GEORG-AUGUST-UNIVERSITÄT GÖTTINGEN IN PUBLICA COMMODA SEIT 1737

HPS

Seminar Report

Performance Analysis using Score-P, Scalasca and Vampir

Claas Kochanke

MatrNr: 22646052

Supervisors: Jonathan Decker and Jack Opanga Ogaja

Georg-August-Universität Göttingen
Institute of Computer Science

May 13, 2024

Abstract

Implementing parallel software is significantly more complex than developing sequential applications due to the need for communication and synchronization. These complexities not only increase the risk of introducing bugs but also complicate performance analysis. To address these challenges, the tools Score-P and Vampir were utilized to analyze the performance of an implementation of Conway's "Game of Life" [Gar70], which was also developed throughout this report. Despite the intended inclusion of Scalasca, technical difficulties prevented its utilization.
Score-P and Vampir allow developers to methodically and in-depth examine their application's performance. This report presents an approach to performance analysis based on a *Performance Engineering Lifecycle*. The method demonstrated that structured performance analysis helps in both the identification and resolution of bugs, as well as optimization. It helped to uncover several bottlenecks related to memory usage, cache misses, and communication efficiency in the application developed and analyzed. Subsequently, multiple optimization approaches have been elaborated to address these issues, possibly enhancing the overall performance of the application.

Acknowledgements

I want to thank my supervisors Jonathan Decker and Jack Opanga Ogaja, who have guided me throughout this project. First, I would like to thank Jack Opanga Ogaja, who helped me find the right direction at the beginning of the project and introduced me to the topic.
Further, I would like to express my special thanks to Jonathan Decker. His commitment and strong support were crucial for this report. His willingness to engage intensively with my challenges enabled me to successfully complete this project and maintain my motivation throughout.

Declaration on the use of ChatGPT and comparable tools in the context of examinations

In this work I have used ChatGPT or another AI as follows:

- ☐ Not at all
- ☑ During brainstorming
- ☐ When creating the outline
- ☐ To write individual passages, altogether to the extent of 0% of the entire text
- ☐ For the development of software source texts
- ☐ For optimizing or restructuring software source texts
- ☑ For proofreading or optimizing
- ☑ Further, namely: - GitHub Copilot for faster programming

I hereby declare that I have stated all uses completely.
Missing or incorrect information will be considered as an attempt to cheat.

Contents

List of Figures

List of Listings

List of Abbreviations

CPU Central Processing Unit

GWDG Gesellschaft für wissenschaftliche Datenverarbeitung mbH Göttingen

SCC Scientific Compute Cluster

OpenMP Open Multi-Processing

MPI Message Passing Interface

OTF2 Open Trace format 2

GUI Graphical User Interface

1 Introduction

In computational science, the apparent speed of a Central Processing Unit (CPU) can mislead one into underestimating the complexity and scale of processing required for large-scale scientific problems. For perspective, a task that might overwhelm a human, such as processing 100 million lines of text, is trivial for current computational systems due to their processing speeds.
However, these systems encounter significant limitations when scaled to real-world scientific data. To illustrate, the human male body contains approximately 36 trillion cells [Hat+23]. If each cell is represented with a single bit - a considerable simplification - the resulting data would require approximately 4,000 petabytes of storage. This exceeds the storage capacities of most supercomputing centers, such as the Scientific Compute Cluster (SCC) at the Gesellschaft für wissenschaftliche Datenverarbeitung mbH Göttingen (GWDG), which has around 27 petabytes of combined storage.[1]
When compared to the execution times of the implementation discussed in this report and applying them to a model of 36 trillion cells, a single simulation step would take over 28 billion years. Even with a CPU that is a hundred times faster, single-core processors remain insufficient for handling complex, large-scale computations efficiently.
The limitations of single-core processors have led to the adoption of multi-core and multi-processor systems. It is common to see compute clusters with more than 100,000 cores. These systems split tasks across several cores and machines, which greatly speeds up the processing speed of tasks that can be done in parallel. However, the use of multi-core systems introduces new challenges in software development and optimization. The sheer complexity of these systems, compared to single-core machines can lead to more errors, require more development time and make it harder to fine-tune the performance. This highlights the importance of advanced tools for performance analysis like Score-P, Vampir, and Scalasca. They provide deep insights into the run-time behavior of applications and help developers to optimize their software more effectively.

