Acta Univ. Sapientiae, Informatica, **16, 2** (2024) 139–159

DOI: 10.47745/ausi-2024-0009

Optimising the Force-Directed Layout Generation

Mátyás Komáromi
ELTE, Eötvös Loránd University
Budapest, Hungary
makom789@gmail.com

István Bozó
ELTE, Eötvös Loránd University
Budapest, Hungary
bozo_i@inf.elte.hu
0000-0001-5145-9688

Melinda Tóth
ELTE, Eötvös Loránd University
Budapest, Hungary
toth_m@inf.elte.hu
0000-0001-6300-7945

Abstract. A graph visualisation tool can be invaluable in code comprehension. It is a well-known and researched field of graphical informatics. Several good algorithms were developed, but most of the graph drawing tools mainly focus on the generation of static drawing. In this paper, we present an approach to force-directed layout generation that is orders of magnitudes faster than the trivial implementation. This technique is based on the Runge-Kutta methods and is efficient enough to visualise the user-requested parts (views) quickly for relatively large Semantic Program Graphs of Erlang projects in soft real-time. Such a graph might assist code comprehension in the RefactorErl framework even better.

Key words and phrases: RefactorErl, Parallel computation, Graph drawing, Erlang, GPU programming.

1 Introduction

Tool-supported software development is an accepted and desirable part of the software development lifecycle. Several static source code analyser tools exist and aim to help code comprehension by presenting the analysed data in various ways. Taking the size of the presented data into account, a focused graph representation of the required data is one of the most useful.

The tool RefactorErl [1] is a static source code analyser and transformer tool for Erlang. It aims to present source code dependencies and help in code comprehension tasks with different graphs. However, the size of these graphs represented in a static format for industrial-scale software goes beyond the limits a human can comprehend. To overcome this, we needed to define new dynamic graph views for the users of RefactorErl.

Graph visualisation is a well-known and researched field of graphical informatics. Several good algorithms were developed and reviewed by our days [2]. However, most of the graph drawing tools mainly focus on the generation of static drawing. Our goal is to create a tool to support dynamic views and provide an efficient layout generation.

In this paper, we present an approach to a force-directed layout [3] generation based on the Runge-Kutta methods. This method is efficient enough to quickly visualise the user-requested parts (views) of relatively large Semantic Program Graphs of Erlang projects in soft real-time. We studied different parallelisation of the method on GPUs to achieve a better performance.

The rest of the paper is structured as follows. In Section 2 we introduce the RefactorErl tool and our first dynamic graph visualisation prototype, Gview. Sections 3 and 4 present our motivation to use force-directed layout generation, and demonstrate our solutions for efficient parallelisation and improvements of the algorithms. In Section 5 we evaluate our results. Sections 6 and 7 describe related works and conclude the paper.

2 Background

RefactorErl [1] is an open-source static source code analyser and refactoring tool for Erlang, developed by the Department of Programming Languages and Compilers at the Faculty of Informatics, Eötvös Loránd University, Budapest, Hungary. The phrase ”refactoring” [4] means a semantic preserving source code transformation, so a structural change is performed in the program while it does not alter its original behaviour. Erlang is a dynamically typed functional programming language, thus to gather all of the necessary information for a behaviour-preserving transformation is not straightforward. The RefactorErl comes with an easily extensible lexer and parser. The framework of RefactorErl applies several static semantic analyses on the source code and represents the source code and the gathered information in a Semantic Program Graph [5].

The main focus of RefactorErl is to support the daily code comprehension tasks of Erlang developers. Among the features of RefactorErl is included a user-level Semantic Query Language, that can assist Erlang developers in everyday tasks such as program comprehension, debugging, finding relationships among program components, etc. The queries are mapped to traversals in the Semantic Program graph. For industrial-scale software, the size of this graph can become incredibly huge. Therefore, the processing of a query may take from a few seconds up to several hours depending on its complexity.

2.1 The Semantic Program Graph

RefactorErl keeps the information extracted through static semantic analyses in a special data structure called the Semantic Program Graph (SPG) [5]. This graph represents the lexical, syntactic and semantic structure of the analysed program. Typically the lexical, syntactic and semantic elements of a program map to one node in the SPG, while the connection between these elements maps to tagged edges between these nodes. The SPG is a rooted graph. The root is a special node which does not represent a program unit. The role of this root node is to be the common ancestor of nodes not having one naturally. Care must be taken when traversing the SPG as it is not a tree and may contain directed loops.

Although it is not a tree, the SPG exposes hierarchical properties as well. As an example taking the subgraph of SPG representing the syntactic data of the program, we find the nodes of files right below the root node. Below the files, we find function forms. Going one level deeper we have clauses that build up the previously mentioned forms. Finally, on one level deeper there are the symbols of clause names, parameters of clauses and syntactic trees of expressions in the rows of bodies of clauses.

2.2 Code comprehension

Nowadays code comprehension or program comprehension is an increasing duty of IDEs and other tools for software development and maintenance. Be the user of such a tool a newcomer to the task, an employee transferring from one project to another, a project manager or a team picking up a new piece of technology, understanding the code base and getting familiar with it leads to a much higher efficiency on both building functional and stable software and keeping such systems running.

It is common knowledge that humans can process much more information visually than through text or audio in a short amount of time. RefactorErl already has methods for supporting code comprehension. Therefore, our goal is to extend these features with a tool (Gview) for dynamically visualising parts of the SPG (so-called views) in soft real-time, through which the user can explore aspects of this huge graph and learn about the project at hand. In our previous work [6], we studied the structure of such a tool and the way the data can be transferred from RefactorErl to Gview. We also examined the different aspects of the rendering environment.

