ms	479,36 ms	7.263,83 ms
Method 4	43,19 ms	502,27 ms	7.786,40 ms
Method 4 variation 1	50,40 ms	635,76 ms	9.449,93 ms
Method 3	69,89 ms	854,43 ms	9.595,12 ms
Method 2	90,61 ms	1.049,95 ms	14.805,64 ms
Method 1	103,04 ms	1.051,85 ms	11.417,77 ms
Method 4 variation 3	8.190,61 ms	-	-

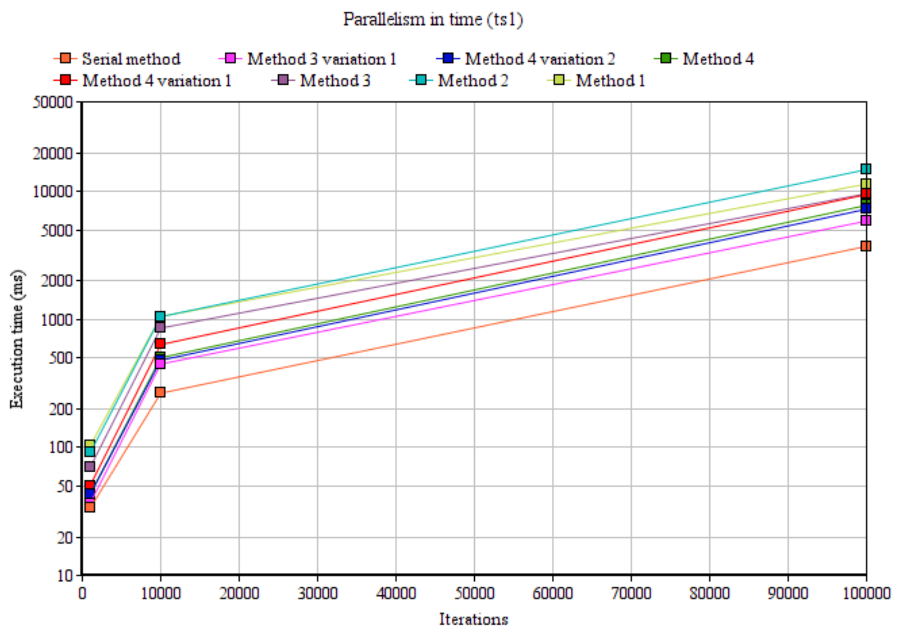


Figure 6.1: A chart with the results of the methods using parallelism in time for ts1

6.3.1 Results for Method 1

Table 6.5: Parallelism in time, results for method 1

iterations	1.000	10.000	100.000
execution time	103,04 ms	1.051,85 ms	11.417,77 ms

There are two main reasons responsible for this performance. The most important of them is that tasks are sent to threads at every step which is eventually a serious time waste. The other reason is the manual synchronization happening at the end of each step waiting for futures to complete.

6.3.2 Results for Method 2

Table 6.6: Parallelism in time, results for method 2

iterations	1.000	10.000	100.000
execution time	90,61 ms	1.049,95 ms	14.805,64 ms

To begin with, making one calculation at a time in a thread and return the result is not efficient because the time between the end of a calculation and the next action dispatched to the thread is idle time. Next is the one to one ratio of functions and agents which leads to frequent context switching and slows down the overall procedure. Additionally, the use of 'send-off' implies the creation of a new thread, which is a costly procedure. The higher the number of functions the higher this cost will be. To add up, the use of promises makes things kind of slower because the use of 'deliver' in order to deliver a value to a promise is slower than an immediate change of a value. Some of the above slowdowns could be overcome if the functions were grouped into chunks depending on the number of logical cores of the system. No extra threads would be created because the agent fixed size thread pool would be used and the context switching would be less. Furthermore, more functions would be calculated in the same thread so the idle time mentioned before would be lower.

6.3.3 Results for Method 3

Table 6.7: Parallelism in time, results for method 3

iterations	1.000	10.000	100.000
execution time	69,89 ms	854,43 ms	9.595,12 ms

The expectations for the performance of this method were high and it was thought it would be on par with the serial one. Benchmarks showed that it was indeed the faster parallel method tested so far. However, the serial one was still faster and the performance gap was big. It is believed that the contention in the atom holding the hash-map is mainly responsible for this bad performance. Most likely all the threads try to deliver new values to the hash-map and the atom makes a lot of retries. Of course, the usual suspects, the heavy use of promises and the number of hash-map searches make the situation worse.

6.3.4 Results for Method 3 variation 1

Table 6.8: Parallelism in time, results for method 3 variation 1

iterations	1.000	10.000	100.000
execution time	36,69 ms	447,65 ms	5.866,25 ms

Eventually, after these tweaks the performance was on par with the serial method for lower iteration counts, but stayed more and more behind as the iterations increased.

6.3.5 Results for Method 4

Table 6.9: Parallelism in time, results for method 4

iterations	1.000	10.000	100.000
execution time	43,19 ms	502,27 ms	7.786,40 ms

Benchmarks showed that holding all the values in a data structure kept by an atom caused contention, which in turn led to many retries in order to set each value.

6.3.6 Results for Method 4 variation 1

Table 6.10: Parallelism in time, results for method 4 variation 1

iterations	1.000	10.000	100.000
execution time	50,40 ms	635,76 ms	9.449,93 ms

The substitution of vectors with maps didn't have any positive effect. The lookup performance of maps is great but it still isn't a constant time operation like in vectors.

6.3.7 Results for Method 4 variation 2

Table 6.11: Parallelism in time, results for method 4 variation 2

iterations	1.000	10.000	100.000
execution time	42,91 ms	479,36 ms	7.263,83 ms

The changes to the previous method gave a slight boost to performance but nothing special. As it seems the recreation of a vector happens very fast due to structural sharing and it's not a source of delay.

6.3.8 Results for Method 4 variation 3

Table 6.12: Parallelism in time, results for method 4 variation 3

iterations	1.000	10.000	100.000
execution time	8.190,61 ms	-	-

The performance of this method not only didn't improve compared to the previous one, but it was the worst recorded. The performance gap of this method and the one using promises is that promises are made to work in an asynchronous way, not consuming core cycles and keeping the threads busy like this method does. The execution time of the 10.000 and 100.000 iterations is undefined because the method was left running for hours and still didn't finish.

6.4 Results for parallelism across the system

Again, for the results of the methods using parallelism across the system two summary tables are provided and then they are examined individually. This table also contains the corresponding serial method to show the speedup/slowdown compared to it. The first table contains the results of the test system ts2 and the second of ts3. In addition, the results of each test system are plotted in charts. The CPU of the target machine had four logical threads. For the test system ts2, each thread handled one subsystem. For the test system ts3, two threads handled 2 subsystems each and the other two three subsystems each.

Table 6.13: Cumulative results of the methods using parallelism across the system for the system ts2. The results of the serial method are also included.

iterations	100	1.000	10.000
Serial method	3,17 ms	31,66 ms	302,86 ms
Method 1	2,29 ms	21,38 ms	203,77 ms
Method 2	2,65 ms	24,93 ms	250,91 ms
Method 2 variation 1	4,51 ms	42,27 ms	420,59 ms
Method 3 variation 1	4,86 ms	47,64 ms	490,79 ms
Method 3	8,13 ms	79,84 ms	841,71 ms

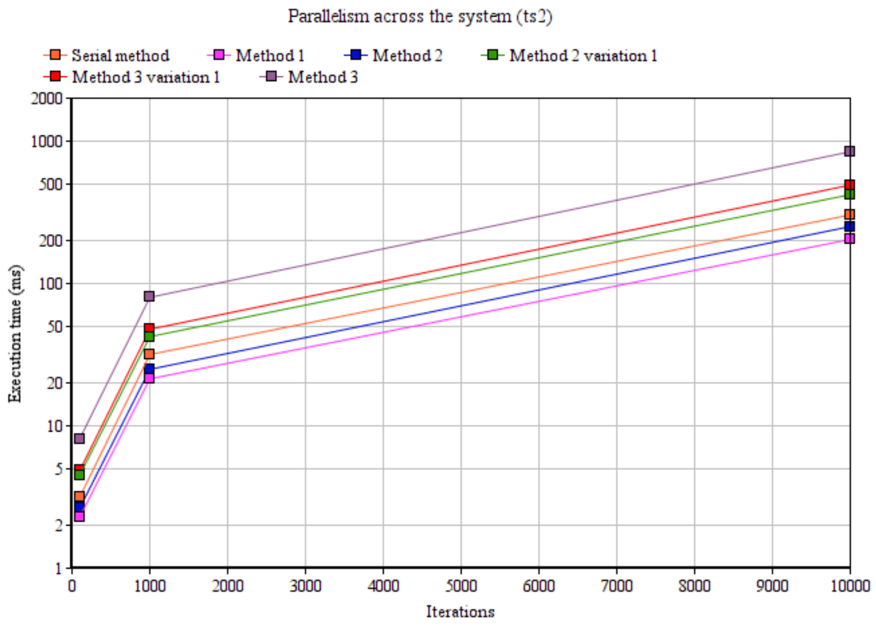


Figure 6.2: A chart with the results of the methods using parallelism across the system for ts2