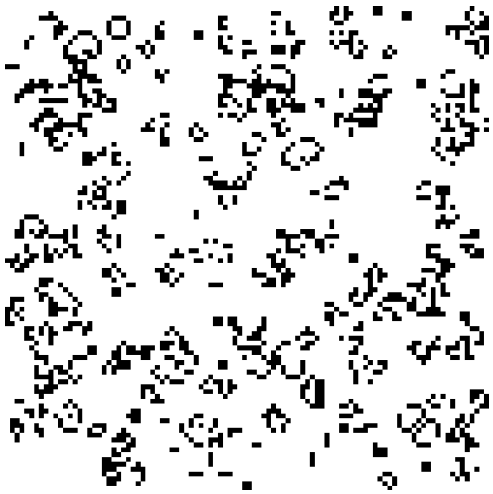


Figure 1: Example representation of "The Game of Life"

[1] `https://gwdg.de/hpc/systems/scc/` (visited on April 30, 2024)

1.1 Goals and contributions

This report focuses on developing a computer application for John Conway's Game of Life, a simple cellular automaton that provides an entry point into performance analysis of simulations (see example in Figure 1). It describes the process of developing a multi-core version using Message Passing Interface (MPI). The primary objective is to analyze the behavior of this parallel implementation to identify bugs and performance bottlenecks and give an overview of different aspects of the tools used for that.
The tools utilized in this report are Score-P and Vampir. They enable a systematic analysis and optimization of the source code. This report outlines the discovered bottlenecks and provides recommendations for improving the software's performance.

Unfortunately, various technical issues occurred during this project, significantly impacting the scope of the report. Originally Scalasca was also intended to be used. However, problems with this tool were only resolved in mid April 2024. Consequently this report focuses on Score-P and Vampir. Although a deeper investigation of the three tools was intended, the need to stay within the projects timeframe, combined with the time lost to troubleshooting, prevented a more comprehensive analysis.
A detailed discussion of the technical challenges can be found in Chapter 6 at the end of this report.

To achieve the aforementioned goals, this work has contributed in the following ways:

- Development of a single-core application of John Conway's "Game of Life".
- Development of a multi-core application of the same, explaining the workload distribution and how the problem of distributed memory was solved
- Implementation of a method to display large grids as JPEG files due to the impracticality of displaying them in text files or the console
- A basic performance analysis of the developed implementation
- Assisting in the resolution of issues to ensure proper functionality and usability of the software tools Score-P, Scalasca and Vampir.

1.2 Report Outline

This report begins by introducing the essential topics in Section 2, discussing the basics of performance analysis, the software involved, and a brief explanation of Conway's Game of Life. Section 3 outlines the hardware and software used and the necessary steps for running the simulation. In Section 4 the key elements of the code for both the single-core and multi-core versions are highlighted, along with the method used for visualization of the simulation. Section 5 introduces an approach to performance analysis and presents the outcomes of these analyses. Afterward, Section 6 explores the various challenges faced during the study. Finally, in Section 7, the report concludes with a brief summary and future perspectives.

2 Background

2.1 Performance Analysis

Performance analysis plays a crucial role in the development of parallel programs. It enables developers not only to design their software efficiently but also to examine its behavior and optimize it. Analysis of performance can be carried out at both hardware and software level, with this report focusing specifically on software analysis.
Performance analysis can be described by the *Performance Engineering Life Cycle*, which is shown in Figure 2, and goes through several phases: Prepare, Measure, Analyze and Optimize. First, the application is prepared, and the recording of data is set up (Prepare). The application is then executed, and the desired data is recorded (Measure). Once the data has been recorded, this information is analyzed (Analyze), after which improvements can be made to the application (Optimize). Then the cycle starts again from the beginning if required. Through this cyclical process, continuous improvements can be achieved and the performance of parallel programs can be maximized.