In this paper, we investigate one of the popular graph drawing methods: the force-directed layout generation method. Particularly the probe of algorithmic and computational optimisation through the usage of higher-order methods and the more efficient usage of the GPU is the target of this work.

2.3 Euler's method

As we will see in Section 3.1, the problem of generating the force-directed layout of a directed or undirected graph can be formulated as simulating a physical system over time and running this simulation until the layout is sufficiently close to a fixed point. This physical system, as shown below, is described by a set of differential equations, which ought to hold throughout the entire lifespan of the system. To ensure convergence, the simplest method that can be applied in this context is Euler's method [7].

The essence of Euler's method in this context is to start the simulation from an initial layout of the graph (random or generated by other means), and then take discrete steps. In each step the algorithm calculates the derivative of the approximated function, which can be interpreted as accumulating the acting forces on each body of the system and letting the simulation run for a given h constant amount of time and use the resulting layout as the next best setup, continuing the loop.

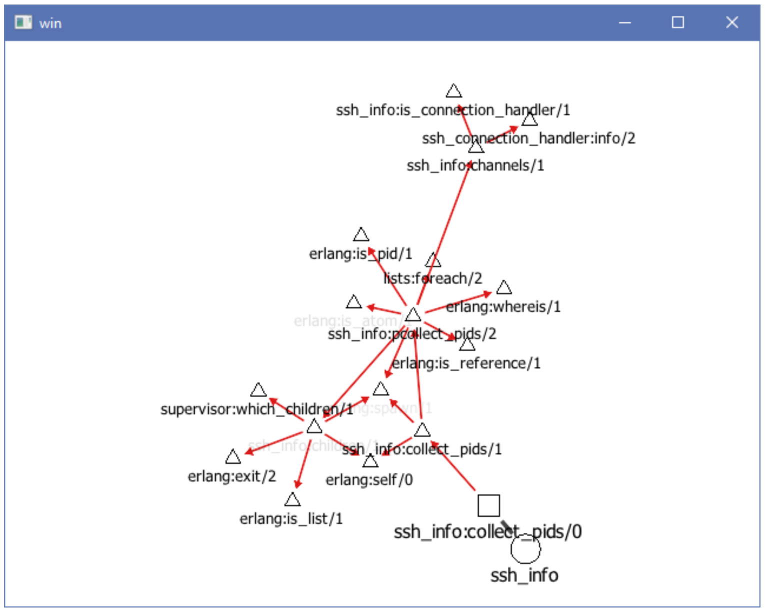
2.4 Room for improvements

The dilemma of Euler's method is how to choose h. When choosing a too-small value, the simulation may take ages. On the other hand, too large values will lead to oscillations and the potential lack of convergence. Our previous approach was to decrease h over time to ensure convergence. However, the optimal h may not only depend on the arrangement of the graph but on the currently approximated optimal layout. To address both issues, in this paper, we inspect the adoption of a well-known generalisation of the above method: the adaptive Runge-Kutta method family [8].

3 Motivation

Our goal was to create a tool for dynamically and interactively displaying parts of the SPG in RefactorErl to aid code comprehension. To this end, we want to research the possibilities of speeding up the above-mentioned algorithm. One of such plots by Gview can be seen in Figure 1.

Figure 1: Function call view plotted using the researched algorithm.



3.1 The force-directed layout generation

The method of force-directed layout [9] generation is a way of generating a two-dimensional layout for directed and undirected graphs alike.

The core concept of creating such a layout involves fitting a physical system on the graph, in the following way. Take a graph $G = (V, E)$ with weighted vertices V and weighted edges E, totalling n vertices, weight functions m and l! Each node of the n nodes corresponds with a body in the system with a position denoted by $p_i(t)$, a constant charge, m_i, and a velocity $v_i(t)$. As the position and velocity depend on the time passed since the start of the simulation, p_i and v_i have the type of $\mathbb{R} \to \mathbb{R}^2$. The generation of force-directed layout for graph G starts with calculating an initial, usually random, layout of the nodes, denoted by $p_i(0)$ for $i = 1..n$. After the initial layout has been set up, the algorithm follows by simulating the evolution of the physical system using the following equations.

$$v_i(t) = \sum_{j=1, i \neq j}^{n} e(i,j,t)) \tag{1}$$

$$e(i,j,t) = \frac{d(i,j,t)}{\|d(i,j,t)\|_2} * \left(H * ln(\|d(i,j,t)\|_2^2) * l_{i,j} - \frac{G * m_i * m_j)}{\|d(i,j,t)\|_2^2} \right) \tag{2}$$

$$d(i,j,t) = p(t)_i - p(t)_j \tag{3}$$

$$v = \frac{\partial p}{\partial t} \tag{4}$$

Equation 1 means that at any given time point t the velocity of the ith body equals the sum of the forces exerted by other bodies. Here we use the term force, to describe instantaneous forces, which have a direct effect on the velocity of the bodies rather than the acceleration. Such forces are characterised by Equation 2: knowing the position of the ith and jth body at time t, we can easily calculate the force acting on body i at time t.

