Inclusion and Evaluation of CPU and GPGPU Parallel Processing Backends for a Chemical Reaction Network Simulator

Sirlenon de Araujo Macedo[1], **Sarah Merigue Baraldi**[1], **Renan Albuquerque Marks**[1]

[1]Faculdade de Computação – Universidade Federal de Mato Grosso do Sul (UFMS)
Campo Grande – MS – Brasil

{sirlenon.araujo,sarah.baraldi,renan.marks}@ufms.br

Abstract. *This paper presents the incorporation and evaluation of two new backends for the DNAr molecular computing simulator. The tool was extended to support dynamic selection among the original R backend* `deSolve`*, a Julia implementation* `DifferentialEquations.jl` *with multicore parallelism, and a Python/CUDA backend* `ginSODA` *for GPU execution. Results across twelve benchmarks show that the Julia backend reduces execution time by up to 67.7% in complex simulations while preserving high numerical accuracy, which remains superior to 98.5% in all cases. The analysis highlights that multithreading and GPU acceleration efficiency depend directly on circuit scale to amortize initialization overheads.*

1. Introduction

Since their creation, computers have played a fundamental role in the evolution of society, being essential tools for processing, storing, and analyzing information.

The evolution of computers is deeply linked to Moore's Law, an observation formulated by Gordon Moore, co-founder of Intel, in 1965 [Moore 1965]. He predicted that the number of transistors in an integrated circuit would double approximately every two years, resulting in exponential increases in processing power and reductions in the cost per transistor. This trend has driven the growth of the computing industry and enabled the creation of faster, more compact, and more affordable devices.

However, as transistors approach nanometer sizes, significant physical challenges arise, such as overheating, increased power consumption, and limits to miniaturization [Bohr 2009]. These limitations have led researchers to explore new approaches, such as quantum computing [Arute et al. 2019], neuromorphic computing [Schuman et al. 2022], photonic computing [Shen et al. 2017], parallel computing [Asanović et al. 2006], field-coupled nanocomputing [Anderson and Bhanja 2014], molecular computing [Lent et al. 2016], and others, in addition to the development of new architectures that optimize performance without compromising energy efficiency [Hennessy and Patterson 2011].

In this scenario, DNA computing has emerged as a promising area, being one of the branches of molecular computing, which exploits the chemical properties of DNA to perform computational operations [Qian and Winfree 2011]. Unlike traditional computers based on silicon and transistors, this approach uses synthesized DNA

molecules to process and store information, paving the way for new applications and computational paradigms.

This paper investigates performance improvements in Chemical Reaction Network simulations using modern CPU and GPU architectures, incorporating two backends into DNAr [Vieira et al. 2020]. These backends aim to improve simulation efficiency while maintaining numerical accuracy: one implementation uses Julia `DifferentialEquations.jl`, designed for high-performance scientific computing with multicore support, and the other is based on Python `ginSODA` for GPU execution.

2. Related Works

Some tools and simulators were developed to aid the design, test, and validation of DNA Strand Displacement (DSD) circuits at various levels of precision. Peppercorn tool [Badelt et al. 2020] generates reaction enumeration systematically and constructs a Chemical Reaction Network (CRN) from initial molecular species and the rules governing their interaction mechanisms. Multistrand [Schaeffer et al. 2015] is a software package for simulating, through stochastic simulation, the kinetics of multiple interacting nucleic acid strands. It supports multiple interacting strands and includes various simulation modes for studying trajectory samples. NUPACK [Zadeh et al. 2011] is a software suite for the analysis and design of nucleic acid structures, devices, and systems. It contains algorithms that address two fundamental classes of problems: sequence analysis and sequence design. KinDA [Berleant et al. 2018] is a Python package that provides a framework for analyzing the kinetic behavior of domain-level strand displacement (DSD) reaction networks with nucleotide sequences assigned to each domain, using the Peppercorn and Multistrand tools. Visual DSD [Lakin et al. 2011] is a programming language and software tool for designing nucleic acid circuits. Surface CRN Simulator [Clamons et al. 2020] is a Python-based web simulator for surface chemical reaction networks, supporting deterministic cellular automata and Boolean logic circuits. Table 1 presents a comparison of these tools regarding abstraction, hardware, and supported parallelism.

Simulator	**Abstraction**	**Hardware**	**Parallelism**
Peppercorn [Badelt et al. 2020]	DSD	CPU	Singlethread
Multistrand [Schaeffer et al. 2015]	Molecular	CPU	Singlethread
NUPACK [Zadeh et al. 2011]	Molecular	CPU	Singlethread
KinDA [Berleant et al. 2018]	DSD/Molecular	CPU	Singlethread
Visual DSD [Lakin et al. 2011]	CRN/DSD	CPU	Singlethread
Surface CRN Simulator [Clamons et al. 2020]	CRN	CPU	Singlethread
This work	CRN/DSD	CPU/GPU	Singlethread, Multithread

Table 1. Comparison of molecular system simulators.

The simulators previously listed were not compared against the DNAr software [Vieira et al. 2020] because: 1) they either are incapable of simulating the high number of species/reactions, or 2) have distinct degrees of precision, such as nucleotide sequences assigned to each domain. The DNAr software enhanced in this work is capable of simulating fairly large circuits (containing thousands of species and reactions) described at a more abstract level as Chemical Reaction Networks (CRNs). In other words, those other tools have different objectives, and it is not possible to make direct comparisons.

In this context, a gap in the literature is identified regarding the systematic evaluation of different backend implementations for accelerating the deterministic simulation of DNA-based CRNs on high-performance architectures. To address this gap, this work compares three backends integrated with DNAr: the original R `deSolve` backend, a Julia `DifferentialEquations.jl` backend with multicore parallelism, and a Python `ginSODA` backend with GPU execution, evaluating computational performance, resource consumption, and numerical precision across different simulation scenarios.