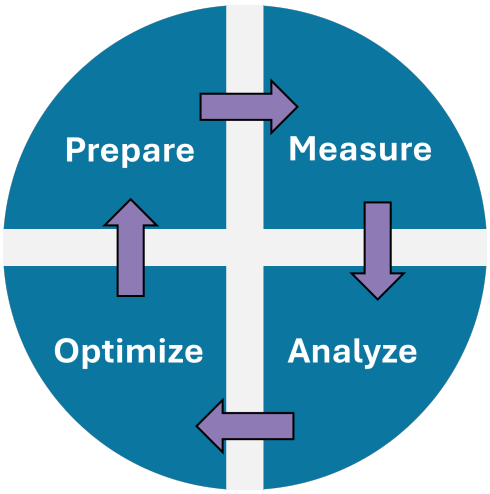


Figure 2: Performance Engineering Life Cycle[2]

However, it is important to balance the optimization efforts and not spend too much time optimizing every aspect. This is underscored by the Pareto Principle, which states that for many cases, 80% of the desired speedup can be achieved with 20% of the time invested, whereas achieving the remaining 20% speedup requires 80% of the time investment.

2.2 Performance Analysis Tools

This report uses several of advanced tools specifically designed for not only profiling and tracing but also for in-depth analysis of parallel applications. These tools are very useful for identifying performance bottlenecks and optimizing parallel programs. Below is a brief overview of the three tools used in this report, Score-P, Scalasca and Vampir.

[2]Inspired by: `https://youtu.be/0E4bCrdOWxc` (visited on April 30, 2024)

2.2.1 Score-P

Score-P stands for *Scalable Performance Measurement Infrastructure for Parallel codes*. It is a suite of software tools designed for profiling and tracing events throughout the execution of parallel applications. It supports a wide variety of data formats like Open Trace format 2 (OTF2), Cube4 and Opari2. This data can be used by various analysis tools such as Vampir and Scalasca. Score-P makes it possible to record exactly the data of interest through various code instructions, filters and other settings. This enables an uncomplicated and precise analysis of parallel programs.[3]

2.2.2 Scalasca

Scalasca[4] is an open-source software tool that enables in-depth performance analysis of parallel programs. It is mainly used for applications that use the programming interfaces Open Multi-Processing (OpenMP) and MPI. With the help of the tools provided by Scalasca, it is possible to find bottlenecks in the synchronization and communication of parallel applications. Scalasca provides the user with a command line interface to perform automated analyses and create reports. This information can then be visualized and examined using the "CUBE-QT" software. The code instrumentation can either be done by Scalasca itself or utilize the measurement infrastructure Score-P. The main difference to Vampir is that it is open source under the "New BSD open-source license" and can therefore be used by anyone.

2.2.3 Vampir

Like Scalasca, Vampir[5] is a collection of tools for analyzing the performance of parallel applications. It mainly uses the data collected by Score-P and visualizes it. Vampir offers a powerful user interface with diagrams, statistics and timelines. It allows zooming and scrolling in all displays of the recorded program and thus enables a dynamic and deep performance analysis. The Vampir Graphical User Interface (GUI) is very advanced, easy to use and allows for exploration of the behavior of the parallel application, whereas Scalasca is mainly used for automatic analysis. In contrast to Scalasca it is closed source and the licensing fees are several thousand euros.

2.3 Game of Life

The "Game of Life" [Gar70] by the British mathematician John Horton Conway is a cellular automaton. It simulates living organisms using a checkerboard and cells. The Game is a zero-player game where the initial setup of the board specifies the process of the game. In this game, a set of simple rules allows for the observation of birth, death and survival of patterns over several generations.