Here H and G are arbitrary positive constants, used to regulate the strength of the two types of acting force. In our research having $H = 2$ and $G = 6100$ turned up the best-looking results. The most important equation is 4, which describes the analytic connection between position and velocity in the physical system. It can be used to rewrite the former equation system as a differential equation with the common vector function of positions $p = t \longrightarrow (p(t)_1, p(t)_2, ..., p(t)_n)$ as the unknown, as follows.

$$\frac{\partial p(t_{cur})}{\partial t} = f(t_{cur}, p(t_{cur})) \tag{5}$$

$$p(t_0) = p(0) = p_0 \tag{6}$$

$$f(t,p) = \sum_{j=1, i \neq j}^{n} e(i,j,t) \tag{7}$$

The canonical form of the ordinary differential equation is given in Equation 5 with the definition of f in Equation 7, while the initial position requirement is defined in Equation 6.

Therefore, our goal is to approximate the vector function p for increasing values of t until we get close enough to its fixed point. For this purpose, we want to use higher-order methods, to better utilise computational power and stable convergence as t approaches ∞.

3.2 Higher order methods

The Runge–Kutta methods [10] are a family of explicit iterative methods, used in temporal discretization for the approximate solutions of ordinary differential equations. The methods include the well-known routine called the Euler Method and can be considered as the generalisation of the routine. The method has an adaptive version, which can adjust the step size of the simulation and thus keep the error below a given value, ϵ.

These methods are called a family for the reason that they depend on numerous parameters: a, b, and c. The latter two are vectors and the former one is a matrix [8]. These parameters can be arranged in a so-called Butcher tableau as can be seen in Figure 2.

Figure 2: Example of an extended explicit Butcher tableau of degree s

$$\begin{array}{c|ccccc}
0 & & & & & \\
c_2 & a_{21} & & & & \\
c_3 & a_{31} & a_{32} & & & \\
\vdots & \vdots & & \ddots & & \\
c_s & a_{s1} & a_{s2} & \cdots & a_{s,s-1} & \\
\hline
 & b_1 & b_2 & \cdots & b_{s-1} & b_s \\
 & b^*_1 & b^*_2 & \cdots & b^*_{s-1} & b^*_s
\end{array}$$

With the initial positions given in p_0, the method proceeds to create further approximations, $p(t_{i+1})$, from the previous one, $p(t_i)$ for $i = 1..\infty$, according to Equations 8 and 9. The difference in the result of these two equations is used to approximate the error introduced by taking a step of length h. Using this calculated error term, we can choose a new step size to decrease the error below ϵ or allow larger steps in exchange for a larger error term.

$$p_{i+1} = p_i + \sum_{j=1}^{s} b_j k_j \tag{8}$$

$$p^*_{i+1} = p_i + \sum_{j=1}^{s} b^*_j k_j \tag{9}$$

where

$$k_j = f\big(t_i + c_j * h, p_i + h * \sum_{l=1}^{j-1} a_{j,l} k_l\big) \tag{10}$$

The exact steps of choosing the next h and the very mathematics behind this algorithm is a well-known topic and has been discussed in many papers. Our goal is to apply this method to the problem at hand and to optimise it for the parallel architecture of modern GPUs.

As we can see in Equation 7, our f does not depend on the p parameter directly. The indirect dependence through d is replaced by the previous best approximation p_i for k_1 and $p_i + h * c_{j+1} * k_j$ for k_{j+1}, thus eliminating the need for the matrix a of the RK method. This property of the simulation resulted from the fact that we do not employ friction, which would depend on the velocity of the bodies but rather use instantaneous forces.

4 Methodology

Our goal is to find ways to improve the performance of the force-directed layout generation by utilising the massively parallel architecture of modern GPUs.

The final form of the equation that we are using, taking into consideration the relevant properties of the physical system, described at the end of Section 3.2, is Equation 11. In each step of the simulation, we have to calculate the k coefficients for each of the n bodies. Since the dimension of k is s and evaluating f requires $O(n)$ operations, the total cost of advancing one step comes out to be $O(sn^2)$.

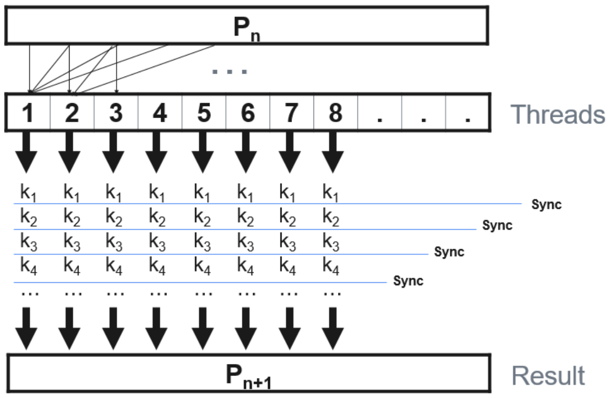
$$k_j = f\big(t_i + c_j * h\big) = \sum_{b=1, a\neq b}^{n} e(a, b, t) \tag{11}$$

4.1 Linear parallelisation

The first idea for parallelisation that one should consider is simply assigning the task of evaluating the exerted forces of all other bodies to a single body. Thread i allocates a single two-dimensional vector v, loops through all the bodies in the system and calculates the force exerted on body i through body j at the current time point t_n[1]. The calculated forces are accumulated on the fly into the local variable v of the thread and the result is stored in the k_1 array. The k_{j+1} approximations are generated in a similar manner, however p_i is replaced by $p_i + h * c_{j+1} * k_j$. A schematic representation of the linear parallelisation method can be seen in Figure 3.

[1]with exception of the i^{th} body

Figure 3: Architecture of basic parallelisation technique



The above-mentioned dependence of k_{j+1} on k_j results in the need for global synchronisation of all the employed threads to make the calculated k_j values visible to the other threads. This explicit synchronisation can only be realised on the GPU if the number of invocations is below certain driver-defined limits, which can be queried using the OpenGL command and is usually at least 1024.