3. Methodology

3.1. Experimental Environment

The experiments were conducted on two distinct computing platforms, allowing for the evaluation of the backends' behavior across different hardware architectures. Table 2 presents the key characteristics of the environments used.

Characteristic	Machine 1	Machine 2
CPU	Ryzen 7 5700G (8 cores, 16 threads)	Core i3-8100 (4 cores, 4 threads)
GPU	Integrated	RTX 3080 Ti (12 GB, 10,240 CUDA cores)
Memory RAM	32 GB	8 GB
Operating System	Debian 13 (kernel 6.12.63)	Ubuntu 24.04 (kernel 6.8.0-124-generic)
R	4.5.0	4.5.0
Julia	1.12.1	1.12.1
Python	3.11.5	3.11.5

Table 2. Configuration of the experimental platforms.

All experiments were performed under the same conditions for each backend, using the same models, input parameters, initial conditions, stoichiometric matrix, integration interval, and time points.

3.2. Implementation

The DNAr architecture was extended to allow for the dynamic selection of the backend responsible for solving systems of ordinary differential equations (ODEs). Three backends were implemented:

1. **deSolve (R):** the original DNAr backend, utilizing the LSODA solver via the `deSolve` package;
2. **DifferentialEquations (Julia):** a backend developed in Julia using the `DifferentialEquations.jl`, `LSODA.jl`, and `LinearAlgebra` packages. System solving employs multi-threaded CPU parallelization and optimized matrix operations;
3. **GinSODA (Python/CUDA):** a backend developed using the GinSODA library [S. Nobile et al. 2019]. Data is exported by DNAr, processed in Python, and automatically converted into C++/CUDA code, which is compiled and executed on the GPU.

In the new backends, communication between DNAr and external environments is handled via CSV files.

Figure 1 illustrates the proposed architecture. The implementation preserves the original workflow based on the `deSolve` backend while adding new execution flows for the backends developed for `DifferentialEquations.jl` and `ginSODA`, enabling the comparison of different numerical solving strategies.

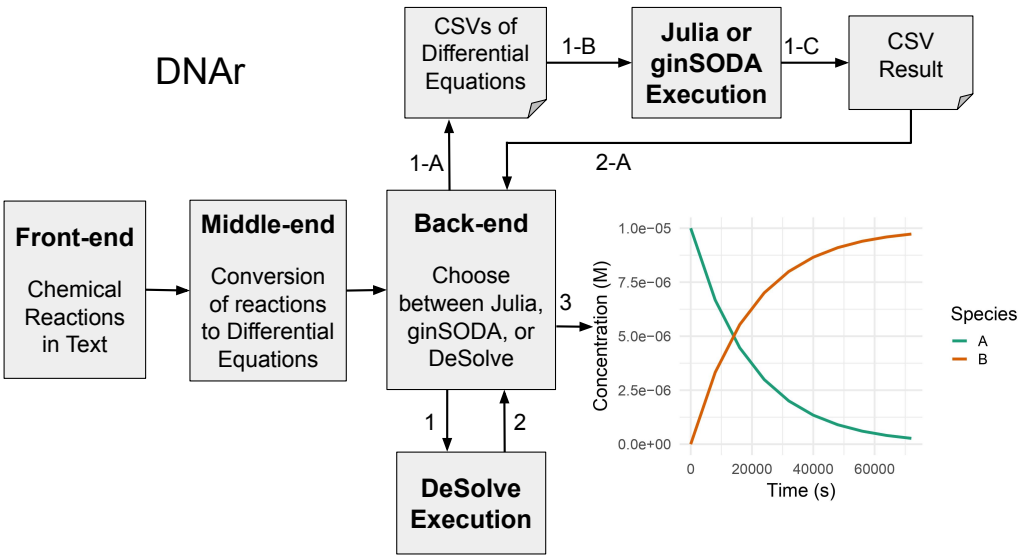

Figure 1. DNAr architecture with support for the new backends.

3.3. Benchmarks

To evaluate performance, twelve benchmarks representing different types of chemical reaction network models—varying in complexity and number of reactions—were selected. These benchmarks were:

- **4_bit_counter.R**: a 4-bit counter, generating the circuit's formal CRN reactions [Marks et al. 2019];
- **adapted_bhaskara_dsd_crn.R**: an analog DNA circuit to solve the adapted Bhaskara formula, generating the circuit's DSD reactions [Oliveira et al. 2023];
- **adapted_bhaskara_formal_crn.R**: an analog DNA circuit to solve the adapted Bhaskara formula, generating the circuit's formal CRN reactions [Oliveira et al. 2023];
- **chemical_neurons_react_0_9.R**: Chemical Neurons with an input of 0.9, generating the circuit's formal CRN reactions [Vieira et al. 2020];
- **chemical_neurons_react_1_1.R**: Chemical Neurons with an input of 1.1, generating the circuit's formal CRN reactions [Vieira et al. 2020];
- **clafa.R**: a 4-bit Carry Look-ahead Full Adder, generating the circuit's formal CRN reactions [Marks et al. 2019];
- **jkff.R**: a JK flip-flop, generating the circuit's formal CRN reactions [Marks et al. 2019].
- **neural_network_dsd_crn.R**: a network of analog DNA artificial neurons, generating the circuit's DSD reactions [Oliveira et al. 2023];
- **neural_network_formal_crn.R**: a network of DNA-based artificial neurons, generating the circuit's formal CRN reactions [Oliveira et al. 2023];

- **pam_clean.R**: the k-medoids clustering algorithm, generating the circuit's DSD reactions [Marks 2023];
- **simulate_dna_based_crn.R**: a simple chemical reaction, generating the circuit's DSD reactions [Vieira et al. 2020];
- **simulate_formal_crn.R**: a simple chemical reaction, generating the circuit's formal CRN reactions [Vieira et al. 2020].

