Key Engineering Materials
ISSN: 1662-9795, Vol. 1050, pp 59-68
doi:10.4028/p-86zA6K
© 2026 The Author(s). Published by Trans Tech Publications Ltd, Switzerland.
Submitted: 2026-01-15
Revised: 2026-02-16
Accepted: 2026-02-17
Online: 2026-04-16

Investigation of Code Optimization Strategies for Enhancing the Performance of Static Recrystallization Cellular Automata Models

Mateusz SITKO[1,a*], Szymon SITARZ[1,b] and Łukasz MADEJ[1,c]

[1]AGH University of Krakow, al. Adama Mickiewicza 30, 30-059 Kraków

[a]msitko@agh.edu.pl, [b]sitarzs@agh.edu.pl, [c]lmadej@agh.edu.pl

Keywords: cellular automata, static recrystallization, computation efficiency, microstructure evolution

Abstract. Understanding and predicting static recrystallization (SRX) behavior is crucial for controlling the microstructure and mechanical properties of metals during thermomechanical processing. Among various numerical modelling approaches that can be used to support experimental studies on this topic is the cellular automata (CA) method. This approach gained significant attention due to its ability to simulate microstructural evolution at the mesoscale with high spatial resolution. However, the main limitation of CA models is their significant simulation time, especially for the 3D computational domains. Therefore, the paper focuses on enhancing the efficiency of CA SRX simulations to deliver results within an acceptable time frame. The goal is to minimize computation time and memory usage through code-level optimization, without altering the hardware or compiler settings. Optimization is performed on the sequential version of the validated CA SRX code. Initially, the source code was analyzed using a profiler tool to identify performance bottlenecks. The most inefficient parts of the code were then reimplemented to eliminate these bottlenecks. Optimization methods included eliminating redundant functions, modifying neighbor assignments in the automata space, reducing class data structures, enabling direct access to attributes, simplifying mathematical formulas, and removing unused objects. The obtained results are also validated against the output from the sequential version to ensure the model's predictive capabilities. The work clearly demonstrates that the optimization improved simulation efficiency across all tested variants, with only minor increases in memory usage.

Introduction

Numerical simulations are widely applied across many scientific areas to model complex physical, thermal, and deformation processes [1]. Continuous advances in computational power have enabled simulations on three-dimensional domains at micro- and nanoscale levels, allowing explicit representation of microstructural features such as grains, crystallographic orientations, inclusions, and phases. Understanding material behavior during processing makes it possible to predict final material properties with increasing accuracy. Consequently, modeling phenomena such as recrystallization, phase transitions, and texture evolution has become an essential research tool that complements and extends experimental investigations, particularly where direct observation is limited or costly. Numerical approaches, including Monte Carlo (MC) and Cellular Automata (CA) methods [2,3] are commonly employed for this purpose, as they provide flexible frameworks for simulating microstructural evolution. Nevertheless, such simulations are often computationally demanding, especially for large domains or long process times, which motivates research into methods for improving computational efficiency.

Among the available numerical techniques [4], the cellular automata method is desirable due to its ability to represent local interactions and discrete microstructural changes directly. However, the efficiency of CA-based simulations strongly depends on several factors, including neighborhood definitions, data structures, update rules, and memory-access patterns [5]. As model resolution and domain size increase, the computational cost grows significantly, making performance optimization a key aspect of practical CA applications in materials science [6].

This article is an open access article under the terms and conditions of the Creative Commons Attribution (CC BY) license (https://creativecommons.org/licenses/by/4.0)

Efficient use of available computational resources is therefore crucial for simulation development. Performance improvements can be achieved through hardware upgrades, compiler-level optimizations, and algorithmic or manual code-level changes. While modern hardware generally enhances performance, notable gains can also be obtained through appropriate compiler selection and optimization settings. Different compilers may generate executables with varying performance characteristics for identical source code due to optimization techniques such as loop unrolling, inlining, software prefetching, and dead code elimination. Consequently, compilation strategy plays a non-negligible role in overall simulation efficiency [7].

Manual optimization remains an important step in high-performance simulation development and requires careful analysis of the source code to identify computational bottlenecks. Profiling tools are essential in this process, as they enable detailed examination of execution time, memory usage, function call frequency, and call hierarchies during runtime. Such insights guide targeted modifications that can substantially reduce computational overhead. In addition, the choice of programming language and supporting libraries significantly affects code performance. The C++ programming language offers fine-grained control over memory management and computational resources, while the selection and configuration of appropriate data structures and algorithms from the Standard Template Library (STL) can further influence code execution speed and scalability.