Table 6.14: Cumulative results of the methods using parallelism across the system for the system ts3. The results of the serial method are also included.

iterations	100	1.000	10.000
Serial method	114,01 ms	1.193,63 ms	11.114,80 ms
Method 1	56,39 ms	567,26 ms	5.390,43 ms
Method 3 variation 1	57,77 ms	566,51 ms	5.770,89 ms
Method 2	68,48 ms	598,86 ms	6.188,28 ms
Method 3	63,13 ms	633,30 ms	6.410,67 ms
Method 2 variation 1	72,48 ms	712,09 ms	7.034,90 ms

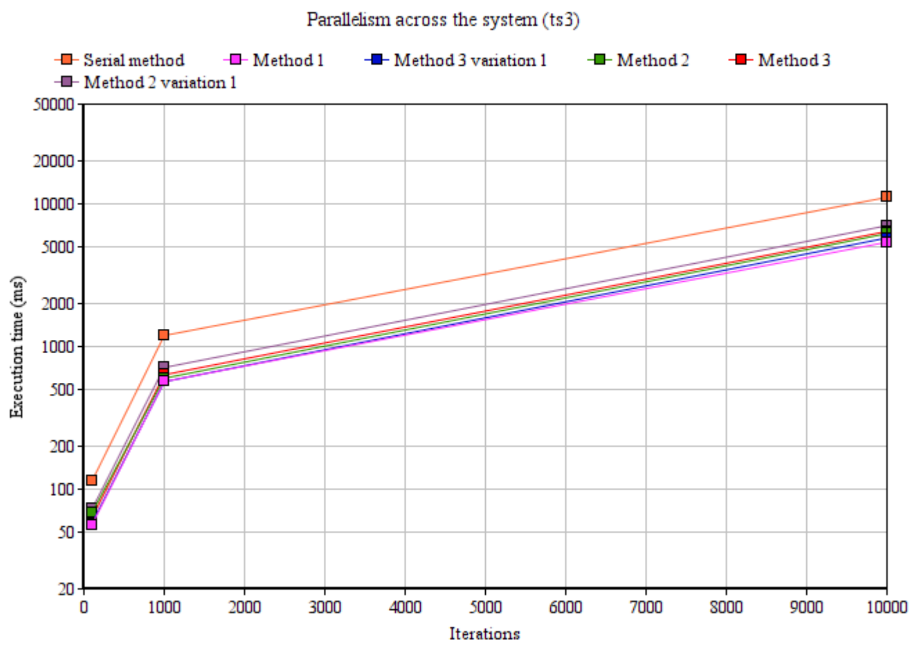


Figure 6.3: A chart with the results of the methods using parallelism across the system for ts3

6.4.1 Results for Method 1

For the system ts2 the results were

Table 6.15: Parallelism across the system, results for method 1 (ts2)

iterations	100	1.000	10.000
execution time	2,29 ms	21,38 ms	203,77 ms

and for the system ts3

Table 6.16: Parallelism across the system, results for method 1 (ts3)

iterations	100	1.000	10.000
execution time	56,39 ms	567,26 ms	5.390,43 ms

For the first time the parallel methods surpassed the serial's performance. As it can be seen, the more teams grow in numbers and size, the better for the performance compared to the serial method.

6.4.2 Results for Method 2

For the system ts2 the results were

Table 6.17: Parallelism across the system, results for method 2 (ts2)

iterations	100	1.000	10.000
execution time	2,65 ms	24,93 ms	250,91 ms

and for the system ts3

Table 6.18: Parallelism across the system, results for method 2 (ts3)

iterations	100	1.000	10.000
execution time	68,48 ms	598,86 ms	6.188,28 ms

Another interesting result, as this method is only a bit slower than the previous one, but eventually slower than expected. A possible explanation is that the delay comes from the receiver part, as it has to process the messages in every iteration before extracting their values, combined with the fact that the values are being sent in every iteration.

6.4.3 Results for Method 2 variation 1

For the system ts2 the results were

Table 6.19: Parallelism across the system, results for method 2 variation 1 (ts2)

iterations	100	1.000	10.000
execution time	4,51 ms	42,27 ms	420,59 ms

and for the system ts3

Table 6.20: Parallelism across the system, results for method 2 variation 1 (ts3)

iterations	100	1.000	10.000
execution time	72,48 ms	712,09 ms	7.034,90 ms

For the small system a significant slowdown was noticed compared to the previous method with channels. For the bigger system the difference is smaller but still noticeable. Assumptions for these disappointing results were that the bigger message size may have affected performance and the map preparation before sending every time might have been too much. It is noteworthy that the number of sends didn’t affect much the performance of the method.

6.4.4 Results for Method 3

For the system ts2 the results were

Table 6.21: Parallelism across the system, results for method 3 (ts2)

iterations	100	1.000	10.000
execution time	8,13 ms	79,84 ms	841,71 ms

and for the system ts3

Table 6.22: Parallelism across the system, results for method 3 (ts3)

iterations	100	1.000	10.000
execution time	63,13 ms	633,30 ms	6.410,67 ms

The results confirmed the initial guess about the number of the functions. When there are only a few like in the first test system (ts2), the thread creation overhead dominates the gains. Furthermore, the guess about the slowdown when the logical threads are not enough was also verified by the results of this method, where four threads run the serial method for their independent teams and for each of these threads another four are created to calculate the functions at each integration step. However, the performance of this method is believed to be the highest of all methods in a system with enough logical threads, but such a system wasn't available for testing.

6.4.5 Results for Method 3 variation 1

For the system ts2 the results were

Table 6.23: Parallelism across the system, results for method 3 variation 1 (ts2)

iterations	100	1.000	10.000
execution time	4,86 ms	47,64 ms	490,79 ms

and for the system ts3

Table 6.24: Parallelism across the system, results for method 3 variation 1 (ts3)

iterations	100	1.000	10.000
execution time	57,77 ms	566,51 ms	5.770,89 ms

For larger and more teams, this method is on par with method 1 which is the fastest method tested. For smaller input, its performance is worse than most of the above methods.

6.5 Discussion

The results of the methods using parallelism in time were disappointing. None of these methods surpassed the serial implementation in terms of performance and the best one was on par with it only for a low amount of iterations. This has to do with the amount of equations each thread has to calculate and the type of communication between the threads. On the other hand, the methods using parallelism across the system outperformed the serial method by far. The difference got even bigger when more teams with more equations were found. In addition, it is showed that in numerical procedures like the Euler method which need a lot of synchronization to calculate their values, parallelism across the system might be the right way to go.

Chapter 7

Web Application

This chapter presents the web application that was developed, where the users can simulate their systems.

7.1 Why this application was developed

There were several practical reasons that led to the creation of this application. Before it became a thing, at the time when the methods using parallelism in time were being tested, all the equations describing a system had to be converted manually to prefix form and be put inside a function. It gets even worse, because these methods took as input the dependencies between the equations as explained in chapter 4. This was a very time consuming operation and something had to be done about it. Also, the results were just numbers printed in the console; making it hard to check it they were correct especially for larger numbers of iterations. Furthermore, it would be nice to have an application with a GUI where users could write equations, simulate their systems and inspect the results.

7.2 Deciding on a desktop or web application

There were two options for the application, a desktop app or a client server app. For the first option the 'seesaw' [45] library could be used, which is a wrapper of Swing [46] for Clojure. However, the second was clearly a better option as it would allow access from any place and most importantly it would run on a server which meant that the user doesn't need to have a multi-core CPU to get the benefits of the parallel integration.

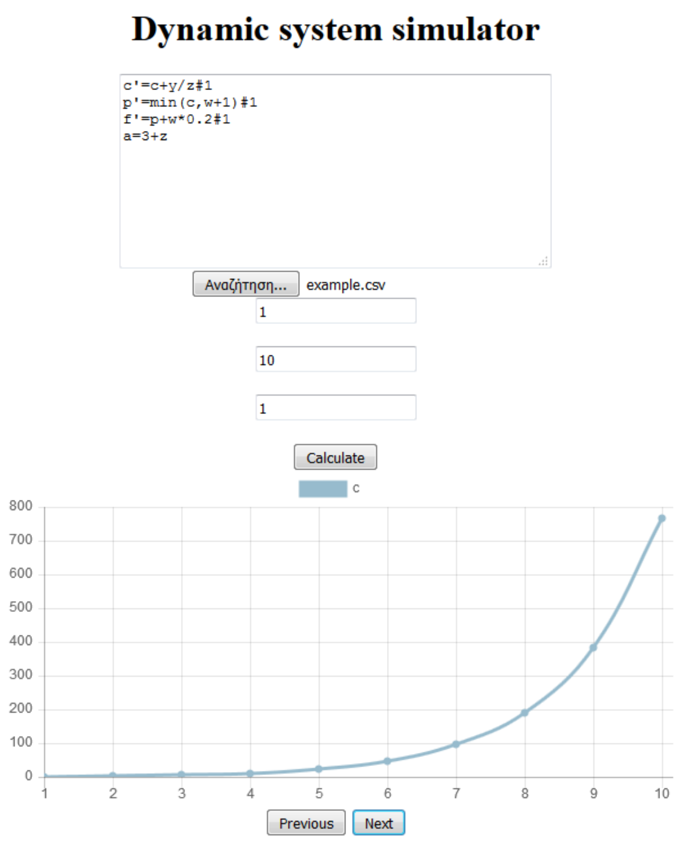
There were many ways to do this, from ready to use frameworks like Pedestal [47] and Hoplon [48], to composable libraries like Ring [49], Compojure [50], Hiccup [51] etc. One could develop such an application writing only Clojure and Clojurescript [52]. In this work, due to time constraints, it was decided that only the server side would be written in Clojure, using some of the above libraries that can be stacked together and the client in classic HTML/Javascript which were familiar.