Benchmark	Species	Reactions	Points
4_bit_counter	329	571	513
adapted_bhaskara_dsd_crn	153	90	50
adapted_bhaskara_formal_crn	29	28	50
chemical_neurons_react_0_9	5	8	10
chemical_neurons_react_1_1	5	8	10
clafa	376	676	41
jkff	236	408	181
neural_network_dsd_crn	528	316	50
neural_network_formal_crn	87	90	50
pam_clean	968	648	51
simulate_dna_based_crn	3	1	10
simulate_formal_crn	2	1	10

Table 3. Characteristics of the benchmarks used in the experiments.

Table 3 presents the benchmarks used in the experiments, which span varying levels of complexity — from reduced systems to circuits with hundreds of species and reactions — enabling an evaluation of backend behavior under different computational loads.

3.4. Performance Evaluation

The experimental evaluation compared the three backends implemented in DNAr: `deSolve`, `DifferentialEquations.jl`, and `ginSODA`. Each experiment was run independently ten times, and the mean and standard deviation of the results were calculated.

Execution time was measured using the `perf` tool via the `perf stat -r 10` command, considering the "seconds elapsed" metric. Peak memory consumption was obtained using the `/usr/bin/time -v` utility, considering the "Maximum Resident Set Size (Max RSS)" metric.

For the backend developed in Julia, measurements were taken using the `BenchmarkTools.jl` package to minimize the influence of Just-In-Time (JIT) compilation during comparisons. For the ginSODA-based backend, scenarios were evaluated both including and excluding the initial cost of CUDA code generation and compilation.

Result accuracy was assessed by comparing the solutions produced by the new backends against the original `deSolve`-based backend. The Normalized Root Mean Square Error (NRMSE), calculated for each state variable, was used as the error metric.

4. Results and Discussion

4.1. Accuracy of Solutions

The accuracy of the solutions produced by the new *backends* was evaluated using NRMSE (%), with the original *backend* based on the `deSolve` package serving as the reference (Table 4). Both *backends* demonstrated agreement exceeding 98.5% across all scenarios, confirming that the performance optimizations preserved the precision of the numerical solutions.

Benchmark	Julia (%)	ginSODA (%)
4_bit_counter	98.53	99.97
adapted_bhaskara_dsd_crn	100.00	100.00
adapted_bhaskara_formal_crn	99.99	100.00
chemical_neurons_react_0_9	100.00	100.00
chemical_neurons_react_1_1	100.00	100.00
clafa	100.00	100.00
jkff	99.47	100.00
neural_network_dsd_crn	100.00	100.00
neural_network_formal_crn	100.00	100.00
pam_clean	100.00	100.00
simulate_dna_based_crn	98.80	98.80
simulate_formal_crn	99.55	99.53

Table 4. Accuracy of the Julia and `ginSODA` backends relative to `deSolve`.

Small variations observed, such as the 98.53% obtained by Julia in the `4_bit_counter`, are due to differences in integration step control and the specificities of floating-point operations across different CPU and GPU architectures.

4.2. Computational Performance

The performance of the *backends* was evaluated based on the average execution time of the simulations. Two measurement scenarios were analyzed: (1) execution time including *overhead — encompassing* environment initialization, data preparation, and costs associated with the different architectures; and (2) execution time excluding *overhead — representing* solely the cost of solving the system of differential equations for the Julia and ginSODA implementations. The comparison between the R and Julia *backends* was conducted on both Machine 1 and Machine 2. An additional comparison involving GPU acceleration via `ginSODA` was performed only on Machine 2, as Machine 1 lacked a CUDA-compatible GPU. Across all measurements, the standard deviation remained low (e.g., less than 0.19 seconds for `4_bit_counter` and less than 0.09 seconds for `pam_clean`), confirming the reliability of the experiments.

4.2.1. Machine 1: Comparison between R and Julia

Figure 2 shows the average execution times obtained on Machine 1. Regarding measurements that include *overhead*, relative performance depends on the specific benchmark and its computational complexity. Benchmarks with a larger number of species, reactions, and time points benefit more from the Julia implementation. For the `4_bit_counter` benchmark (329 species, 571 reactions, and 513 time points),

the average time for R was 115.898 seconds, whereas the Julia implementation took an average of 37.4425 seconds, representing a reduction of approximately 67.7%. In the `pam_clean` benchmark (968 species, 648 reactions, and 51 time points), R took 84.575 seconds and Julia took 73.0481 seconds. In these scenarios involving longer execution times, the environment's startup costs are outweighed by the total runtime.