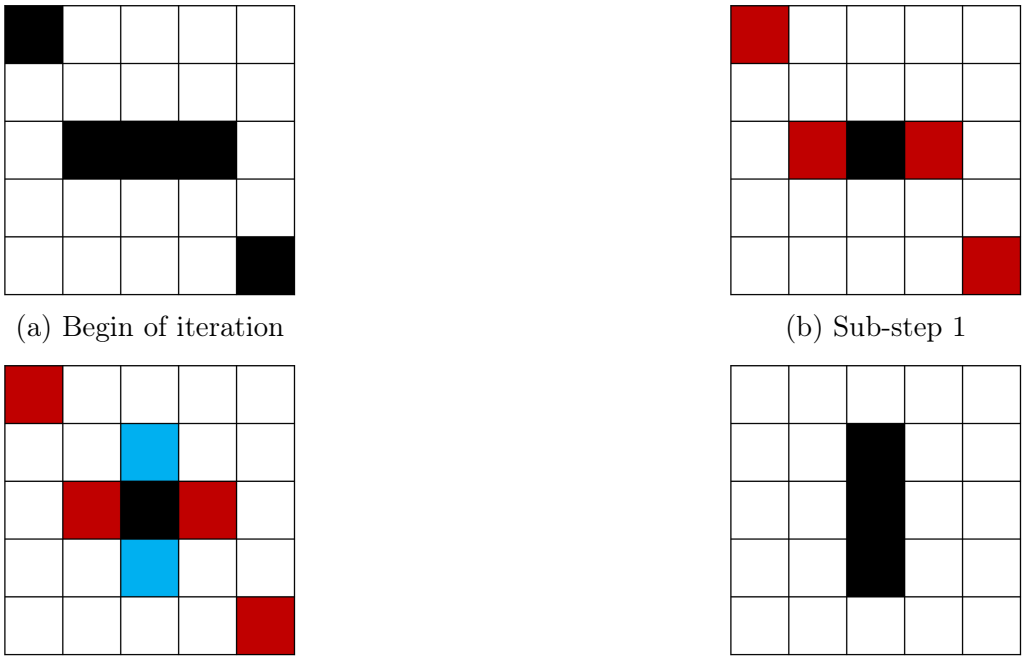
The "Game of Life" is played on a grid of square cells that are either alive or dead. Each cell interacts with its eight neighbors and follows three simple rules.

- Survival: A cell with two or three neighbors survives to the next round.

[3] `https://docs.gwdg.de/doku.php?id=en:services:application_services:high_performance_computing:performance_engineering_and_analysis:score-p` (visited on April 30, 2024)

[4] `https://www.scalasca.org/scalasca/about/about.html` (visited on April 30, 2024)

[5] `https://vampir.eu/` (visited on April 30, 2024)



(a) Begin of iteration (b) Sub-step 1

(c) Sub-step 2 (d) End of iteration

Figure 3: Game of Life - Single Iteration Visualization
Black cells are alive. Red cells will die. Blue cells will be born

- Death: A cell with less than two neighbors dies of isolation and a cell with more than three neighbors dies of overpopulation
- Birth: A dead cell that has exactly three live neighbors becomes alive in the next round.

2.3.1 Example visualization

Figure 3 provides a detailed visualization of a single iteration within a 5x5 board example for Conway's Game of Life. Panel 3a displays the initial configuration, where all black cells represent living cells. According to the rules previously outlined, four cells experience death due to underpopulation. These are illustrated in red in panels 3b and 3c. At the same time, two new cells appear, shown in blue in panel 3c. The resulting configuration after this iteration is illustrated in panel 3d. This process shows how Conway's rules work in a controlled setting, highlighting how simple rules can lead to complex behaviors over time.

3 Method

This section provides a comprehensive overview of the hardware and software used for this practical. It details the hardware configuration, the software environment and the steps necessary for executing the provided software.

3.1 Hardware Environment

As of April 28, 2024, the specifications of the SCC are as follows:

- **Compute Nodes**: The system is made up of 410 heterogeneous compute nodes.
- **Total CPU Cores**: The cluster consists of 18,376 CPU cores. These cores are distributed among several models, including Xeon Platinum 9242, Broadwell Xeon E5-2650 v4, Haswell Xeon E5-4620 v3, and Xeon Gold 6252.
- **Total RAM**: The aggregated memory available across the cluster totals 99 terabytes.
- **Node Configuration**: The configuration of individual nodes within the cluster is highly variable, ranging from 4 to 48 CPU cores per node, with RAM varying from 32 GB to 2048 GB per node.
- **Interconnection**: The Nodes are connected with a 100 GBit/s Omni-Path

For more detailed information on the current cluster architecture and node-specific configuration, refer to `https://gwdg.de/hpc/systems/scc/`.