Further acceleration of CA-based microstructural simulations can be achieved using parallel computing techniques [8]. Modern computing architectures increasingly use multi-core processors and heterogeneous systems, making parallelization a natural direction for improving performance. Cellular Automata and Monte Carlo methods are particularly well-suited for parallel execution due to their synchronous transition rules and limited data dependencies. In this case, parallel implementations using shared-memory approaches (e.g., OpenMP) [9], distributed-memory frameworks (e.g., MPI) [10], or graphics processing units (GPUs) [11] can significantly reduce computation time for large simulation domains. However, appropriate domain decomposition, load balancing, and efficient communication strategies are crucial for achieving good scalability and minimizing parallel overhead during computations. A combination of these strategies is often required to achieve significant and scalable performance improvements in practice. Such customised parallel implementation enables high-resolution and large-scale CA simulations that would otherwise be impractical within reasonable computation times [12].

Thus, in this work, selected optimization strategies are applied to a CA-based static recrystallization (SRX) model to evaluate their impact on computational performance. The study focuses on identifying critical performance-limiting components of the simulation and assessing the achievable reduction in computational time, with the aim of improving the applicability of CA simulations under operational industrial conditions.

Methodology

The investigated CA SRX model was developed, validated and in detail presented in [13]. The current CA SRX simulation case studies were done in three variants. In the first one, the simulation was executed until the recrystallized volume fraction reached 30% (Fig. 1a). The second variant is based on a fully recrystallized microstructure, which is treated as input data for the grain growth (GG) simulation, where the driving force is only due to grain-boundary curvature. The initial temperature in this case was 600 °C, and the sample was heated for 14s to reach the final temperature of 614°C (Fig. 1b). In the third variant, the recrystallisation is simulated until the material is fully recovered, and then the grain growth is analysed. The sample is heated from 600 °C for 80s to the final temperature of 680 °C (Fig. 1c). Therefore, the latter variant is characterized by significantly longer simulation times.

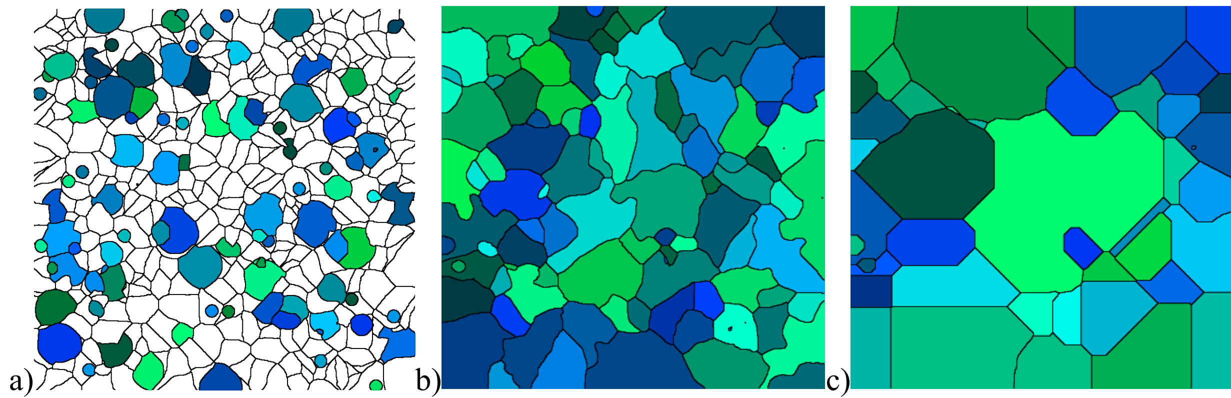

Fig. 1 Final microstructure after : a) variant 1 (SRX), b) variant 2 (GG), c) variant 3 (SRX+GG) simulation.

In the first step of the investigation, an extensive code profiling with the use of the Visual Studio profiling tool for the three mentioned variants was performed. Five main model functions responsible for the simulation of the recovery phenomenon, stored energy-driven grain growth, grain boundary curvature growth, simulation and computational domain update were evaluated. Results from the analysis are presented in Tab. 1.

Table 1 Results from initial code profiling.