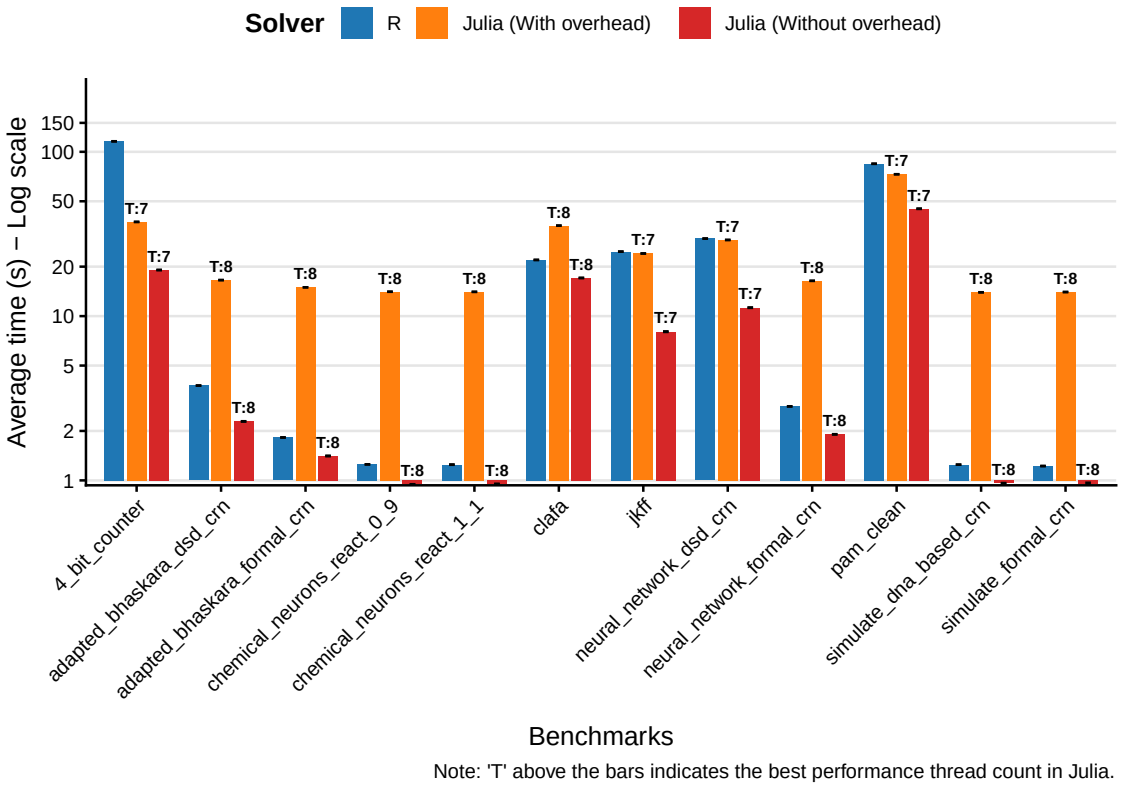

Figure 2. Average execution times on a logarithmic scale on Machine 1.

On the other hand, for simulations with lower computational cost, the costs associated with initializing the Julia environment account for a significant portion of the total time. In the `adapted_bhaskara_dsd_crn` example (153 species, 90 reactions, and 50 time points), the R *backend* showed an average time of 3.7778 seconds, whereas Julia took 16.5679 seconds. In the `chemical_neurons_react_0_9` (5 species, 8 reactions, and 10 time points) and `simulate_formal_crn` (2 species, 1 reaction, and 10 time points) benchmarks, the times recorded for R were 1.2518 seconds and 1.2231 seconds, respectively, while Julia took over 14 seconds.

Overhead-free measurement using the `BenchmarkTools.jl` package and the `@benchmark` macro isolates execution time from initial setup and JIT compilation. In smaller benchmarks, the impact of `overhead` is evident: for `adapted_bhaskara_dsd_crn` (153 species, 90 reactions, and 50 time points), the Julia execution time dropped from 16.5679 seconds to 2.2865 seconds (a reduction of approximately 86.2%). In the `chemical_neurons_react_0_9` (5 species, 8 reactions, and 10 time points) and `simulate_formal_crn` (2 species, 1 reaction, and 10 time points) benchmarks, times decreased from around 14 seconds to 0.9469 seconds and 0.9505 seconds, respectively. This result highlights the importance of using

`@benchmark` to analyze the performance of repeated executions with a pre-initialized environment. In more complex benchmarks, such as `4_bit_counter` (329 species, 571 reactions, and 513 time points), the time decreased from 37.4425 seconds to 19.0835 seconds, and for `pam_clean` (968 species, 648 reactions, and 51 time points), it dropped from 73.0481 seconds to 45.0219 seconds.

4.2.2. Machine 2: Comparison between R, Julia, and ginSODA

Figure 3 shows the average execution times obtained on Machine 2. When measuring with *overhead*, cross-language communication costs and the preparation stage penalize *backends* external to the R environment. The R *backend* remained competitive in simple cases but took approximately 114.18 seconds for `pam_clean` (968 species, 648 reactions, and 51 time points) and 146.48 seconds for `4_bit_counter` (329 species, 571 reactions, and 513 time points). The Julia *backend* showed intermediate performance, taking approximately 93.75 seconds for `pam_clean` (968 species, 648 reactions, and 51 time points). Meanwhile, the `ginSODA` *backend* took about 195.72 seconds for `pam_clean` (968 species, 648 reactions, and 51 time points) due to additional steps involving symbolic expression generation, benchmark preparation, and communication with the Python environment.