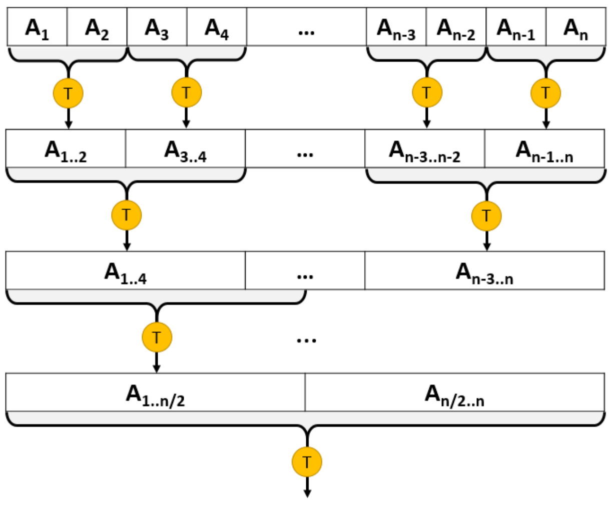
In case of having a larger amount of bodies in the system than the hardware exposed limit, we need CPU synchronisation, which means splitting the calculation of each k_j vector into different dispatches.

The performance bottleneck, however, comes from the fact that the amount of work each thread is doing is proportional to sn which can grow too large. The OpenGL standard guarantees [11] the ability to dispatch at least a maximum of 2^{16} workgroups, all of which may consist of a maximum of at least 1024 work items (threads). This means that reducing the number of threads dispatched from n can potentially result in a great increase in performance. This idea is further supported by the fact that GPU cores are much less powerful than CPU cores and thus employing more of them can lead to better resource utilisation, which gives reason to the optimisation in Section 4.2.

4.2 Refining work per thread

To better utilise the parallel architecture of modern GPUs for the problem at hand, we want to dispatch more than $O(n)$ threads. Thus we take Equation 11, and for each (a, b) pair, we create an invocation, totalling in $n(n-1)$ threads. After calculating each $e(a, b, t)$ value in the summation on Equation 11, however, we need to evaluate the actual sum of these two-dimensional vectors and this is where parallel reduction comes in. Reduction (or folding) is the generalisation of summation: for a given A array of size n and a binary combining function f, the result of the folding expression is $b = f(A_1, f(A_2, f(A_3, \dots)))$ where the … goes until n. Parallelising a reduction is not a trivial task and is a well-known candidate for optimisation [12]. In our example, we have the benefit that our combination function is associative and commutative. Thus enabling us to employ a divide and conquer technique as shown in Figure 4.

Figure 4: Basic idea of divide and conquer strategy used in the parallel reduction. The circles with T represent threads of execution.

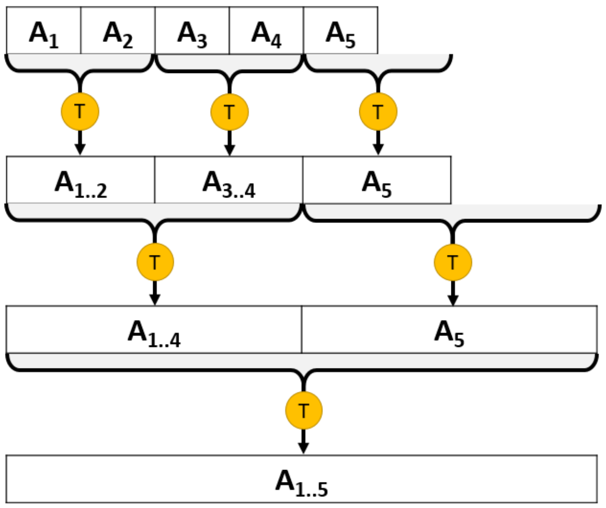


In the presented approach, we divide the reduction into levels of reduction, in each level, the size of the array that is to be combined is decreased by half. In each level, one thread only has to combine two elements of the array of the current level, which would imply that the work per thread has changed to be $O(1)$. However, by noting that the levels must come in increasing order, each one depending on the previous one, after reusing the allocated threads through levels, the total work per thread totals $O(log_2(n))$.

Figure 4 shows an optimal scenario, with n being a power of two, if n is not a power of the amount of work one thread is responsible for, then the last thread may index out of the array. To avoid this, we need to employ bound checking. An example of the size of 5 can be seen in Figure 5. This bound checking may induce a maximum of $O(n)$ extra work.

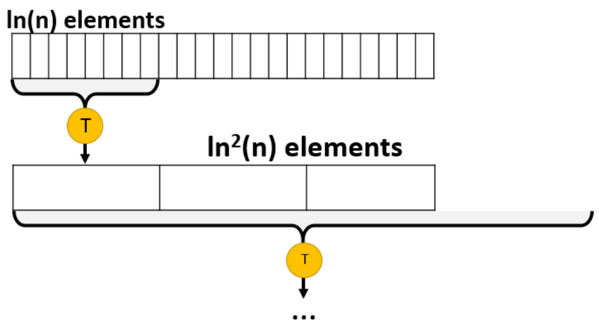
The theoretical optimum of giving one thread $O(ln(n))$ work can also be achieved.

Figure 5: Example of the maximum amount of extra work introduced by not a power of two n, for $n = 5$.



However, it requires extra mathematics behind the indexing and bound checking, and also the potential extra work growth to $O(nln(n))$. A visual representation is shown in Figure 6.

Figure 6: Optimal $ln(n)$ work per thread.