Function name		% of code execution time		
		Variant 1	**Variant 2**	**Variant 3**
Recovery Stage		20.32%	17.42%	3.86%
Recrystallization Stage	Stored Energy Driven Grain Growth	49.56%	0.00%	66.61%
	Curvature Driven Grain Growth	0.43%	25.43%	19.28%
Update Progress of Simulation		20.76%	33.93%	7.49%
Update CA Space		5.47%	14.49%	2.67%

The profiling results revealed noticeable differences in computational performance and resource utilization of the CA SRX model depending on the simulation variant. Such different utilization of the main models' components provides potential opportunities for code optimization in these areas.
Therefore, in the next step, code optimizations were performed based on these initial profiling results, and are presented in the following chapters.

Analysis of Data Storage Containers

The first issue considered in increasing application efficiency was selecting the appropriate container for data storage. The code profiling evaluation clearly indicates that containers are a problematic area requiring further investigation. Therefore, suitable data structures from the Standard Template Library have been analyzed including deque, vector, list or forward_list. The calculated average simulation times for different containers for each investigated simulation variant are presented in Fig. 2 - Fig. 4. The red line in the plot represents the time reduction in comparison to the initial container used in previous work (vector<Cell>*). In terms of comparison, additional objects stored in a container Cell, were compared to pointers to those objects, Cell*.

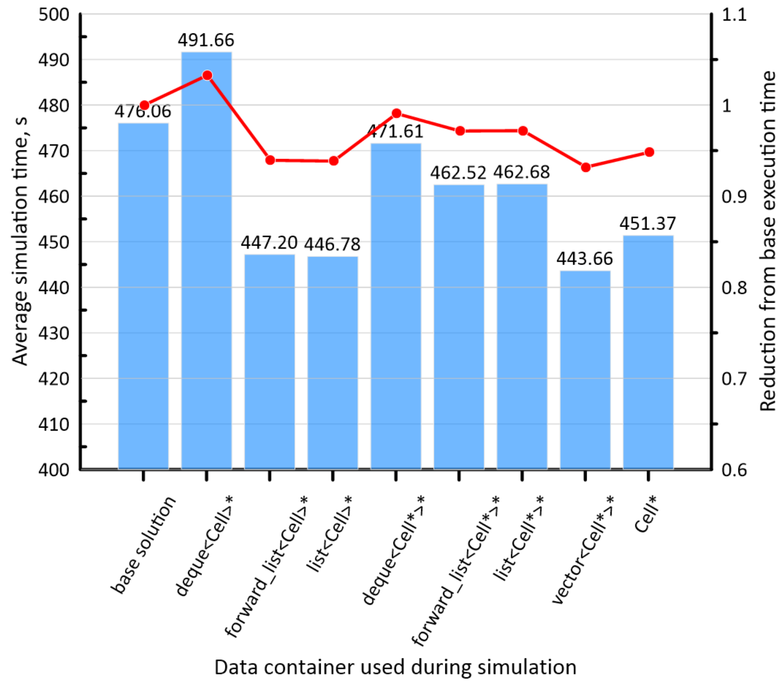

Fig. 2 Influence of data container type on simulation times for variant 1 (Bars represent average time from 3 runs of simulation, red lines overall time reduction compared to the base solution vector<Cell>*).

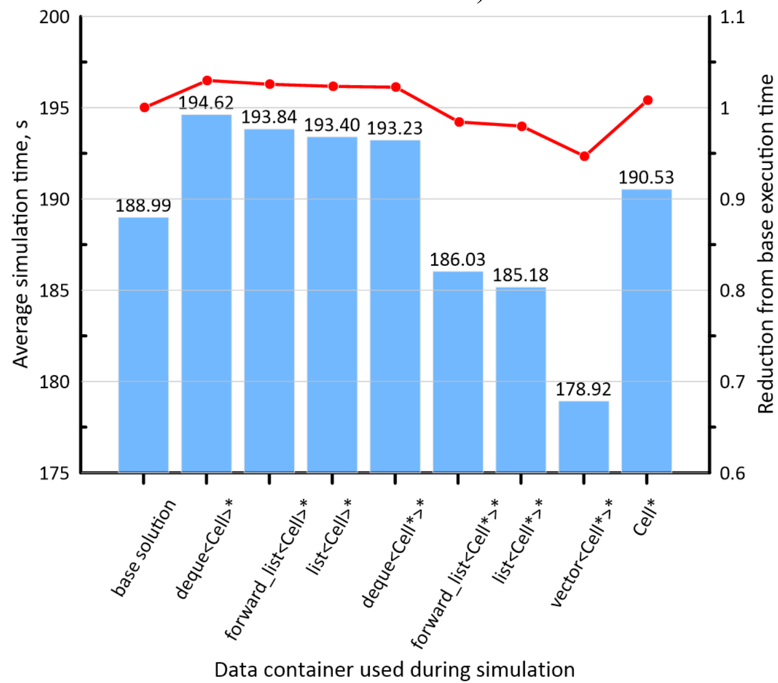

Fig. 3 Influence of data container type on simulation times for variant 2.