7.3 Application capabilities

The application can take as input

- differential equations with their initial values
- ordinary equations
- constants appearing in the equations
- CSV files containing values

and plots the results, one variable at a time.


Dynamic system simulator

```
c'=c+y/z#1
p'=min(c,w+1)#1
f'=p+w*0.2#1
a=3+z
```

Αναζήτηση... example.csv

1

10

1

Figure 7.1: A screenshot from the web application developed. The input is three differential equations, an ordinary one and a csv file with variables appearing in the equations. Next follow the fields containing the start, end and step of the integration. After that, there is a plot containing the results for the variable c.

Chapter 8

Conclusion and Future Work

This chapter discusses the conclusions of this work and presents thoughts about what could be improved to make the system partitioning algorithm better and the integration procedure faster.

8.1 Discussion

The experiments showed that Clojure’s features facilitate the creation of parallel systems. The reference types provide ways to develop synchronous and asynchronous systems with the desired performance characteristics accordingly. Refs can be used to create systems with coordinated changes without implementing the coordination mechanisms from scratch and with CRU [53] functions, agents can be used for asynchronous systems where a task is dispatched to another thread and it is uncertain when the other thread will return without having to program the queueing of the actions and the dispatching to the provided threadpool, atoms provide synchronous changes meaning that only one thread can access it at a time without having to keep track which threads has to take the lock each time and vars are nice to use because of their global score and different value on a per thread basis.

The effectiveness of the created methods depend on the parallelism type used. For parallelism in time, the results showed that it is slower than its serial counterpart. The main reason was that a lot of time was wasted for the thread synchronization and also because some threads were idle while waiting for others to complete their calculations in order to proceed to the next iteration. On the other hand the results of parallelism across the system showed an important performance boost as the previous two factors holding back the performance in parallelism in time didn’t exist.

It is worth noting that these methods were made modular and the Euler method used can be substituted easily by another. Apart from that, the performance of parallelism in time is tied to the integration procedure used. So, the best performance through parallelization be achieved, in an integration procedure independent way can be achieved by using parallelism across the system with a partitioning algorithm like the one used in this work.

8.2 Conclusion

To sum up, the methods using parallelism in time performed worse than expected. None of them surpassed the performance of the serial method. That was mainly due to the synchronization operations between the threads which created overhead. Things changed with the methods using parallelism across the system, as all of them had better performance than the serial one. An important component of these methods is an efficient system partitioning algorithm which will find as many independent teams as possible. In all of these methods the numerical procedure used can easily be replaced by another one due to the modular design, which Clojure made easier. Of course, there is still plenty of room for experimentation, to try new communication techniques between the threads, alternative algorithms which may be faster for the system partitioning and libraries that boost the performance of mathematical operations.

8.3 Future work

There are many areas where the discussed systems could be improved. For absolute performance, Java's primitives could replace the auto inferred types of Clojure. Also, type hinting, extensive use of transients and inlining functions aggressively using 'definline' would provide a performance boost. Furthermore, libraries like clj-tuple [30] that provide more efficient data structures could be beneficial.

For the part of grouping functions into teams, load balancing is an interesting dynamic problem that is worth to be investigated, as the teams may be equally distributed to threads but that doesn't mean the number of functions is too.

In addition, a better implementation of the grouping algorithm could be achieved by using a faster graph exploration algorithm than the depth first search.

Finally, the grouping algorithm could be enhanced to find even bigger subsystems which is unable to recognize now. For example, in the system below only the functions in the red circles will form teams. But all these functions could be grouped (blue circle) as they are only a component of a bigger system.

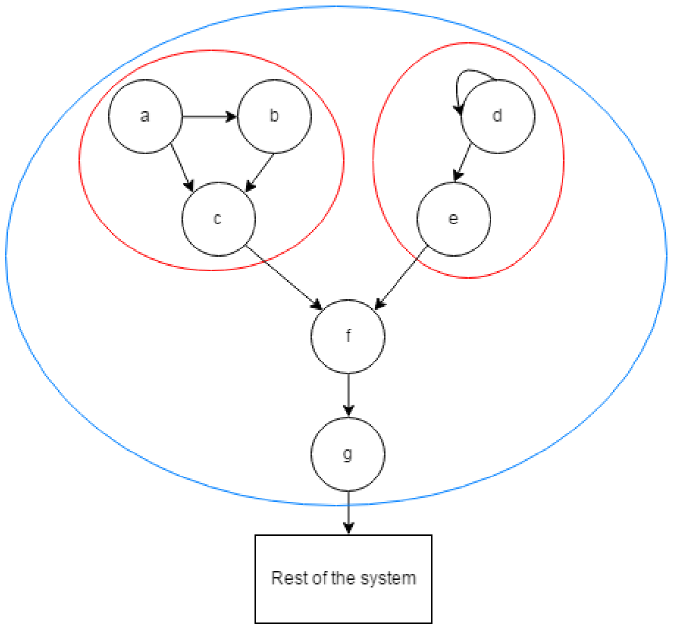


Figure 8.1: An example of a graph showing the detection capabilities of the algorithm used for graph partitioning. The red circles show what the algorithm can detect as independent teams. The blue circle shows the ideal team, a whole subsystem, which the algorithm is unable to recognize.

References

[1] Shrirang Abhyankar, Alexander J. Flueck: Real-Time Power System Dynamics Simulation using a Parallel Block-Jacobi Preconditioned Newton-GMRES scheme

[2] R. S. Hwang, D. S. Bae, J. G. Kuhl : Parallel Processing for Real-Time Dynamic System Simulation

[3] Matthieu Schaller, Pedro Gonnet, Aidan B. G. Chalk, Peter W. Draper: SWIFT: Using task-based parallelism, fully asynchronous communication, and graph partition-based domain decomposition for strong scaling on more than 100 000 cores.

[4] National Instruments: http://www.ni.com/tutorial/11606/en/

[5] Wikipedia: https://en.wikipedia.org/wiki/Dynamic_simulation

[6] Andrew Ford: Modeling the Environment, Second Edition, ISLAND PRESS, Chapter 4 page 39, Chapter 5 page 48

[7] Masahiro Yoshida, Osamu Okamoto, Heihachiro Kamimura, Keisuke Ishihara: Study of Structural Analysis by Large Scale Parallel Multi-body Dynamic Simulation

[8] Wikipedia:
https://en.wikipedia.org/wiki/Parker%E2%80%93Sochacki_method#Advantages

[9] Wikipedia: https://en.wikipedia.org/wiki/Cash%E2%80%93Karp_method

[10] Wikipedia: https://en.wikipedia.org/wiki/Dormand%E2%80%93Prince_method

[11] Wikipedia:
https://en.wikipedia.org/wiki/Adaptive_stepsize#Embedded_error_estimates

[12] Wikipedia: https://en.wikipedia.org/wiki/Initial_value_problem

[13] Dana Petcu, Andrian Baltat: Studies in Computational Intelligence 217, Intelligent Systems and Technologies: Methods and Applications, Springer, Chapter 8, page 155

[14] Kevin Burrage: Parallel Methods for ODEs, Advances in Computational Mathematics 7 (1997) 1-3

[15] http://www.openmp.org/

[16] https://www.open-mpi.org/

[17] Wikipedia: https://en.wikipedia.org/wiki/OpenMP#Pros_and_cons

[18] Wikipedia: https://en.wikipedia.org/wiki/Open_MPI#Overview

[19] Jacques-Louis Lions, Yvon Maday, Gabriel Turinici: A “parareal” in time discretization of PDE’s, Comptes Rendus de l Académie des Sciences - Series I - Mathematics 332(7) – January 2001

[20] Julien Cortial, Charbel Farhat: A Time-Parallel Implicit Method for Accelerating the Solution of Nonlinear Structural Dynamics Problems, International journal for numerical methods in engineering

[21] Matthew Emmett, Michael L. Minion: Toward an efficient parallel in time method for partial differential equations, Communications in Applied Mathematics and Computational Science, Vol. 7, No. 1, 2012

[22] Anastasios Georgoulas, Allan Clark, Andrea Ocone, Stephen Gilmore, Guido Sanguinetti: A subsystems approach for parameter estimation of ODE models of hybrid systems

[23] C.W. Gear: Waveform methods for space and time parallelism, Journal of Computational and Applied Mathematics 38 (1991) 137-147

[24] C.W Gear: Massive parallelism across space in ODEs, Applied Numerical Mathematics 11 (1993) 27-43

[25] Wikipedia: https://en.wikipedia.org/wiki/Read%E2%80%93eval%E2%80%93print_loop

[26] http://vensim.com/

[27] https://www.iseesystems.com/store/products/stella-professional.aspx

[28] Andrew Ford: Modeling the Environment, Second Edition, ISLAND PRESS, Chapter 4 page 39, Chapter 4 page 44