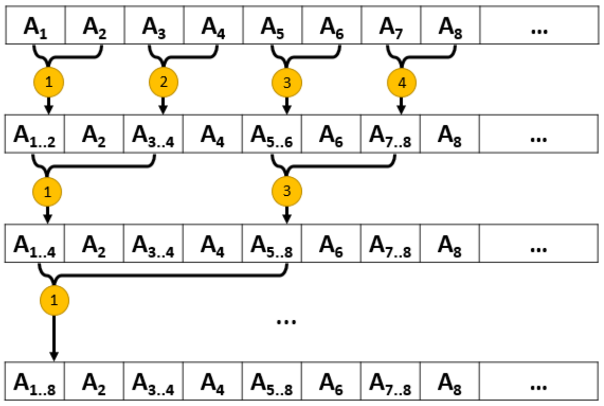


4.3 Memory usage and synchronization

When programming for a modern GPU, it is possible to arrange threads of execution (say invocations) into so-called workgroups. Threads (also referred to as work items in this context) in a workgroup can share group local memory and can synchronise group-wise without the need for CPU intervention.

Because the levels of reduction are calculated linearly, we can store the partial results in place, which thanks to workgroups, can be synchronised on the GPU. This in-place storing of partial results can be seen in Figure 7. Thanks to this technique, we only need $O(n)$ memory and need to transfer one element to the CPU each frame.

Figure 7: In-place memory usage of the parallel reduction.


A1 A2 A3 A4 A5 A6 A7 A8 ...
1 2 3 4
A1..2 A2 A3..4 A4 A5..6 A6 A7..8 A8 ...
1 3
A1..4 A2 A3..4 A4 A5..8 A6 A7..8 A8 ...
1
...
A1..8 A2 A3..4 A4 A5..8 A6 A7..8 A8 ...

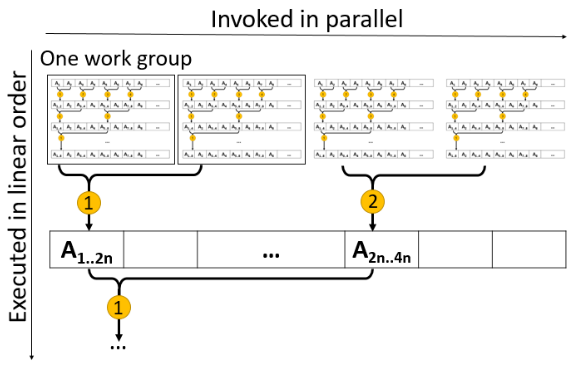
However, to be able to utilise the local synchronisation of the GPU workgroups, a workgroup size equal to the number of bodies is needed, which brings us back to our previous problem. To solve this, we investigated how we could organize the parallel reduction into batches of maximal size, and recursively apply the already presented reduction method. As Figure 8 shows, by creating partial results of maximal size, using multiple workgroups and then applying a global synchronisation, we can take advantage of the parallel architecture.

5 Results

The researched techniques were tested on a laptop with 5^{th} generation Intel Core i5 processor, 8GB of memory, using Intel HD 6000, running OpenGL 4.5. Parallel GPU programs were realised with GLSL version 430 [13].

In our tests, we incrementally generated square grids of increasing sizes from 1x1 to 11x11, with random starting positions and ran first the CPU, then the GPU and refined GPU algorithms on the same starting points, around 100 times each. As these measurements may vary according to environment properties such as the OS scheduler or extra load from updates and scheduled cleaning etc, we ignore the

Figure 8: Dividing the parallel reduction into smaller tasks that can be handled by one workgroup.



highest and lowest 5% of data and perform a normal distribution fitting on the rest. The resulting expected value of generation time is plotted against the number of nodes in Figure 9 and Figure 10.

Figure 9: The huge difference of CPU and GPU algorithm, note the logarithmic scale! Tested on grid graphs.

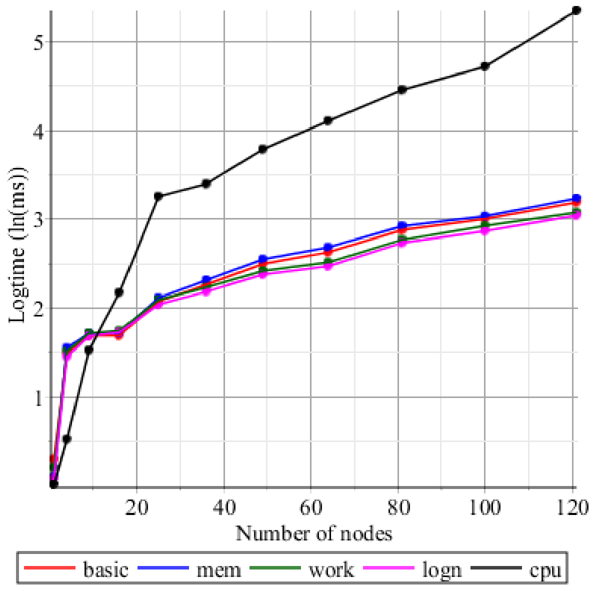
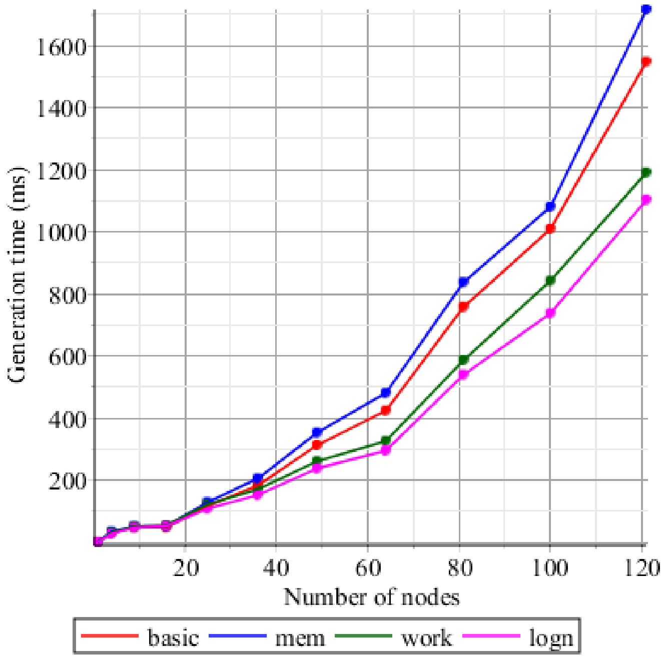