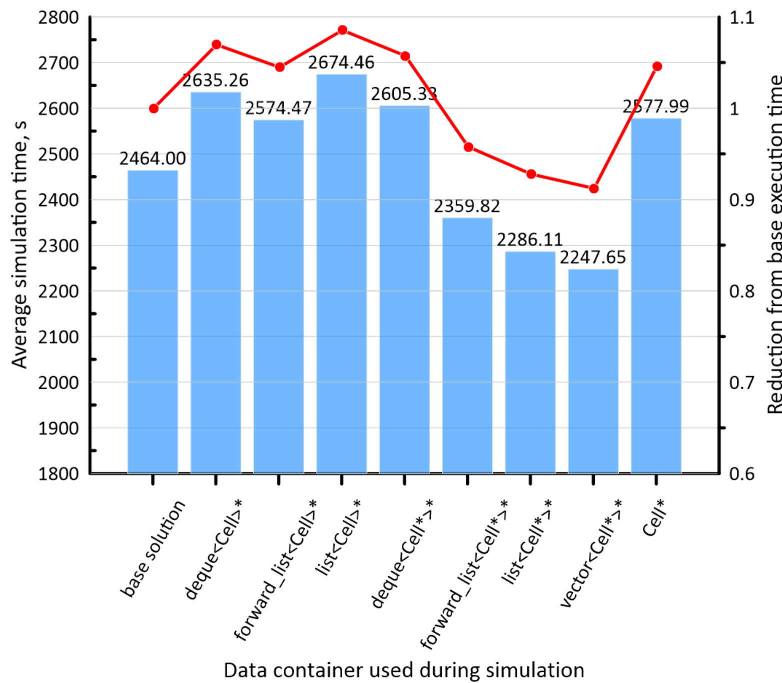

Fig. 4 Influence of data container type on simulation times for variant 3.

The shortest simulation time was achieved with a vector of pointers to objects. When vector structure is used to store objects, it is also the most efficient data container for the second and third variants (Fig. 2-3). It is worth noting that the main operation on a vector is limited to a single CA iteration (single CA time step). Overall, the benefits of such a change result in a time reduction of around 9%. The results indicate that the least efficient data structure for CA simulation is a deque. In all cases, the deque container is the slowest when storing pointer data types. Deque and vector are sequential containers that provide random access to the stored elements. The main difference between them is how they store elements in memory. A vector stores data in fast cache memory; on the other hand, elements in a deque are scattered throughout the memory. That is the main reason why vectors are useful when operating on long adjacent data sequences.

Subsequent analyzes should be focused on the container forward_list, which implements a one-way list data structure. In this case, each element has a pointer to the next element, so iteration can be done only forward. However, another more advanced container called a list implements a two-way data structure, where each element additionally has a pointer to the previous element. In both cases, direct access to the elements of the data structure is not possible. To reach a particular stored element, the container must be iterated over until that element is accessed. In SRX simulations, containers are iterated only forward, so both data structures achieve similar performance (Fig. 2- Fig. 4).

A standard dynamic array derived from the C language has also been investigated. This data structure does not deviate from STL container implementations in terms of simulation speed. Efficiency results are in the middle of the chart as presented in (Fig. 2- Fig. 4).

Finally, the average time has been calculated separately for containers storing pointers and for containers storing objects, as shown in Fig. 5. In this analysis, an array from the C language was not considered.