[29] Gurunath Gurrala, Aleksandar Dimitrovski, Pannala Sreekath, Srdjan Simunovic, Michael Starke: Parareal in Time for Dynamic Simulations of PowerSystems

[30] Robert D. Skeel, Hon-Wah Tam: Limits of parallelism in explicit ODE methods, Numerical Algorithms 2 (1992) 337-350

[31] Wikipedia: https://en.wikipedia.org/wiki/Multi-level_technique

[32] Aaron S. Pope, Daniel R. Tauritz and Alexander D. Kent: Evolving Multi-level Graph Partitioning Algorithms

[33] Umit V. Catalyurek, Erik G. Boman, Karen D. Devine, Doruk Bozdag, Robert Heaphy, Lee Ann Riesen: Hypergraph-based Dynamic Load Balancing for Adaptive Scientific Computations

[34] Karen D. Devine, Erik G. Boman, Robert T. Heaphy, Rob H. Bisseling, Umit V. Catalyurek: Parallel Hypergraph Partitioning for Scientific Computing

[35] Rice University, Department of Computer Science: https://www.cs.rice.edu/~vs3/comp422/lecture-notes/comp422-lec4-s08-v1.pdf

[36] Bruce Hendrickson, Tamara G Kolda: Graph partitioning models for parallel computing

[37] Wikipedia: https://en.wikipedia.org/wiki/Conditional_dependence

[38] https://clojure.org/reference/reducers

[39] Wikipedia: https://en.wikipedia.org/wiki/Topological_sorting

[40] Wikipedia: https://en.wikipedia.org/wiki/Closure_problem

[41] Wikipedia: https://en.wikipedia.org/wiki/Strongly_connected_component

[42] Wikipedia: https://en.wikipedia.org/wiki/Dependence_analysis

[43] Wikipedia: https://en.wikipedia.org/wiki/Eulerian_matroid

[44] https://github.com/hugoduncan/criterium

[45] https://github.com/daveray/seesaw

[46] Wikipedia: https://en.wikipedia.org/wiki/Swing_(Java)

[47] https://github.com/pedestal/pedestal

[48] https://hoplon.io/

[49] https://github.com/ring-clojure/ring

[50] https://github.com/weavejester/compojure

[51] https://github.com/weavejester/hiccup

[52] https://clojurescript.org/

[53] Wikipedia: https://en.wikipedia.org/wiki/Create,_read,_update_and_delete

[54] https://github.com/ztellman/clj-tuple

[55] Chas Emerick, Brian Carper, Christophe Grand: Clojure Programming, Chapter 4 page 209

[56] https://github.com/clojure/core.async

[57] Wikipedia: https://en.wikipedia.org/wiki/Topological_sorting#Kahn.27s_algorithm

[58] https://github.com/ring-clojure/ring/tree/master/ring-jetty-adapter

[59] http://www.http-kit.org/

[60] http://immutant.org/

[61] https://github.com/dgrnbrg/spiral

[62] https://github.com/dakrone/cheshire

[63] http://www.chartjs.org/

[64] https://github.com/rm-hull/infix

[65] https://github.com/tristan/clojure-infix

[66] https://blog.idrsolutions.com/2014/04/profiling-vs-sampling-java-visualvm/

Appendix A. Euler method

The Euler method is a first order numerical procedure for solving ordinary differential equations with a given initial value. It is an explicit method which means that the state of a system is calculated at a later time from the state of the system at the current time. Also, it is the simplest Runge-Kutta method and serves as a building block for more complex methods. The Euler method is given by the following equation

$$y_{n+1} = y_n + h * f(t_n, y_n)$$

where h is the step size. Example: Given that $y' = y+1, y(0)=1, \ h=1$ approximate $y(5)$. Using the above equation to successive iterations it occurs

$$\begin{aligned} y_0 + h * f(y_0) = y_1 \\ y_1 + h * f(y_1) = y_2 \\ y_2 + h * f(y_2) = y_3 \\ y_3 + h * f(y_3) = y_4 \\ y_4 + h * f(y_4) = y_5 \end{aligned}$$

and $y_5 = 63$.

Appendix B. Clojure data structures and reference types

Built-in data structures

Clojure has a rich set of data structures. The most common are lists, maps and vectors. Lists are an integral part of the Clojure syntax as every command is a list (Clojure handles code as data). Also, they are rarely used as data containers. The complexity to access an element is $O(N)$.

The maps are mostly used to model data containers and can be nested, thus creating very flexible storage containers. The complexity to access an element of a map is $O(\log_{32} N)$ which is nearly constant.

Vectors have common elements to arrays of other languages. There are several ways to iterate through them but the most familiar one is by using their indices. Their access time complexity is $O(1)$.

Choosing the right data structure is crucial because they follow different abstractions and their performance heavily depends on their use. For example, a list can efficiently append an element to the front while vectors to the back. It is possible to change this but the constant time guarantee for the append operation will be lost.

Structural sharing

Data structures in Clojure are immutable. In order to update a value a new data structure holding it will be created. So the question is how is this efficient. The answer is structural sharing, which is a technique to decide what portion of an existing data structure will be copied to the new one. The smallest the portion the fastest the operation will be. In many situations Clojure just puts new pointers to existing data structures, with each pointer meaning a new state of it. For example having a list and appending an element is done in the aforementioned way. Before the append, there is a pointer pointing to the first element. At the moment of the append, a new pointer is created pointing to the new element at the front but the rest of the list remains the same.

Transient data structures

Transient data structures solve the problem of performance when structural sharing is not enough. They are mutable data structures that provide fast transformations and are ideal when there are frequent changes in data. When the change operations end these data structures can be converted into immutable. The drawback is that while they are mutable they don't implement some interfaces of the immutable, which makes them dangerous to handle and that's why their use is advised to be in a single threaded scope.

Reference types

In Clojure there are four reference types: vars, refs, agents and atoms. These behave very differently when used for concurrency and have their own semantics for managing change. One thing they have in common is that in order to access their contained value they need to be dereferenced. Dereferencing will return a snapshot of the state of a reference when the dereferencing was done. It is important to note that the dereferencing operation will never block.

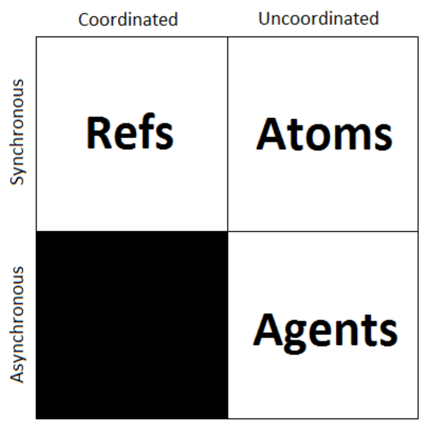


Figure B.1: Clojure reference types and their concurrency semantics (taken from 'O'REILLY-Clojure Programming, Chas Emerick, Brian Carper, Christophe Grand').

Atoms are the most basic reference type, they implement synchronous, uncoordinated, atomic compare-and-set modification. Operations that modify the state of atoms block until the modification is complete and each modification is isolated; without using other components it's impossible to orchestrate the modification of two atoms.

The only coordinated type is refs. Using them guarantees some things. First, there will be no race conditions between the refs. Second, there is no possibility of deadlocks. Third, there is no possibility of the involved refs to be in an observable inconsistent state. Finally, there is no need for manual locks, monitors or other low level synchronization primitives. This is achieved with the built in software transactional memory (STM) that Clojure has.

Vars differ from the other reference types in that their state changes are not managed in time; A var is a namespace-global identity that can be rebound to have a different value on a per-thread basis.

Agents are an uncoordinated, asynchronous reference type. Changes to an agent's state are independent of changes to other agents' states, and all such changes are made away from the thread of execution that schedules them. There are two characteristics that separate them from atoms and refs. First, I/O and other side-effecting functions may be safely used in conjunction with agents. Also, agents are STM-aware, so that they may be safely used in the context of retrying transactions. Each agent queues the actions that assigned to it and at some unknown time dispatches them to a thread chosen from a thread pool in order to calculate its new value. An implementation detail regarding thread handling is that in order to send an action to an agent there are two functions, 'send' and 'send-off'. With 'send' the agent will dispatch the action to a thread in the existing fixed size thread pool that Clojure uses for agents. Using 'send-off' will result in the creation of a thread that will stay alive for one minute if it completes its task and is not assigned other ones, and then will be terminated. In Clojure relevant literature [55] ‘send’ is recommended to be used for computationally intensive tasks and ‘send-off’ for Input/Output tasks or tasks that may block the threads.

Futures

Futures are the simplest way to execute a body of code in another thread. Future returns immediately allowing the current thread of execution to carry on. The result of the evaluation is retained by the future and can be obtained by dereferencing it. A future is a construct which allows the execution of code in another thread. This thread is chosen from a thread pool if the pool is not empty; otherwise a new thread is created. To clarify, with the current future implementation in Clojure, when a future is called ‘send-off’ is used in order to create a thread. This thread completes its task and then stays idle in a thread pool. If it remains unused for one minute it dies. This pooling of resources can make futures more efficient than creating native threads as needed.