Figure 9 clearly shows the enormous improvement of the GPU parallel algorithm, even on the integrated card used in testing. We expect that with a higher-tier dedicated card, this gap is to increase further.

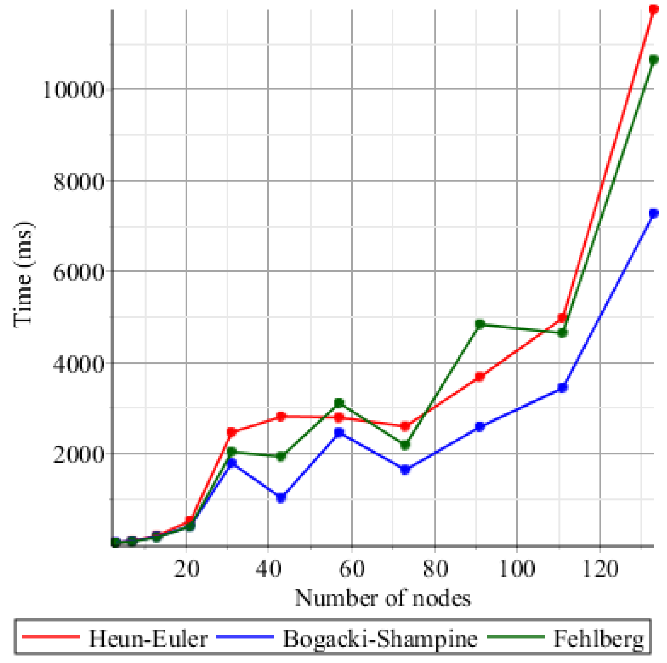
Figure 10: Comparison of the refined GPU algorithms, tested on different-sized grid graphs.



The comparison presented in Figure 10 shows how different techniques described in Section 4 improve generation time for increasing sizes of grids. The interesting thing to note is that the memory (but not workload) optimised method performs poorer than the trivial approach. This can be due to the hardware memory locality of Intel integrated GPUs, which implies that the extra copy operations introduced by memory optimisation by hand have a higher toll on performance than the improvements in the locality it creates. Thus on a dedicated card, the memory-optimised version might improve performance considerably.

We also investigated the performance of different realisations of the Runge-Kutta family: the Heun-Euler method, the less famous Bogacki-Shampine method and a high-order Fehlberg method. The tests were performed on a tree graph with nodes ranging from 3 to 133 and the same refining techniques were applied as mentioned previously. The results of this comparison can be seen in Figure 11. One can see how the advantage of being able to take larger steps turns into a disadvantage of higher required work per step. Based on these measurements, we can conclude that the Bogacki-Shampine method is most efficient for our problem.

Figure 11: Comparison of different RK methods, tested on varying-sized tree graphs.



5.1 Usage in RefactorErl

Figures 12 and 13 demonstrate a generated graph about the Mnesia application and a function call graph generated by clicking on `verify_merge/1`.

6 Related work

In the following, we would like to compare our tool with some well-known graph visualisation tools.

Figure 12: View of all the modules in the Mnesia DBM.

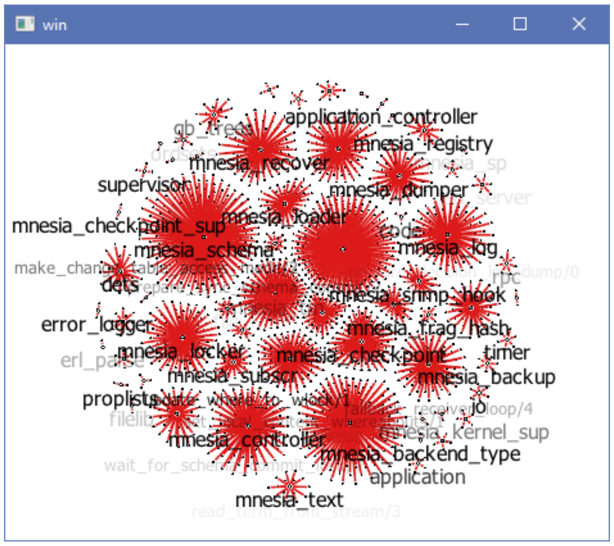
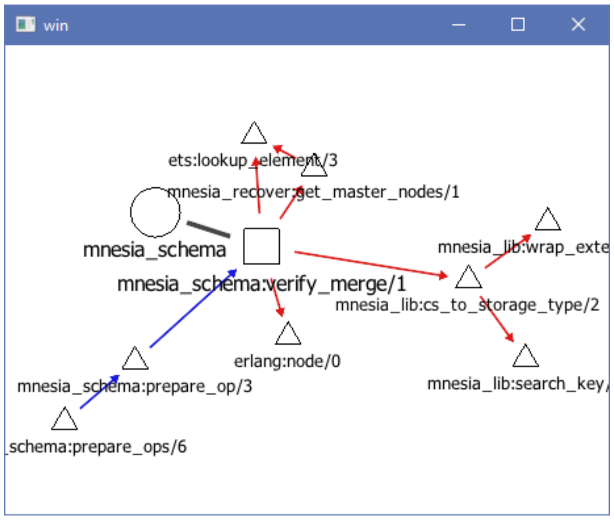