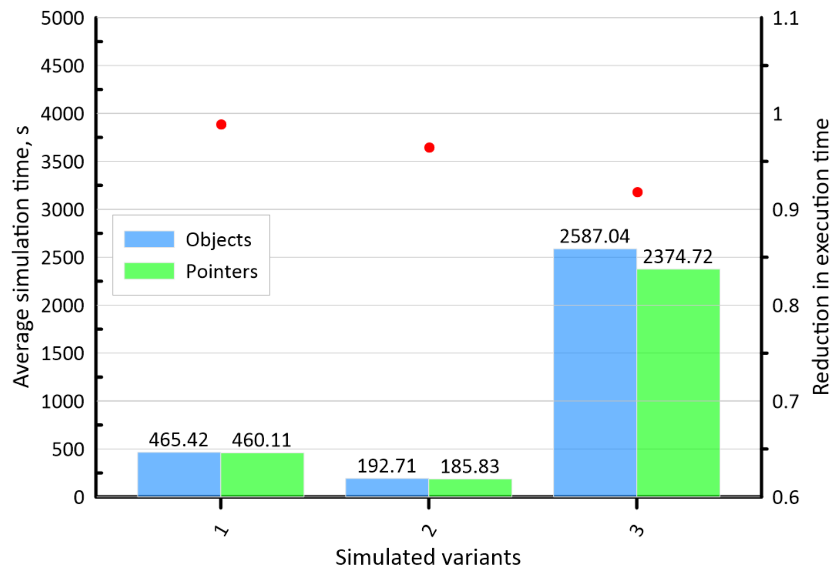

Fig. 5 Analysis of the types of elements stored in different containers (red dots represent time reduction in terms of moving from Objects: Cell to Pointers: Cell*).

As presented, the vector with pointers to CA cell objects is the most effective method in that simulation case (the highest profit of around 9% was achieved in variant 3, where both recrystallization and curvature-driven growth were simulated). In conclusion, implementing pointers accelerates CA code execution. However, if a container with many elements is needed, implementing a list data structure is still a better choice.
Based on the current observations, further optimization tasks were performed using vector-storing pointers and the results are gathered in the next chapter.

Analysis of Code Optimization Techniques

Eight different CA code optimizations were carried out based on the initial profiling results.

1. Overloaded functions were eliminated. To improve simulation, all iterations in the data structure have to be handled in the same way. Taking this into consideration, the reading elements from the vector were changed from the at() function to an iterator. The at() function returns a reference to the element at the given position. This function can be used to iterate over a vector, but calling a function in each iteration can be very time-consuming; thus, an iterator was implemented.

2. Rearrangement of CA neighbor storage in the memory. In the CA SRX model, three different neighborhood types can be used, as represented in Fig. 6.

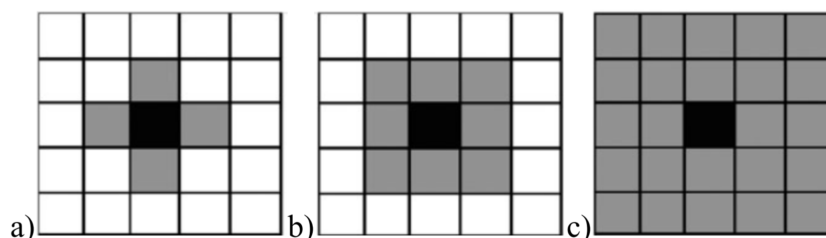

Fig. 6 a) neighbourhoodZERO - von Neumann neighbourhood b) neighbourhoodONE - Moore's neighborhood c) neighbourhoodTWO - extended Moore's neighbourhood.

In the original CA SRX model, neighbors were assigned to each CA cell at every simulation step. To accelerate the simulation, a new approach was developed. New data structures were added to the definition of the Cell class. They store the neighbours of a particular cell in the object. In that case, a new function for assigning neighbors was developed and implemented. The function is executed during the CA code initial phase. Therefore, it does not impact the main simulation computational

time. During the simulation, time is not wasted on assigning CA neighbors at each step, but additional memory is required.

3. Elimination of duplications in the new vector containers. It is possible to contain all neighborhood types in a single data structure if the data is arranged in the proper order (Fig. 7). This code modification not only accelerated the SRX model execution but also reduced memory usage during the simulation.

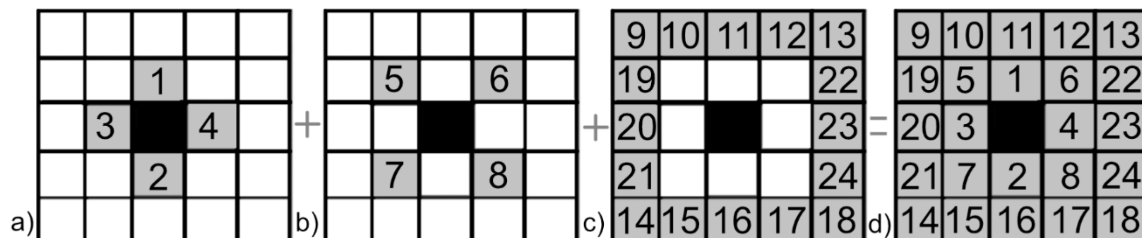

Fig. 7 Representation of neighborhoods: a) von Neumann neighbourhood (CA cells 1-4), b) Moore's neighbourhood (a + CA cells 5-8:), c) extended Moore's neighbourhood (a + b + c), d) new vector container.