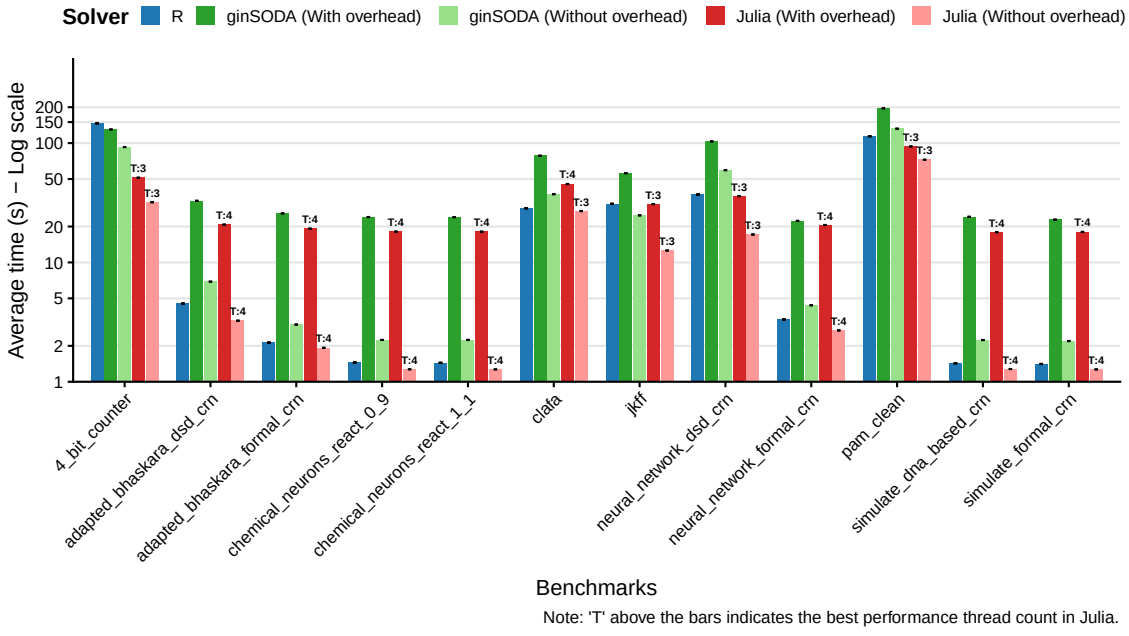

Figure 3. Average execution times on a logarithmic scale on Machine 2.

With *overhead-free* measurement — where `ginSODA` employs techniques such as *source code hashing* to avoid unnecessary recompilation of CUDA code [S. Nobile et al. 2019] — Julia's execution time for `pam_clean` (968 species, 648 reactions, and 51 time points) dropped to 72.50 seconds, while for `simulate_formal_crn` (2 species, 1 reaction, and 10 time points), it lowered from 18.02 seconds to 1.27 seconds. `ginSODA` reduced the time for `pam_clean` (968 species, 648 reactions, and 51 time points) to 132.01 seconds. In smaller benchmarks, memory

transfers between CPU and GPU and CUDA environment setup impose fixed costs that limit performance gains. This demonstrates that GPU usage does not guarantee an automatic reduction in execution time, as the outcome depends directly on the relationship between environment setup costs and the size of the differential system.

4.3. Memory Usage

Maximum memory consumption was measured during the complete execution of the simulations, encompassing data loading, environment initialization, benchmark preparation, and numerical resolution, as shown in Figure 4.

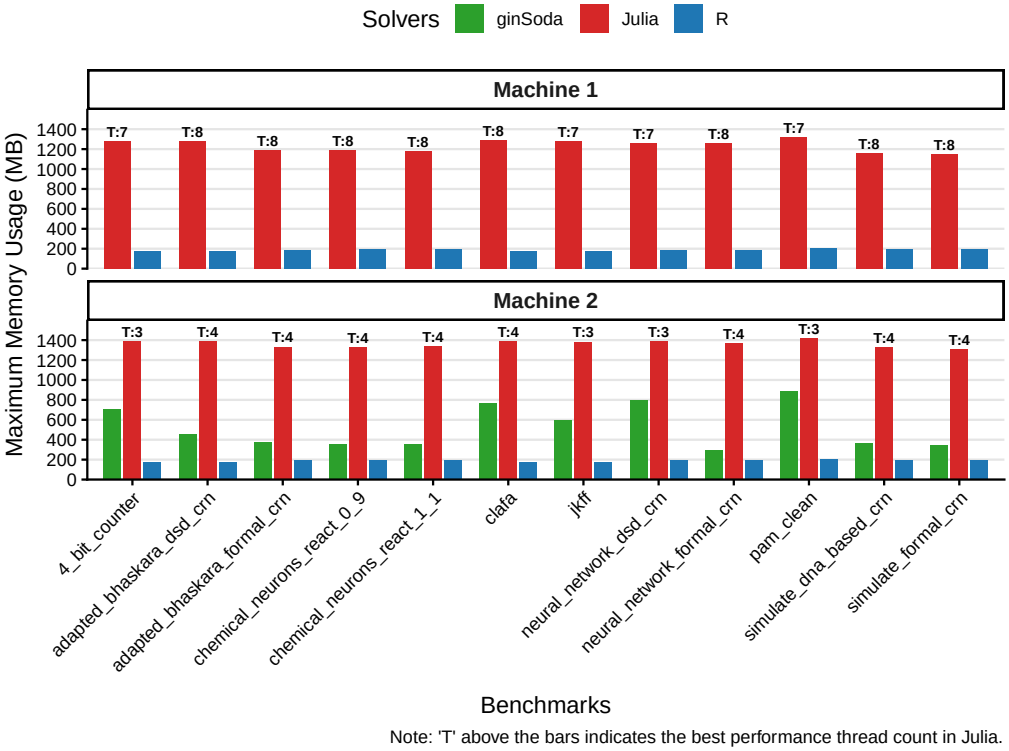

Figure 4. Maximum memory consumption in megabytes on Machines 1 and 2.

The R *backend* exhibited the lowest memory consumption values on both platforms, ranging from approximately 172 MB to 202 MB on Machine 1 and from 176 MB to 202 MB on Machine 2. This result stems from its direct execution within the native DNAr environment, requiring minimal initialization of external components.

In turn, the Julia *backend* recorded the highest memory consumption, ranging from approximately 1.13 GB to 1.32 GB on Machine 1 and from 1.31 GB to 1.42 GB on Machine 2. This increase is associated with the language's internal operations, including *just-in-time* (JIT) compilation, the loading of the `DifferentialEquations.jl` package, and the maintenance of structures supporting *thread-based* parallelism [Bezanson et al. 2017, Rackauckas and Nie 2017]. The minimal variation in memory usage across benchmarks suggests the predominance of fixed environment overheads.