Promises

Promises are not created with any code or function to define their values, they are just bare containers. Data is inserted one time and can be obtained by dereferencing them. This behavior is sometimes called dataflow variable and is the building block of declarative concurrency. A promise may be dereferenced with an optional timeout, dereferencing a promise will block until it has a value to provide and a promise will only ever have one value.

Appendix C. Evolution of the system partitioning algorithm

The presented system partitioning algorithm went through several changes to evolve into its current state. In the beginning, some rules had to be devised which would lead to function grouping. By taking a closer look at the way the functions are given as inputs many rules became evident. All the self dependent functions of the form

$$[0\ f\ f]$$

$$[0\ g\ g]$$

could be grouped together. Also, on the same team could be functions that depend only on functions of the first category like the following

$$[0\ f\ g\ h]$$

$$[0\ f\ k]$$

Another way to find teams is to look at the functions of each input vector and find which vectors have exactly the same. To clarify the rule refers to situations like this

$$\begin{gathered}[0\ a\ b\ c]\\ [0\ a\ c\ b]\end{gathered}$$

where a, b, c are functions which depending on their position in the vector, show a function name or from where a value is taken. These functions will form a team that can run asynchronously if all their dependencies are functions inside the team. For example, these functions form an independent team

$$\begin{gathered}[0\ a\ b\ c\ k]\\ [0\ b\ k\ a\ c]\\ [0\ c\ b\ k\ a]\\ [0\ k\ a\ c\ b\ b]\end{gathered}$$

while the next ones don't

$$\begin{gathered}[0\ l\ b\ c\ k]\\ [0\ b\ k\ l\ c]\\ [0\ c\ l\ k\ b]\end{gathered}$$

because even though they have the same functions, function l is not included in the team. As a result, the whole team will have to wait for l to produce its values in order to continue. Notice that the last function of the first example depends on itself. This doesn't cause any problems and the code is easily adjusted to handle such cases.

This method works for specific dependency topologies by it fails in more complex situations like below

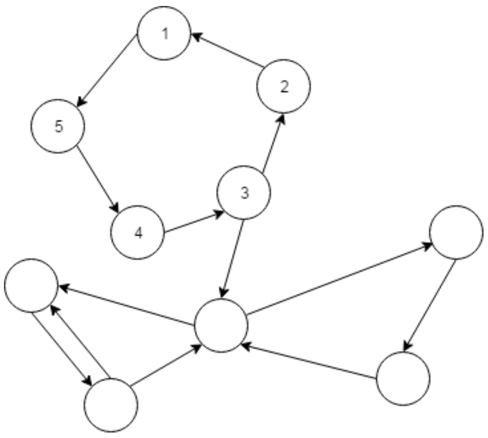


Figure C.1: Example of a subsystem which won't be detected (nodes 1, 2, 3, 4, 5).

The nodes 1, 2, 3, 4, 5 could form a team but they won't be recognized as such with the rules defined before. The reason for this is that it is not a systematic method. It is based on some observations and the rules cover only a handful of the available dependency cases.

Then the first part of the current algorithm was created. According to it, if some nodes have the same reachability, they are grouped together. The groups are independent if all the dependencies of the nodes in the groups are nodes of the group.

A known flaw of the aforementioned algorithm was that it couldn't find independent teams in some general function dependency patterns. These cases are often met in real world systems so something had to be done about it. For example

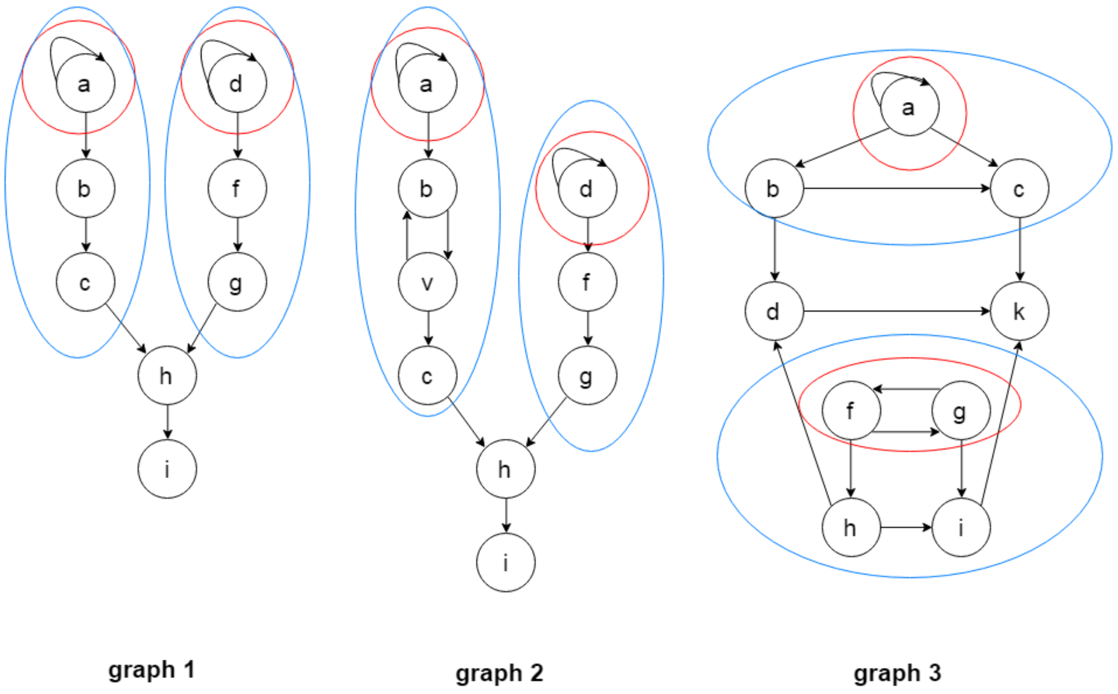


Figure C.2: Cases where the initial partitioning algorithm would fail to detect the independent teams. In the red circles are the teams detected by the initial algorithm. The blue ones show what the algorithm should be able to detect and was modified to do.

In the red circles is what the previous version detects as independent teams and in blue the current one. All the examples fall into the same general pattern of sequential nodes, but they are chosen to show how they algorithm works and what it can detect.

graph 1: the simplest case, sequential nodes. So, building on the existing algorithm

- For each neighbor of each node of each independent team, check if it can be added to the team. If it has no other dependencies except from inside the team, get it.

graph 2: notice that nodes (b, v) connect to each other. The bullet would fail because when checking (b) it would be shown that it depends from nodes (v) outside the independent team. But it is known that nodes which have the same reachability are grouped into dependent teams. And the nodes (b, v) have the same reachability so they form a dependent team. Now let's improve it

- For each neighbor of each node of each independent team, check if the dependent team it belongs to can be added to the independent team. This can happen if the dependent team has no other outer dependencies than the independent team.

  Continuing with graph 2, notice that nodes (c, g) have the same reachability. So, they form a dependent team. Jumping in the step where the algorithm has found that (a, b, v) form an independent team, when it checks (c), it will see a dependency from (f). That's because it checks the whole dependent team the node belongs to for outer dependencies. To improve again

- For each neighbor of each node of each independent team, check if the dependent team it belongs to can be added to the independent team. This can happen if the dependent team has no other outer dependencies than the independent team. If it can't, check only for this node.

graph 3: the unique element of this graph is that the order of the examination of the nodes to take into team matter. Starting from node a, with bad luck it happens to check neighbor (c) first. It is evident that it can't be taken into the independent team because it depends from another node (b), which is in another dependent team. To overcome this problem the third bullet becomes

- For each neighbor of each node of each independent team, check if the dependent team it belongs to can be added to the independent team. This can happen if the dependent team has no other outer dependencies than the independent team. If it can't, check only for this node. If something was added to the team start checking from the beginning the neighbor nodes.

The above also works if the corresponding node of (b) was dependent from another node of the independent team and that is shown with nodes (h, i).

Appendix D. Method implementation details for parallelism across the system

core.async

This library [56] was used to invoke threads and facilitate the communication between them for the methods using parallelism across the system. It provides a nice way of interprocess communication through channels. A channel works like a FIFO queue, where a message is put on one end, and it is delivered on the other end. When a process sends a message it has no control over it and what will happen with it. This queue can have a buffer to support multiple messages if needed. It also supports drop policies to decide what to do if the queue is full or the receiver is busy. Furthermore, the insertion and extraction of messages from a channel can happen with both blocking and non-blocking ways. To clarify, a process will block if it tries to put a message on a buffered channel that is full or if it puts a message on an unbuffered channel where the receiver has not finished processing the previous message. In addition, the extraction can block the waiting the thread if desired. Another nice feature is that it provided the ability to use go threads. These threads are lightweight, meaning that they are cheap to create and don't use the JVM's native threads. They also follow the inversion of control principle giving them the ability to park when a process blocks and leave the actual thread of control free for another thread.

An optimization for the calculation of the ordinary functions

This section was put in this specific place because this optimization was done only for the methods using parallelism across the system, as the methods using parallelism across time weren't supported by the web application.