4. Introduction of CA space composition. CA cells in the computational domain are stored in a three-dimensional array, which slows CA SRX model iteration, but also affects memory allocation. It causes memory fragmentation, with three distinct segments of memory randomly placed. During each iteration, it is necessary to jump across different segments of memory to access specific elements in the array. Therefore, in the current investigation, a three-dimensional array was converted into a one-dimensional array. Allocation and deallocation of such array also reduce the number of code lines from a threefold-nested loop to a single line of code. All elements in memory are next to each other, facilitating fast access to values. Iteration on the whole CA space is a frequent operation in the CA SRX simulation. A one-dimensional array can use only one loop. To maintain the cellular automata's three-dimensional conception, threefold nested loops were retained during each iteration. As a result, speed and memory usage were improved, but one-dimensional arrays also have their limitations. The C++ language allows the allocation of only a limited amount of memory. In the paper, the testing CA computational domain was 800×800×1, so there are 640000 CA cell objects to allocate, which is a relatively small number.

5. Introduction of attributes encapsulation on main objects. Correct practices for object-oriented programming include using getter and setter functions to access or modify attribute values. Data encapsulation provides hiding private data, so it cannot be accessed directly. It is possible to implement sensitive data handling within the class. On the other hand, it adds additional lines of code, which could potentially slow down the CA SRX simulation. The main purpose of the paper is to accelerate the CA SRX code, so data sensitivity was skipped, and attribute visibility was changed to public. That amendment eliminated the necessity of using functions, so attributes could be referred to directly.

6. Substitution of Push_back() by emplace_back(). Both of them are vector methods that realize the same functionality, which is adding elements at the end. Theoretically, emplace_back first reserves memory for the new element, then inserts the element into the allocated space. Whereas push_back first creates a temporary object on the stack. Then it extends the vector, and std::move inserts an element from the stack into a new memory space. In theory, emplace_back avoids additional copying and moving data, which is more efficient.

7. Rewriting of algebraic functions. The C++ language provides math.h library, which enables the execution of various mathematical operations such as root extraction, exponentiation, rounding, and calculating exponents. After different combinations of such elements and many tests, the following conclusions have been drawn:

- sqrt() is faster than pow(x, 0.5)
- pow(x,1/3) is faster than cbrt()
- exp(-x) is faster than 1/exp(x)

- exp(x) is faster than pow(e, x);
- 1/pow(x,0.5) is faster than pow(x, -0.5)
- 1/sqrt(x) is faster than 1/pow(x, 0.5)

8. Detailed code analysis to localize unnecessary operations, which slow down the whole CA SRX code. After introducing the above-mentioned changes, the profiling process was repeated. As a result, it pointed out additional redundant data structure allocation. The most problematic part was a map container from STL. Finally, allocation was executed in the main simulation loop, so the map was removed.

Each presented code optimization technique improved simulation efficiency for at least two simulation variants as presented in Figure Fig. 8a-c. Only a change in the assigning CA neighbors greatly increased memory consumption (Fig. 8d). This was eventually reduced in the final code optimization step by reducing unnecessary object allocation. Afterwards, memory consumption continues to remain on the same level, which was 287 MB.