The `ginSODA` *backend* exhibited intermediate memory usage ranging from approximately 295 MB to 886 MB on Machine 2. This behavior is attributed to symbolic

math generation performed by libraries such as `SymPy`, which maintain expression trees during model conversion [Meurer et al. 2017], as well as the structures required for data management within the heterogeneous CPU-GPU architecture [Nickolls et al. 2008].

4.4. Scalability and Impact of Threads in Julia

The scalability analysis evaluated the impact of increasing the number of Julia *threads* from 1 to 16 on Machine 1, and from 1 to 4 on Machine 2, on the `4_bit_counter` (329 species, 571 reactions, and 513 time points) and `simulate_formal_crn` (2 species, 1 reaction, and 10 time points) benchmarks, as shown in Figure 5 and Figure 6.

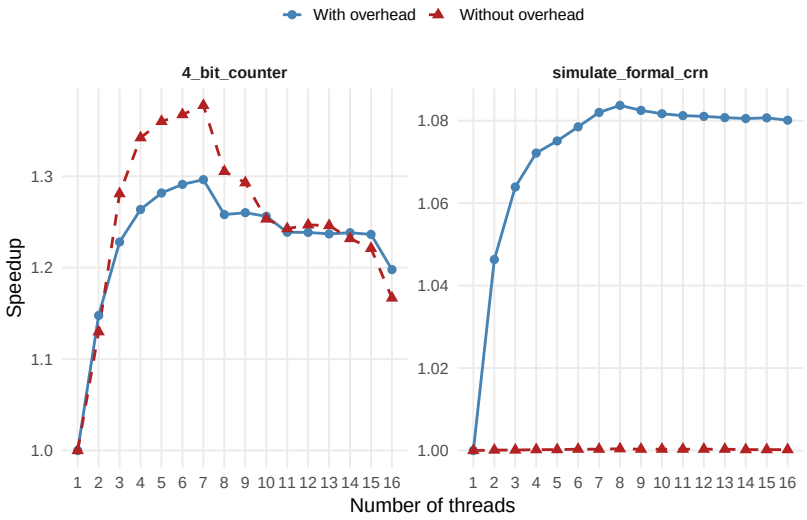

Figure 5. *Speedup* analysis with a progressive increase in the number of *threads* on machine 1.

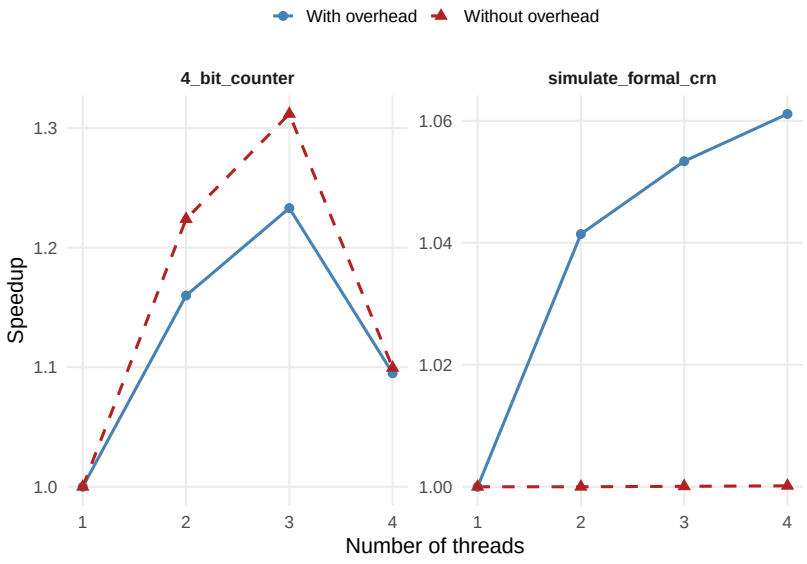

Figure 6. *Speedup* analysis with a progressive increase in the number of *threads* on machine 2.

On Machine 1, the `4_bit_counter` showed gains up to 7 *threads*, achieving a maximum *speedup* of approximately 1.30 in the scenario with *overhead* and 1.38 in the scenario without *overhead*. Since Machine 1 has $N = 8$ physical cores and 16 *threads*, the best performance was observed when using $N - 1$ (seven) cores. The `simulate_formal_crn` benchmark reached a maximum *speedup* of approximately 1.08 with 8 threads in the scenario with *overhead*, while remaining virtually constant with a *speedup* close to 1 in the scenario without *overhead*, demonstrating that small-scale circuits possess insufficient parallelizable workload to offset the management *overhead*.

On Machine 2, the `4_bit_counter` achieved its best performance with 3 *threads* (a *speedup* of approximately 1.23 with *overhead* and 1.31 without *overhead*). Since Machine 2 has $N = 4$ physical cores, the same behavior was observed when using $N-1$ (3, three) cores. The `simulate_formal_crn` benchmark showed little variation, reaching a maximum of 1.06 with *overhead*, and remained practically constant with a *speedup* close to 1 without *overhead*.

For the `4_bit_counter` benchmark, which presents a higher computational load, it was observed on both machines that the greatest *speedup* occurred when using $N-1$ physical cores. These results are affected by two architectural side-effects: (1) Amdahl's Law [Amdahl 1967], demonstrating that sequential stages and synchronization overheads limit the gains from parallelism [McCool et al. 2012, Rauber and Rünger 2013] when using only physical cores up to N-1; (2) the different running threads on the simultaneous multithread architecture (SMT) that saturates the pipeline from each core and shares the lower-level caches with two competing threads.