The way the ordinary functions are calculated during the integration, if a function doesn't have all the parameters ready, because of dependencies from other ordinary functions, it is put on the tail of a queue to be calculated later. This works but there are cases where the order of the functions may be completely reverse from the correct order of calculation. For example

$$\begin{aligned} x' &= x+1 \\ k &= w+1 \\ w &= z+1 \\ z &= y+1 \\ y &= x+1 \end{aligned}$$

the queue for the ordinary equations will be k, w, z, y. But y has to be calculated first. Until then, k, w and z will be pushed to the tail of the queue. That will happen again for the calculation of k and w because z has to be calculated first and so on. The problem is that this happens in every iteration wasting time. A solution is to put them in the correct order before the integration starts. Topological sorting is the ideal candidate for this because the ordinary equations can't have cyclic dependencies as they depend on the present value of each other. Among the algorithm of Kahn [57] and a modified depth first search, the first was used in order to find a possible order. Having the equations in the correct order no pushing to tail was needed and thus they could be calculated in one pass.

Appendix E. Web app implementation details

Ring

The Ring library provides a simplified API to use the HTTP. There are four components consisting an application which uses Ring: handlers, requests, responses and middleware. Handlers are functions which define what will be done with a request and return a response. Requests and responses are modeled as maps with the appropriate headers of the HTTP. Finally, middleware are higher level functions that take a handler as first argument and provide to it additional functionality.

Compojure

Compojure is a routing library whose purpose is to facilitate the routes management. After the route creation, a provided macro outputs a Ring handler for them.

Jetty

For the server, an adapter compatible with Ring must be used if someone doesn't want to write his own. There are plenty of choices for that with the most popular being Jetty [58], http-kit [59] and Immutant [60]. Here Jetty was used as it is also supported by the library described below.

Spiral

Spiral [61] is a library that among others makes a server work in an asynchronous way. It is doing so by sitting upon Ring and blending its functionality with core.async. A nice feature, that is not needed here, is a provided middleware that can prioritize routes that serve static data over routes that serve DB queries.

Client side

The website consists of one page, as more would be redundant. It contains a textarea to write the equations, three fields for start, end, step, a button to send them to the server, a file upload field, a canvas element to draw plots and two more buttons to navigate between the plots. Regarding the communication, the request was decided to be a POST request, because POST requests don't have a data length limit. This is handy because it gives the ability to send a large amount of equations to the server. Furthermore, the request was made asynchronous because the calculations might take a lot of time. For this, the XMLHttpRequest API was used. The format of the sent data was decided to be JSON, as there are libraries converting JSON directly to Clojure data structures, like the Chesire [62] library. The structure of this JSON was

{strings: [strs], start: start, end: end, step: step} where ‘[strings]’ was an array of the equations; the rest of the fields are self-explanatory. The answer was also a JSON string. When it was received, it was converted to a Javascript object and the results were passed to the Charts.js [63] library which was responsible for the plot drawing.

Server side

The server was made to respond in a synchronous manner in the beginning but that was changed, using the Spiral library, as calculations could last long. Three routes were created, one for serving the static page at the default location, one for serving the POST request and one for showing the HTTP not found code for wrong addresses. When the request comes, the JSON string is extracted and is processed with the Chesire library to create a Clojure map with the corresponding keys and values. This map passes through a stage of preprocessing to create the system-map and then the integration can begin. After the integration ends, a map is created with keys being the function names and values being vectors containing the produced values. This map is converted to a JSON string with Chesire and sent back to the client.

Finally, a convention was made for the way the equations would be written on the textarea of the client because the server should know how to handle them. It was decided that they should be written one after the other in the format

$$name' = \exp ression\#initial\_\mathrm{value}$$

Giving files as input

Many dynamic systems need to take values from external sources in order to be simulated. This is done when it’s difficult to model a variable quantity into a function. Such quantities are temperature and carbon dioxide density in the air. For our integration system it was interesting to add this capability.

It was decided to be done for csv files as it is a widespread format. A convention made here was that the first row of the files should contain the variable names and rest the values; for the simple reason that the system should know which symbols of the equations to replace with the values. The first step was to put a file upload field on the website where the user could choose multiple files. Next, there were three options concerning the file handling. The first was to convert the file to Base64 format and create a JSON from it, this JSON and the one with the equations could be merged and be sent in one POST request. The second was to post the file first, the server to respond with a unique ID, and the client to post the JSON with the equations and the ID so the server could make the association between the two. The third was to extract the values from the files and put them in the same JSON with the equations. The third solution was preferred because it was considered simpler. Also, there isn’t any request size limit because of the POST type of request and as a result a huge number of values inside the files is not a problem. The drawback is the processing of the files that must be done to the client side which might be slow if the client is an old computer. The new JSON string is in the format

```
{strings: [strs], start: start, end: end, step: step, fileValues: {a: [1,2,3..], b: [7,8,9...] ...}}
```

where a and b are variables read from files. On the server side, not many things had to change as the Javascript object kept in the fileValues property is converted to a map which can be merged directly with the produced values map allowing the equations to know where to pull from their values. Eventually, no extra overhead is introduced to the integration procedure.

Appendix F. Web app input preprocessing

Infix to prefix conversion

To convert from infix to prefix there were two available libraries, Infix [64] and clojure-infix [65]. Only Infix was being actively maintained so it was preferred. As an extra, it gave the ability to evaluate the parsed expressions and also create functions with multiple inputs.

Infix

The Infix library has functions that handle strings and evaluate their content. It is interesting that it does so without depending on 'eval' and 'read-string', but instead using a monadic parser in combination with an EBNF grammar. It also contains a useful macro that accepts an infix expression and the symbols to be found in the expression as parameters and returns a function that calculates it.

An Infix macro inconveniency

So, the first step was to extract the names of the parameters in a given string in infix form. Then, the collection with the parameter names was passed to the aforementioned macro and that's where a compile time exception occurred because the macro couldn't take a collection as a parameter and unfold it esoterically. Furthermore, in Clojure it's not possible to unfold a collection at runtime in order to pass the contents as parameters, so it had to be done at compile time. For that, a new macro was created, as macros expand to 'regular' code at compile time, which took a collection, unfolded it, and passed the contents to the previous macro. This led to a runtime exception which stated that there were symbols not found in the given parameters and so it wasn't known what to do with them. After communicating with the author it was made clear that the library is meant to be used for creating functions with a known number of parameters, although the second exception is probably a bug. Also, he proposed two ways to achieve the same behavior with the macro but with collections as input.

Preprocessing

After a function has been created with Infix, a map is composed (function-map) with key the function name and value another map. The second map holds the initial value, the parameters and the expression of the function. When all the function-maps are created they are merged together to form the system-map. This map has all the information needed about the functions for the integration.

Extending the preprocessing capabilities

In the beginning the application was able to accept only differential equations. But most real dynamic systems aren't modeled using only differential equations but also ordinary ones. In addition, there are variables that is interesting to know how they develop through time, and who depend on these equations. Also, it would be nice if the users could write constants in along with the equations without having to substitute them to the equations themselves. These reasons led to the extension of the preprocessing and integration systems to support such features. The first step was to make it recognize constants and replace them to the expressions. If after an equality operator follows only a number then that's a constant. Constants that depend on other constants are not treated as constants. That comes down to what is returned to the client to be plotted. Constants are not something interesting for the user to see. On the other hand, constants that depend on other constants is something that maybe the user wants to see as a result. So they are returned with the other results to be plotted as well. After that, the preprocessing stage was expanded to let it distinguish the difference between differential and ordinary equations and handle them accordingly. The sign for a differential equation is the presence of a comma. If an equation is not differential its initial value is calculated from the differentials and constants it depends from. Next, an extra key was added to the function-maps indicating if it's a differential equation or not. This is useful information for the integration system because the differential equations have to be calculated first and then the ordinary ones in each iteration. The reason for this is that the ordinary equations depend on the present value of the differential ones. Also, the mathematic functions of min, max and ln were added to the functions which the Infix library parser recognizes.

Several unexpected obstacles appeared while doing the above changes. Due to some exceptions occurring at runtime at the parsing of the expressions it was thought that there should be whitespaces between the operations and the variables but later that proved to be wrong. Another problem was that the Infix library cannot understand negative numbers. For example, the following is not a valid expression because the minus doesn't have a symbol or number on its left

$$x' = -2 * x$$

It was noticed that a negative number can appear in three ways inside an expression, right after the equality operator like the above, immediately after a parenthesis

$$x' = (-2) * x$$
$$x' = (-2 * x)$$

or inside a parenthesis as an argument

$$x' = 2 * hypot(3, -4 * y)$$

Eventually, to 'correct' such expressions and make them right for the Infix parser, two fixes were made up as the third case was handled correctly by the library. For the first case, the minus is replaced by

$$(0-1)*$$

so the first expression would become

$$x' = (0-1) * 2 * x$$

For the second case, the parenthesis and the minus are replaced by

$$(0-$$

so the other two expressions would become

$$x' = (0-2)*x,\ x' = (0-2*x)$$

Attention was also given to the case where a negative constant should replace a symbol in an expression. If the constant is negative, its minus will make the expression faulty for the Infix parser. For example having this

$$x' = 2 + a + x,\ a = -2$$

its replacement would be

$$x' = 2 + -2 + x$$

The fix for that is to wrap the constant and the minus inside parentheses if it is negative and then replace it in the expression to become