Figure 13: View generated be clicking on `verify_merge/1` in the main view of Mnesia.


win
ets:lookup_element/3
mnesia_recover:get_master_nodes/1
mnesia_schema
mnesia_lib:wrap_exte
mnesia_schema:verify_merge/1
mnesia_lib:cs_to_storage_type/2
erlang:node/0
mnesia_schema:prepare_op/3
mnesia_lib:search_key
schema:prepare_ops/6

6.1 Graphviz

Graphviz [14] is an open-source graph visualisation software developed by AT&T Labs Research.

The Graphviz layout programs take descriptions of graphs in a simple text language and make diagrams in useful formats, such as images and SVG for web pages; PDF or Postscript for inclusion in other documents; or display in an interactive graph browser. It supports many layout generation algorithms, such as hierarchical or the energy-minimizing stress-majoring technique. The software package has many useful features for concrete diagrams, such as options for colours, fonts, tabular node layouts, line styles, hyperlinks, and custom shapes.

Many software use Graphviz as an intermediate tool for displaying graphs. For example, ArgoUML has an alternative UML Diagram rendering, called argouml-graphviz, ConnectedText has a Graphviz plugin, and FreeCAD uses Graphviz to display the dependencies between objects in documents. Other programs can output in DOT [15] format and thus generate drawings with Graphviz. Doxygen also uses Graphviz to generate diagrams including class hierarchies and collaboration for source code. Graphviz targets static rendering of graphs; it optimises the drawing as much as possible and thus takes considerable time on very large graphs, and also limits the interactivity between software and user.

6.2 D3.js

D3.js [16] is a JavaScript library for manipulating documents based on data. D3.js helps bring data to life using HTML, SVG, and CSS. It emphasises on web standards giving the full capabilities of modern browsers without the need of tying to a proprietary framework, combining powerful visualisation components and a data-driven approach to DOM manipulation. This library is a modern, browser-based solution to visualisation problems with countless useful features such as pie charts, hierarchical graph drawing and force-directed layout generation.

D3.js supports force-directed layout generation using velocity Verlet integration which may require a much smaller step size than the RK methods to minimize oscillations in the solution, but the method is symplectic. Thus the two methods were meant to solve different kinds of problems, as our version of the force-directed layout generation uses logarithmic springs and instantaneous forces, our simulation need not be energy conserving or symplectic for short. The key difference between our research and D3.js is that we aim to exploit the parallel architecture of modern GPUs, while the simulations of D3.js get calculated on the CPU[2].

[2]unless some JavaScript optimisation happens

6.3 Gephi

Gephi [17] is an open-source software for graph and network analysis. It uses a 3D render engine to display large networks in real time and to speed up the exploration. Gephi advertises itself as having a flexible multitasking architecture that brings new possibilities to work with complex data sets and produce informative graphics. It has been used in several research projects in academia, journalism and elsewhere. For instance, it was used in visualising the global connectivity of New York Times content and examining Twitter network traffic during social unrest along with more traditional network analysis topics.

Development of Gephi was started in the summer of 2008, while the last stable update was in 2017. It was created in the Java programming language and although it features an OpenGL renderer, it uses immediate mode rendering, which became obsolete with OpenGL 3.1 in 2009 which means Gephi does not use GPU for layout generation. Today, with OpenGL 4.6, much faster rendering tools are available, such as instanced rendering, VBOs and compute shaders. Also, it is built on top of the NetBeans IDE, which means it cannot be integrated into another project, only added as an external tool.

6.4 GoJS

GoJS [18] is a JavaScript and TypeScript library for building interactive diagrams and graphs. GoJS claims to let the user build all kinds of diagrams and graphs, ranging from simple flowcharts and org charts to highly specific industrial diagrams, SCADA and BPMN diagrams, medical diagrams like genograms, and more. The library is meant for the implementation of interactive diagrams and visualization on modern web browsers and platforms. It allows easy construction of custom and complex diagrams of nodes, links, and groups with customizable templates and layouts. It does not depend on any JavaScript libraries or frameworks, so it should work with any web framework or with no framework at all. The library focuses on interactivity and flexibility. There are many demos available online on the webpage of the tool. It also offers rich features like drag-and-drop, copy-and-paste, in-place text editing, tool-tips, templates, data binding and models, transactional state and undo management, palettes, event handlers, commands, and an extensible tool system for real-time custom operations on the diagram. It also features many automatic layout generation algorithms, which can be extended by the user of the library.

One such automatic layout is the force-directed layout generation algorithm. They describe the method as a layout generation method that treats the graph as if it were a system of physical bodies with repulsive electrical, attracting gravitational, and spring forces acting on them and between them. The engine uses the CPU for layout generation, thus it is not optimized for modern parallel GPUs.