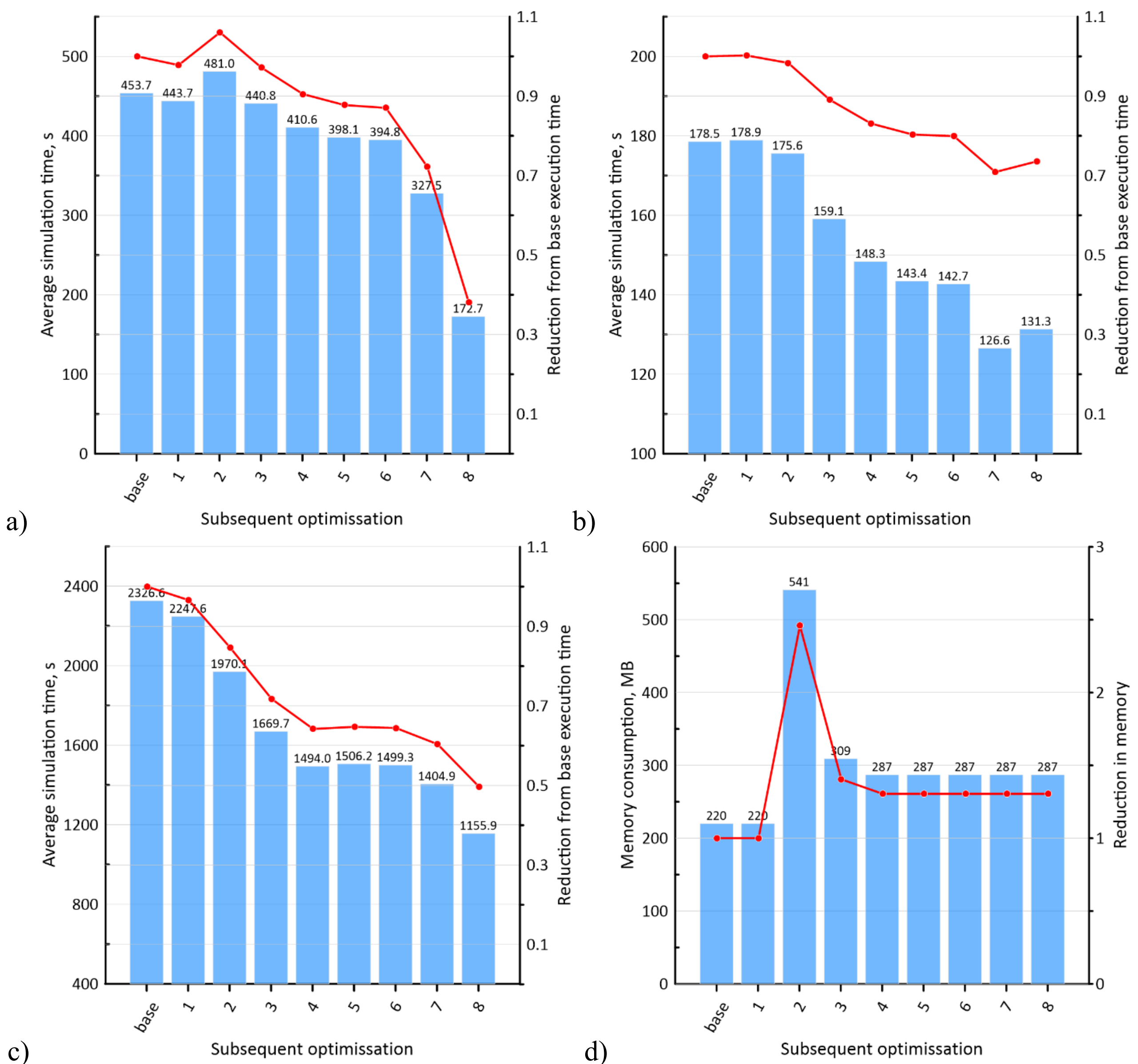

Fig. 8 Time changes after subsequent optimization a) variant 1, b) variant 2, c) variant 3, d) memory consumption.

As shown in Fig. 8, overall, benefits from different optimizations allow simulating the same scenarios in 40-50% percentage points of the initial time.

Summary

The paper points out that code optimization techniques can significantly improve CA SRX code efficiency on different levels. In the investigated case code optimization reduced simulation time approximately two times. However, a crucial step during the implementation of new functionalities is preparing the initial application profile using the profiler tool, which enables proper interpretation and localization of the most problematic areas, often leading to code execution bottlenecks. Such code profiling provides clear guidance on possible further code improvement.

The findings of the current research can be summarized as follows:

- A proposed approach to CA neighbor assignment in cellular automata code results in a significant acceleration of execution times with only a slight increase in memory usage. Eventually, memory consumption increases only by about 60 MB from the original memory usage.
- The most efficient data structure for CA simulations proved to be a vector from the Standard Template Library (STL).
- Reduction of the array dimensions that hold the 3D CA computation domain prevents frequent iterations and speeds up the simulation execution.
- The use of at() function and proper algebraic functions from math.h library, getter or push_back() provides a significant execution time reduction of the CA code.

Based on these findings, the next stage of this investigation will be focused on the CA SRX model adaptation to the parallel code execution and additional code corresponding optimisations to further increase simulation performance.

Acknowledgement

We would like to express our sincere gratitude to Mr. Marcin Chrobak for his initial work on preparing application profiles and proposing various approaches to code optimization. The financial assistance of the National Science Centre project No. 2024/55/D/ST8/00192 is acknowledged.