3.2 Software Environment

The software configuration used in this report includes the following components:

- **Operating System**: Scientific Linux 7.9
- **MPI Library**: OpenMPI Version 4.1.6
- **Performance Monitoring**: Score-P 8.3
- **Visualization Tool**: Vampir 10.0.1

The components can be used on the cluster utilizing the module system.

3.3 Simulation Setup

The source code for this report is accessible via the following GitHub repository link: `https://github.com/C1aas/SCAP-Seminar-MPI`. To prepare the program for execution, compile it by executing the script

```
$ load_and_compile.sh
```

, which sets up the required environment and compiles the source code. After the compilation, the simulation can be run by executing

```
$ run.sh
```

Parameters such as the size of the simulation and the number of CPU cores can be modified within the *run.sh* script.

4 Implementation

This Section gives an overview of the implementation of Conway's Game of Life. Starting with the single-core version and then progressing to the multi-core implementation. As part of this, the distribution of the work load and the general communication pattern is explained.

4.1 Single-Core

The implementation of the single-core version is straightforward, as illustrated in Listing 1. Initially, a grid is created and requires initialization. This can be achieved by manually configuring a specific pattern or by randomly initializing the grid with a specific density, the latter of which is used in this approach. Subsequently, the function 'updateGrid' is invoked a specific amount of times to simulate the grid changes based on the predefined rules mentioned in 2.3. A detailed but simplified version of the 'updateGrid' function can be found in Listing 4 in the appendix. Finally, the resulting grid can be output for review.

```
1  void gameLoop(GameConfig cfg) {
2      int height = cfg.grid_size;
3      int width = cfg.grid_size;
4      unsigned char** grid = createGrid(height, width);
5      initializeGridRandom(grid, height, width, DENSITY);
6
7      printGrid(grid, height, width);
8      for(int i = 0; i < cfg.total_iterations; i++){
9          updateGrid(grid, height, width);
10     }
11     printGrid(grid, height, width);
12     freeGrid(grid, height);
13 }
```

Listing 1: Single-Core - Code Excerpt of the Game Loop

4.2 Multi-Core

To implement the multi-core version, the workload must be divided, and MPI is used to transfer data between processes. In this approach, the board is divided into multiple parts line by line. Each worker process can then simulate its segment. However, a trivial division is not feasible. To simulate a section, each worker also requires information about all adjacent cells.
Therefore, each worker is not only sent his part of the board but also the adjacent lines as shown in Figure 4 (Start and Iteration 0). Once a worker has computed its section, it is missing the newly simulated neighboring lines. These missing lines, highlighted in white in Iteration 1 of the Figure, must be received by the neighboring workers.
Communication during the simulation must therefore only take place with the respective neighboring workers.

In the end, all workers send their sections of the grid back to the main process for output, as visualized in 'End' in Figure 4.