7 Conclusion

The RefactorErl framework has several graphical and command-line interfaces, that support refactoring, static code analysis and code comprehension as well. The tool uses the so-called Semantic Program Graph as the intermediate representation of the source code which includes static semantic information beside the syntactic and lexical information. We have extended RefactorErl with Gview. Gview is an efficient and interactive graph visualisation tool, that uses force-directed layout generation. This algorithm was implemented using Euler's method which is only stable with potentially very small step sizes.

In this paper, we presented a better approach to simulating the evolution of the physical system that is the system of bodies and springs defined by graphs we want to plot. The above-described method is based on the well-known adaptive Runge-Kutta method family, a generalisation of Euler's method. We presented a trivial approach for parallelising the RK methods on the GPU and analysed this linear technique. We further investigated and measured optimisation opportunities for the GPU algorithm. These optimisations included different ways of refining the memory usage, changing the distribution of work among threads to reach a better configuration and the importance of different synchronisation functionalities.

In our future work, we plan to investigate other layout generation methods and optimise them for GPU.

Gview is open source and available on GitHub. The integration with RefactorErl will be released soon with the upcoming release of the tool:

`https://github.com/Frontier789/Gview`.

Funding: "Application Domain Specific Highly Reliable IT Solutions" project that has been implemented with the support provided from the National Research, Development, and Innovation Fund of Hungary, financed under the Thematic Excellence Programme no. 2020-4.1.1.-TKP2020 (National Challenges Subprogramme) funding scheme.

Data Availability: The study did not generate new data.

References

[1] I. Bozó, D. Horpácsi, Z. Horváth, *et al.*, "RefactorErl, Source Code Analysis and Refactoring in Erlang," in *Proceeding of the 12th Symposium on Programming Languages and Software Tools*, Tallin, Estonia, 2011 (⇒ 140).

[2] G. D. Battista, P. Eades, R. Tamassia, and I. G. Tollis, "Algorithms for drawing graphs: An annotated bibliography," *Computational Geometry: Theory and Applications*, vol. 4, no. 5, pp. 235–282, 1988 (⇒ 140).

[3] T. Kamada and S. Kawai, "An algorithm for drawing general undirected graphs," *Information Processing Letters*, vol. 31, no. 1, pp. 7–15, 1989 (⇒ 140).

[4] M. Fowler, K. Beck, J. Brant, W. Opdyke, and D. Roberts, *Refactoring: Improving the Design of Existing Code*. Addison-Wesley, 1999 (⇒ 140).

[5] Z. Horváth, L. Lövei, T. Kozsik, *et al.*, "Modeling Semantic Knowledge in Erlang for Refactoring," in *Knowledge Engineering: Principles and Techniques, Proceedings of the International Conference on Knowledge Engineering, Principles and Techniques, KEPT 2009*, ser. Studia Universitatis Babeş-Bolyai, Series Informatica, vol. 54(2009) Sp. Issue, Cluj-Napoca, Romania, Jul. 2009, pp. 7–16 (⇒ 140, 141).

[6] Mátyás Komáromi, Melinda Tóth, István Bozó, *An Efficient Graph Visualisation Framework For RefactorErl, Paper accepted into the Special Issue of Studia Universitatis Babes-Bolyai, series Mathematica, Informaticaand Physica, MACS'18, 12th Joint Conference on Mathematics and Computer Science, Cluj-Napoca, June 14-17*, 2018 (⇒ 142).

[7] K. Atkinson, *An Introduction to Numerical Analysis*. Wiley, 1989, ISBN: 9780471500230 (⇒ 142).

[8] J. Dormand and P. Prince, "A family of embedded runge-kutta formulae," *Journal of Computational and Applied Mathematics*, vol. 6, no. 1, pp. 19–26, 1980 (⇒ 142, 145).

[9] T. M. J. Fruchterman and E. M. Reingold, "Graph drawing by force-directed placement," *Software - Practice and Experience*, vol. 21, no. 11, pp. 1129–1164, 1991 (⇒ 143).

[10] C. Harper, *Introduction to mathematical physics* (Prentice-Hall physics series). Prentice-Hall, 1976, ISBN: 9780134875385 (⇒ 145).

[11] D. Shreiner, G. Sellers, J. Kessenich, and B. Licea-Kane, *OpenGL programming guide: The Official guide to learning OpenGL, version 4.3*. Addison-Wesley, 2013 (⇒ 147).

[12] M. Harris *et al.*, “Optimizing parallel reduction in cuda,” (⇒ 148).

[13] R. J. Rost, B. Licea-Kane, D. Ginsburg, *et al.*, “Opengl(r) shading language,” 2004 (⇒ 150).

[14] J. Ellson, E. Gansner, L. Koutsofios, S. C. North, and G. Woodhull, “Graphviz—open source graph drawing tools,” in *International Symposium on Graph Drawing*, Springer, 2001, pp. 483–484 (⇒ 155).

[15] E. E. Koutsofios and S. C. North, “Drawing graphs with dot,” 1991 (⇒ 155).

[16] N. Q. Zhu, *Data visualization with D3. js cookbook*. Packt Publishing Ltd, 2013 (⇒ 155).

[17] M. Bastian, S. Heymann, and M. Jacomy, “Gephi: An open source software for exploring and manipulating networks,” in *Third international AAAI conference on weblogs and social media*, 2009 (⇒ 156).

[18] F. Shahzad, T. R. Sheltami, E. M. Shakshuki, and O. Shaikh, “A review of latest web tools and libraries for state-of-the-art visualization,” *Procedia Computer Science*, vol. 98, pp. 100–106, 2016 (⇒ 156).

Received: 06.07.2024; *Revised: 09.10.2024*; *Accepted: 10.10.2024*