$$x' = 2 + (-2) + x$$

which is handled as described before. Some other minor difficulties were caused by unforeseen relationships between the ordinary functions. When an ordinary function depends on another ordinary function like x and y

$$z' = z + 1$$

$$x = z + 1$$

$$y = x + 1$$

and the y is examined first, the value of x will be undefined. To overcome this, if a value isn't ready, this function goes to the tail of the queue of ordinary functions in order to be examined later. This can't cause an infinite loop because cyclical dependencies are impossible with ordinary functions as they depend on the present values. The constant replacement in the expressions proved to be troublesome too. Having the following expression and trying to replace a

$$x = a + ab + x + 1,\ a = 1$$

will also replace the first letter of the constant ab. To fix this, first all the symbols inside an expression were replaced by a unique string of characters that are almost impossible to be inside a symbol name. These strings were chosen to be sequences of '@'. The above expression would become

$$x = @ + @@ + x + 1$$

Then the replacement of the corresponding constants begins but in reverse order of the unique string size. To explain, if we tried to replace $@$ with 1 in the previous example we would get

$$x = 1 + 1@ + x + 1$$

which is obviously wrong. So, we have to start from the biggest sequence of $@$ and then move on to smaller ones. This was later improved by creating a regex that matched the unique string exactly without needing to start replacing in reverse order. Now, remember that the differential equations are distinguished from the ordinary ones from the presence of a comma in the expression according to the chosen convention. What was not taken into account was that commas exist in the function parameter lists too

$$x' = hypot(2,5)$$

So the convention of the differential equation representation was changed to

$$name' = \exp ression \# initial\ \ value$$

in order to avoid many code changes if we wanted to keep the previous one. In addition, another fix was to insert whitespaces after the division operator, if a decimal number starting with zero followed, because it was perceived as a division with zero by the Infix library.

Appendix G. The effects of the import of Infix

The Infix library was imported after the parallel methods using parallelism across time were created. With this import some method design choices had to be reconsidered. A function created by Infix from an expression is not called like normal Clojure functions do. In common Clojure code the arguments are written after the function name. In order to use a function created by Infix, after the function name only a map has to be provided, with keys the parameter names and values the corresponding values.

An arising benefit is that there is no need to keep track of the parameters positions in the parameter list as this is handled by the library. On the other hand, due to the special way of calling functions this library has, major overhead was introduced. As a result, it would be unfair to compare the parallel methods implementing this feature with the serial one which doesn't and so a new serial method was created that also had this overhead. For proof, below are the results of the two serial methods running the test system ts1.

Table G.1: Comparing the performance of the serial methods

iterations	1.000	1.0000	100.000
Serial method	33,51 ms	265,09 ms	3.722,50 ms
Serial method (with Infix)	91,94 ms	938,64 ms	9.428,03 ms

The performance difference is evident, as the old serial is at least three times faster than the one using Infix.

Appendix H. JVisualVM

The JVisualVM is a monitoring and profiling tool for java applications. Its profiler and thread monitoring capabilities were used for this work. It can be downloaded at https://visualvm.github.io/download.html.

When JVisualVm starts the following window appears. On the 'Applications' tab under the 'local' branch are the applications running on the jvm on the current system.

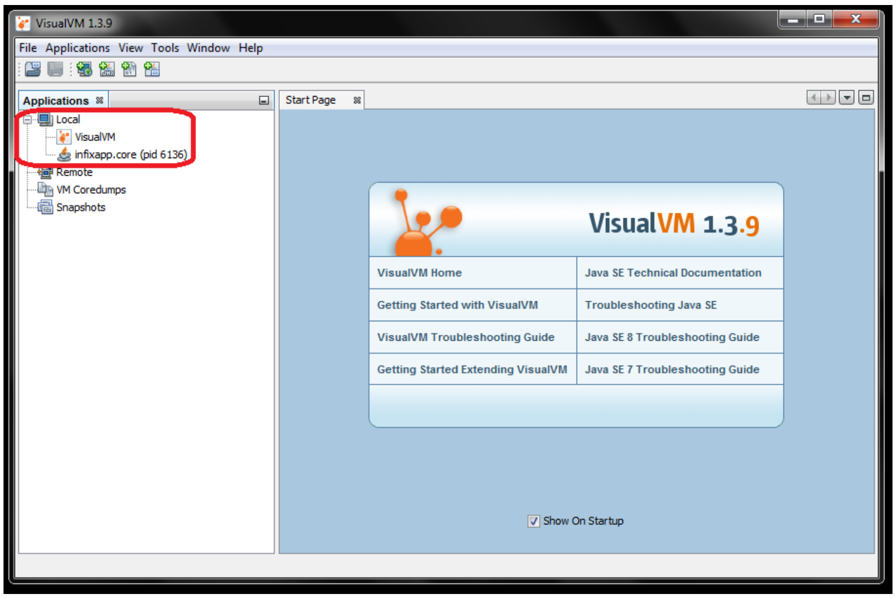
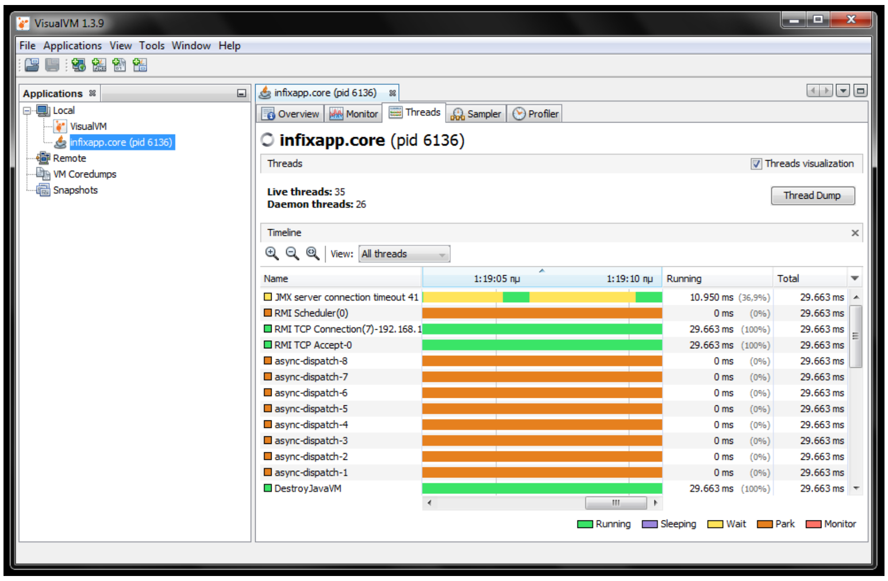

VisualVM 1.3.9
File Applications View Tools Window Help
Applications
Local
VisualVM
infixapp.core (pid 6136)
Remote
VM Coredumps
Snapshots
Start Page
VisualVM 1.3.9
VisualVM Home
Java SE Technical Documentation
Getting Started with VisualVM
Troubleshooting Java SE
VisualVM Troubleshooting Guide
Java SE 8 Troubleshooting Guide
Getting Started Extending VisualVM
Java SE 7 Troubleshooting Guide
Show On Startup

Figure H.1: The welcome screen of JVisualVM.

After choosing an application, several new tabs are created on the right side of the window. The 'Threads' and 'Sampler' tabs were used primarily in this work. In the 'Threads' tab, there is a visual representation of the threads activity.



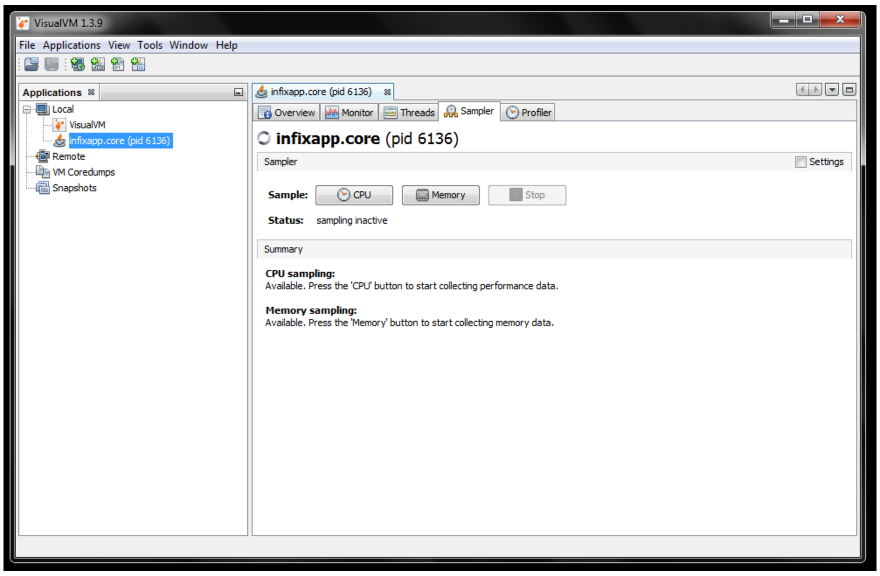
infixapp.core (pid 6136)

Live threads: 35
Daemon threads: 26

Name	Running	Total
JMX server connection timeout 41	10.950 ms (36,9%)	29.663 ms
RMI Scheduler(0)	0 ms (0%)	29.663 ms
RMI TCP Connection(7)-192.168.1	29.663 ms (100%)	29.663 ms
RMI TCP Accept-0	29.663 ms (100%)	29.663 ms
async-dispatch-8	0 ms (0%)	29.663 ms
async-dispatch-7	0 ms (0%)	29.663 ms
async-dispatch-6	0 ms (0%)	29.663 ms
async-dispatch-5	0 ms (0%)	29.663 ms
async-dispatch-4	0 ms (0%)	29.663 ms
async-dispatch-3	0 ms (0%)	29.663 ms
async-dispatch-2	0 ms (0%)	29.663 ms
async-dispatch-1	0 ms (0%)	29.663 ms
DestroyJavaVM	29.663 ms (100%)	29.663 ms

Figure H.2: Monitoring thread activity.