References

[1] M. Pietrzyk, L. Madej, L. Rauch, D. Szeliga, Computational Materials Engineering Achieving high accuracy and efficiency in metals, 2015.

[2] K. Teferra, D.J. Rowenhorst, Optimizing the cellular automata finite element model for additive manufacturing to simulate large microstructures, Acta Mater 213 (2021) 116930. https://doi.org/10.1016/j.actamat.2021.116930.

[3] T.M. Rodgers, D. Moser, F. Abdeljawad, O.D.U. Jackson, J.D. Carroll, B.H. Jared, D.S. Bolintineanu, J.A. Mitchell, J.D. Madison, Simulation of powder bed metal additive manufacturing microstructures with coupled finite difference-Monte Carlo method, Addit Manuf 41 (2021) 101953. https://doi.org/10.1016/j.addma.2021.101953.

[4] O. Vodka, M. Shapovalova, Exploration of cellular automata: a comprehensive review of dynamic modeling across biology, computer and materials science, Computer Methods in Material Science 23 (2023) 57–80. https://doi.org/10.7494/cmms.2023.4.0820.

[5] M. Sitko, M. Czarnecki, K. Pawlikowski, L. Madej, Evaluation of the effectiveness of neighbors' selection algorithms in the random cellular automata model of unconstrained grain growth, Materials and Manufacturing Processes (2023) 1–11. https://doi.org/10.1080/10426914.2023.2196753.

[6] M. Sitko, Q. Chao, J. Wang, K. Perzynski, K. Muszka, L. Madej, A parallel version of the cellular automata static recrystallization model dedicated for high performance computing platforms – Development and verification, Comput Mater Sci 172 (2020) 109283. https://doi.org/10.1016/j.commatsci.2019.109283.

[7] B. Bhowmik, K.K. Girish, H. Pandey, P. Prabhanjans, Optimizing Data Movement in Heterogeneous Computing: A LASSA-based Approach for Efficient Nucleation List Precomputation, in: Proceedings of 2025 3rd International Conference on Intelligent Systems, Advanced Computing, and Communication, ISACC 2025, Institute of Electrical and Electronics Engineers Inc., 2025: pp. 532–537. https://doi.org/10.1109/ISACC65211.2025.10969447.

[8] A. De Rango, A. Giordano, G. Mendicino, R. Rongo, W. Spataro, Tailoring load balancing of cellular automata parallel execution to the case of a two-dimensional partitioned domain, Journal of Supercomputing 79 (2023) 9273–9287. https://doi.org/10.1007/s11227-023-05043-3.

[9] M. Oliverio, W. Spataro, D. D'Ambrosio, R. Rongo, G. Spingola, G.A. Trunfio, OpenMP parallelization of the SCIARA Cellular Automata lava flow model: Performance analysis on shared-memory computers, Procedia Comput Sci 4 (2011) 271–280. https://doi.org/10.1016/j.procs.2011.04.029.

[10] A. Giordano, A. De Rango, R. Rongo, D. D'Ambrosio, W. Spataro, Dynamic Load Balancing in Parallel Execution of Cellular Automata, IEEE Transactions on Parallel and Distributed Systems 32 (2021) 470–484. https://doi.org/10.1109/TPDS.2020.3025102.

[11] A. Giordano, A. De Rango, D. D'ambrosio, M. Gil, D. Macri, X. Martorell, R. Rongo, G. Utrera, G. Mendicino, W. Spataro, Cellular Automata on a Multi-GPU Architecture: A Technical Overview, in: Proceedings - 2024 32nd Euromicro International Conference on Parallel, Distributed and Network-Based Processing, PDP 2024, Institute of Electrical and Electronics Engineers Inc., 2024: pp. 253–259. https://doi.org/10.1109/PDP62718.2024.00042.

[12] M. Sitko, L. Madej, Evaluation of code parallelization solutions in the static recrystallization cellular automata model, Procedia Manuf 15 (2018) 1879–1885. https://doi.org/10.1016/j.promfg.2018.07.201.

[13] F. Lin, M. Sitko, L. Madej, L. Delannay, Non-uniform Grain Boundary Migration During Static Recrystallization: A Cellular Automaton Study, Metall Mater Trans A Phys Metall Mater Sci 53 (2022) 1630–1644. https://doi.org/10.1007/s11661-022-06599-0.

The publication of this article was funded by the AGH University of Krakow 10.13039/501100007751