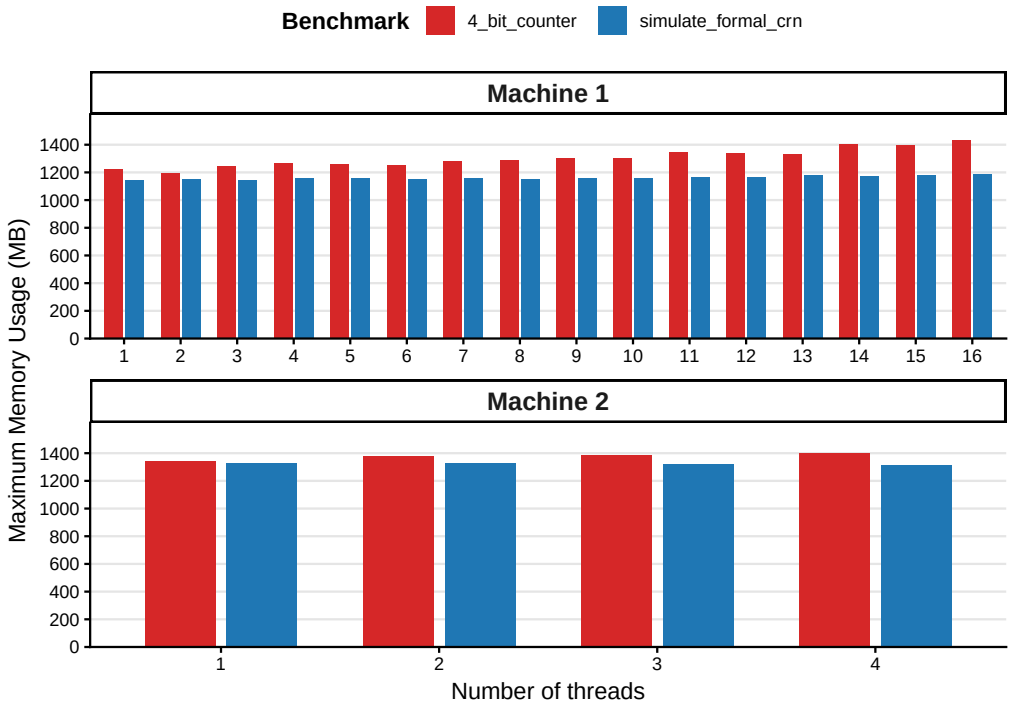

Figure 7. Memory consumption in megabytes according to the number of *threads*.

Regarding the memory consumption shown in Figure 7, the `4_bit_counter`

benchmark went from approximately 1225 MB with 1 *thread* to about 1425 MB with 16 *threads* on Machine 1, and from approximately 1350 MB to 1400 MB on Machine 2. For `simulate_formal_crn`, the variation ranged between approximately 1150 MB and 1185 MB on Machine 1, and between 1325 MB and 1315 MB on Machine 2. Increasing the number of *threads* causes a moderate and predictable rise in memory consumption due to the allocation of concurrent management structures [McCool et al. 2012].

5. Conclusion

This work presented the implementation and evaluation of new *backends* for the simulation of Chemical Reaction Networks (CRNs) within the DNAr package, including a Julia-based approach supporting multicore parallelism and a GPU-based approach utilizing ginSODA. The primary contribution was extending the simulator's architecture to allow for the dynamic selection of the numerical solving method, thereby making the tool more flexible and extensible.

Experimental results demonstrated that the Julia *backend* offers significant performance gains for computationally demanding systems. In contrast, the original R-based *backend* remains competitive for smaller simulations due to lower initialization overhead and memory usage. The GPU-based approach showed acceleration potential; however, its benefits depend directly on the problem size and the ability to amortize the costs associated with setup and execution on a heterogeneous architecture.

Beyond performance, the new *backends* maintained a high level of numerical agreement with the original method, indicating that the implemented optimizations did not compromise the quality of the solutions obtained. Scalability analysis showed that the use of multiple *threads* can reduce execution time, although gains are limited by problem characteristics and synchronization overhead.

Thus, the results demonstrate that various execution strategies can be explored within DNAr depending on the simulation's characteristics. The proposed implementation expands the simulator's utility, enabling a choice between different computational architectures and providing a foundation for future optimizations and new acceleration methods.

Acknowledgments

This work was carried out with the support of the Federal University of Mato Grosso do Sul — UFMS/MEC — Brazil and was financed in part by the Coordenação de Aperfeiçoamento de Pessoal de Nível Superior — Brasil (CAPES) — Finance Code 001.

Other relevant information

Generative Artificial Intelligence tools (ChatGPT, OpenAI; Gemini, Google) were used to support project development and debugging, as well as to review and suggest improvements to the article's text. All technical content, analyses, and conclusions presented were written, reviewed, and validated by the authors.

References

Amdahl, G. M. (1967). Validity of the single processor approach to achieving large scale computing capabilities. *AFIPS Conference Proceedings*, 30:483–485.

Anderson, N. G. and Bhanja, S. (2014). *Field-coupled nanocomputing*. Springer.

Arute, F., Arya, K., Babbush, R., Bacon, D., Bardin, J. C., Barends, R., Biswas, R., Boixo, S., Brandao, F. G., Buell, D. A., et al. (2019). Quantum supremacy using a programmable superconducting processor. *Nature*, 574(7779):505–510.

Asanović, K., Bodik, R., Catanzaro, B. C., Gebis, J. J., Husbands, P., Keutzer, K., Patterson, D. A., Plishker, W. L., Shalf, J., Williams, S. W., and Yelick, K. A. (2006). The landscape of parallel computing research: A view from berkeley. Technical Report UCB/EECS-2006-183, EECS Department, University of California, Berkeley.