On the ‘Sampler’ tab there is a choice to profile the CPU or the memory used.



infixapp.core (pid 6136)

Sample:

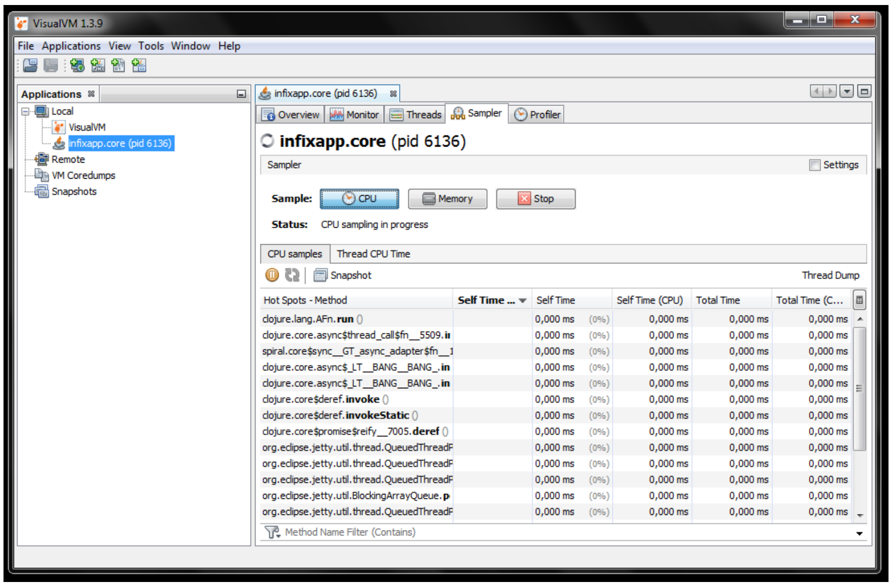
Status: sampling inactive

CPU sampling:
Available. Press the 'CPU' button to start collecting performance data.

Memory sampling:
Available. Press the 'Memory' button to start collecting memory data.

Figure H.3: Selecting to profile the CPU or memory.

By choosing the CPU, information about how much time every invoked method takes is displayed.



infixapp.core (pid 6136)

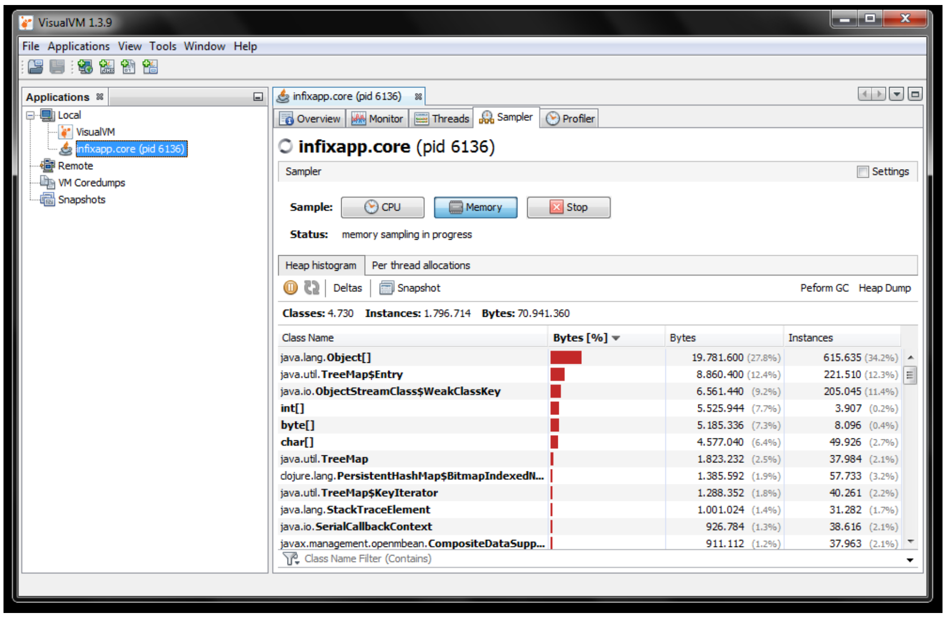
Sample:

Status: CPU sampling in progress

Hot Spots - Method	**Self Time ...**	Self Time		Self Time (CPU)	Total Time	Total Time (C...
clojure.lang.AFn.**run** ()		0,000 ms	(0%)	0,000 ms	0,000 ms	0,000 ms
clojure.core.async$thread_call$fn__5509.**ir**		0,000 ms	(0%)	0,000 ms	0,000 ms	0,000 ms
spiral.core$sync__GT_async_adapter$fn__1		0,000 ms	(0%)	0,000 ms	0,000 ms	0,000 ms
clojure.core.async$_LT__BANG__BANG_.**in**		0,000 ms	(0%)	0,000 ms	0,000 ms	0,000 ms
clojure.core.async$_LT__BANG__BANG_.**in**		0,000 ms	(0%)	0,000 ms	0,000 ms	0,000 ms
clojure.core$deref.**invoke** ()		0,000 ms	(0%)	0,000 ms	0,000 ms	0,000 ms
clojure.core$deref.**invokeStatic** ()		0,000 ms	(0%)	0,000 ms	0,000 ms	0,000 ms
clojure.core$promise$reify__7005.**deref** ()		0,000 ms	(0%)	0,000 ms	0,000 ms	0,000 ms
org.eclipse.jetty.util.thread.QueuedThreadF		0,000 ms	(0%)	0,000 ms	0,000 ms	0,000 ms
org.eclipse.jetty.util.thread.QueuedThreadF		0,000 ms	(0%)	0,000 ms	0,000 ms	0,000 ms
org.eclipse.jetty.util.thread.QueuedThreadF		0,000 ms	(0%)	0,000 ms	0,000 ms	0,000 ms
org.eclipse.jetty.util.BlockingArrayQueue.**p**		0,000 ms	(0%)	0,000 ms	0,000 ms	0,000 ms
org.eclipse.jetty.util.thread.QueuedThreadF		0,000 ms	(0%)	0,000 ms	0,000 ms	0,000 ms

Figure H.4: Profiling the CPU.

Selecting to profile the memory will display how many bytes the program entities require and how many instances of them exist.



infixapp.core (pid 6136)

Sampler

Sample:

Status: memory sampling in progress

Classes: 4.730 **Instances:** 1.796.714 **Bytes:** 70.941.360

Class Name	Bytes [%]	Bytes	Instances
java.lang.**Object[]**		19.781.600 (27.8%)	615.635 (34.2%)
java.util.**TreeMap$Entry**		8.860.400 (12.4%)	221.510 (12.3%)
java.io.**ObjectStreamClass$WeakClassKey**		6.561.440 (9.2%)	205.045 (11.4%)
int[]		5.525.944 (7.7%)	3.907 (0.2%)
byte[]		5.185.336 (7.3%)	8.096 (0.4%)
char[]		4.577.040 (6.4%)	49.926 (2.7%)
java.util.**TreeMap**		1.823.232 (2.5%)	37.984 (2.1%)
clojure.lang.**PersistentHashMap$BitmapIndexedN...**		1.385.592 (1.9%)	57.733 (3.2%)
java.util.**TreeMap$KeyIterator**		1.288.352 (1.8%)	40.261 (2.2%)
java.lang.**StackTraceElement**		1.001.024 (1.4%)	31.282 (1.7%)
java.io.**SerialCallbackContext**		926.784 (1.3%)	38.616 (2.1%)
javax.management.openmbean.**CompositeDataSupp...**		911.112 (1.2%)	37.963 (2.1%)

Figure H.5: Profiling the memory.

There is also another way to profile an application, regarding the CPU, by heading to the 'Profiler' tab. The difference between this and the sampler lies on the way the profiling is performed [66]. The first one (in tab 'Profiler'), adds extra bytecode to every method in order to record when it is called. The second (in tab 'Sampler'), takes a dump of all of the threads of execution on a fairly regular basis, and uses this to work out how roughly how much CPU time each method spends. The problem with the first type of profiling is that it adds a constant amount of extra execution time to every method call, while the second takes a more or less constant amount of time each second to record stack traces for each thread. The disadvantage of the second type of profiling is that the number of invocations recorded is not necessarily accurate, since a short method could easily start and finish between stack dumps.

Appendix I. Thesis code

The code of the methods presented here, including the test systems and the web application can be found in http://gitlab.logismi.co/chripyli/Thesis_Systems_Dynamic_Modeling_with_Clojure.