Badelt, S., Grun, C., Sarma, K. V., Wolfe, B., Shin, S. W., and Winfree, E. (2020). A domain-level dna strand displacement reaction enumerator allowing arbitrary non-pseudoknotted secondary structures. *Journal of the Royal Society Interface*, 17(167):20190866.

Berleant, J., Berlind, C., Badelt, S., Dannenberg, F., Schaeffer, J., and Winfree, E. (2018). Automated sequence-level analysis of kinetics and thermodynamics for domain-level dna strand-displacement systems. *Journal of the Royal Society Interface*, 15(149):20180107.

Bezanson, J., Edelman, A., Karpinski, S., and Shah, V. B. (2017). Julia: A fresh approach to numerical computing. *SIAM review*, 59(1):65–98.

Bohr, M. (2009). A 30 year retrospective on dennard's mosfet scaling paper. *IEEE Solid-State Circuits Society Newsletter*, 12(1):11–13.

Clamons, S., Qian, L., and Winfree, E. (2020). Programming and simulating chemical reaction networks on a surface. *Journal of the Royal Society Interface*, 17(166):20190790.

Hennessy, J. L. and Patterson, D. A. (2011). *Computer architecture: a quantitative approach*. Elsevier.

Lakin, M. R., Youssef, S., Polo, F., Emmott, S., and Phillips, A. (2011). Visual DSD: a design and analysis tool for DNA strand displacement systems. *Bioinformatics*, 27(22):3211–3213.

Lent, C. S., Henderson, K. W., Kandel, S. A., Corcelli, S. A., Snider, G. L., Orlov, A. O., Kogge, P. M., Niemier, M. T., Brown, R. C., Christie, J. A., et al. (2016). Molecular cellular networks: A non von neumann architecture for molecular electronics. In *2016 IEEE International Conference on Rebooting Computing (ICRC)*, pages 1–7. IEEE.

Marks, R. A. (2023). *DNA Strand Displacement (DSD) as a Hardware Substrate for Digital, Analog and Machine Learning Systems*. PhD thesis, Universidade Federal de Minas Gerais.

Marks, R. A., Vieira, D. K., Guterres, M. V., Oliveira, P. A., and Neto, O. P. V. (2019). Dnar-logic: A constructive dna logic circuit design library in r language for molecular computing. In *Proceedings of the 32nd Symposium on Integrated Circuits and Systems Design*, pages 1–6.

McCool, M., Robison, A. D., and Reinders, J. (2012). *Structured Parallel Programming: Patterns for Efficient Computation*. Morgan Kaufmann, Burlington, MA.

Meurer, A., Smith, C. P., Paprocki, M., et al. (2017). Sympy: symbolic computing in python. *PeerJ Computer Science*, 3:e103.

Moore, G. E. (1965). Cramming more components onto integrated circuits. *Electronics*, 38(8):114–117.

Nickolls, J., Buck, I., Garland, M., and Skadron, K. (2008). Scalable parallel programming with cuda. *ACM Queue*, 6(2):40–53.

Oliveira, P. A., Marks, R. A., Teixeira, J. V., Guterres, M. V., and Neto, O. P. V. (2023). Dnar-analog: A library with a multiplexer to easily design, program, and simulate dsd analog circuits. In *2023 IEEE 14th Latin America Symposium on Circuits and Systems (LASCAS)*, pages 1–4. IEEE.

Qian, L. and Winfree, E. (2011). Scaling up digital circuit computation with dna strand displacement cascades. *science*, 332(6034):1196–1201.

Rackauckas, C. and Nie, Q. (2017). Differentialequations. jl–a performant and feature-rich ecosystem for solving differential equations in julia. *Journal of open research software*, 5(1):15–15.

Rauber, T. and Rünger, G. (2013). *Parallel Programming: for Multicore and Cluster Systems*. Springer, Berlin, Germany, 2 edition.

S. Nobile, M., Cazzaniga, P., Besozzi, D., and Mauri, G. (2019). ginsoda: massive parallel integration of stiff ode systems on gpus: M. s nobile et al. *The Journal of Supercomputing*, 75(12):7844–7856.

Schaeffer, J. M., Thachuk, C., and Winfree, E. (2015). Stochastic simulation of the kinetics of multiple interacting nucleic acid strands. In *DNA Computing and Molecular Programming: 21st International Conference, DNA 21, Boston and Cambridge, MA, USA, August 17-21, 2015. Proceedings 21*, pages 194–211. Springer.

Schuman, C. D., Kulkarni, S. R., Parsa, M., Mitchell, J. P., Date, P., and Kay, B. (2022). Opportunities for neuromorphic computing algorithms and applications. *Nature Computational Science*, 2(1):10–19.

Shen, Y., Harris, N. C., Skirlo, S., Prabhu, M., Baehr-Jones, T., Hochberg, M., Sun, X., Zhao, S., Larochelle, H., Englund, D., et al. (2017). Deep learning with coherent nanophotonic circuits. *Nature photonics*, 11(7):441–446.

Vieira, D. K., Guterres, M. V., Marks, R. A., Oliveira, P. A., Fonte Boa, M. C., and Vilela Neto, O. P. (2020). Dnar: An r package to simulate and analyze crn and dsd networks. *ACS Synthetic Biology*, 9(12):3416–3421.

Zadeh, J. N., Steenberg, C. D., Bois, J. S., Wolfe, B. R., Pierce, M. B., Khan, A. R., Dirks, R. M., and Pierce, N. A. (2011). Nupack: Analysis and design of nucleic acid systems. *Journal of computational chemistry*, 32(1):170–173.