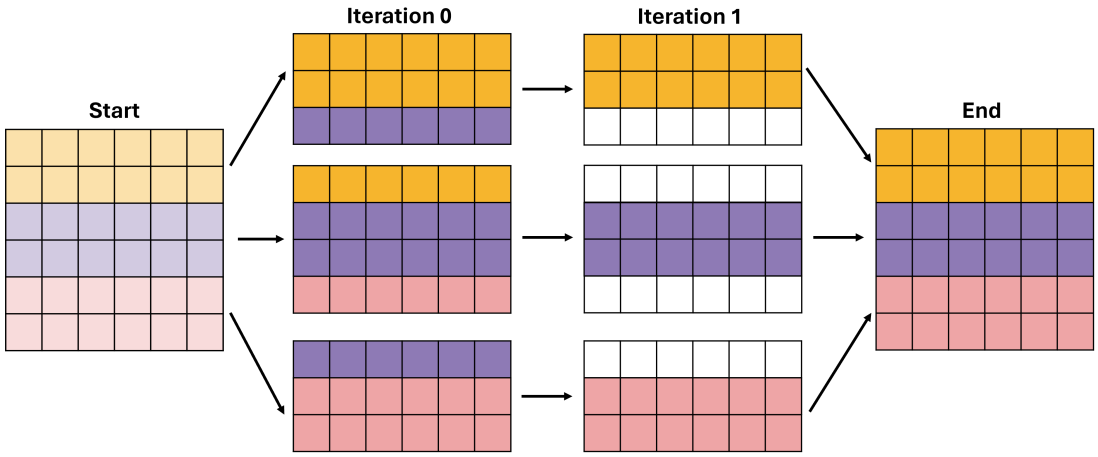


Figure 4: Game of Life - Work Distribution Visualization

Listings 2 and 3 illustrate a simplified version of the code used for the implementation. Initially, the main process generates the grid and distributes the sections and a configuration to the worker processes. Following this, it waits to receive the final grid parts from each worker, as shown in Listing 2.

```
unsigned char** grid = createGrid(height, width);
initializeGridRandom();
distributeAndSendWorkerConfig(...);
sendGridPartsToWorkers(...);
write_jpeg_file("initial_grid", ...);
receiveGridParts(...);
write_jpeg_file("final_grid", ...);
```

Listing 2: Multi-Core - Simplified code of the main process

Upon receiving the initial grid and its configuration, the workers enter a loop that is similar to the single-core code version. The key difference, detailed in Listing 3, is that they continuously send the updated grid rows to the neighboring workers and receive updates from them.

```
WorkerConfig cfg = receiveWorkerConfig();
receiveInitialGrid(&cfg);
for(int i = 0; i < cfg.total_iterations; i++) {
    updateGridWithLimit(cfg);
    sendandReceiveUpdatedGridRows(cfg);
}
sendGridToMain(cfg);
```

Listing 3: Multi-Core - Simplified Code of the worker process

4.3 Visualization

As mentioned in the introduction, challenges arise when dealing with large grids, such as 10,000 x 10,000 cells. Visualizing such massive grids in the console or through text files is impractical due to their size. Therefore, a method was implemented using the 'libjpeg'[6] library, following the *example.c* provided in the repository.
In this approach, each grid cell is represented as a pixel in an image. Cells that are 'alive' appear as black pixels, while 'dead' cells appear as white pixels. This technique allows for a more practical visualization of large grids, especially since nearly any computer capable of displaying images can utilize this visualization.
For an example of the output produced by this method, refer to Figure 1 in the introduction section of this report.

[6] `https://github.com/LuaDist/libjpeg` (visited on Mai 1, 2024)

5 Performance analysis

To assess and enhance the performance of the implementation, the "Performance Engineering Life Cycle", as outlined in Chapter 2.1 of the referenced text, is utilized. This provides a structured approach to performance optimization, which is crucial for multi-core systems. The analysis is organized into the four main stages Prepare, Measure, Analyze, and Optimize.
The subsequent chapters examine the details of each phase involved in the analysis. It is important to note that this method is just one of many possible approaches. Furthermore, it presents a basic performance analysis and does not address every detail. As mentioned earlier, certain technical issues arose during the process, which limited the depth of this report. Consequently, the chapters may not be as detailed as initially desired.

5.1 Preparation

For effective performance analysis with Score-P, proper configuration is required. This includes several essential steps to prepare for the data collection during the measurement phase.
First, the application must be compiled with special commands to integrate Score-P's monitoring tools. Score-P offers various wrappers to facilitate this process. The command utilized in this report is:

```
$ scorep-mpicc <file> -g
```

This embeds the necessary features for data collection into the application's executable file. Additionally, adding the '-g' flag embeds debug information, enhancing the clarity and detail of performance analysis.

Next, certain environment variables must be defined before running the application. These settings direct Score-P on what data to collect. The basic configuration needed for this report is:

```
export SCOREP_ENABLE_PROFILING=true
export SCOREP_ENABLE_TRACING=true
```

Activating these variables initiates profiling and tracing, essential components for an in-depth analysis with Vampir. Additional variables can be adjusted to refine the data collection process. The complete list of configurable options and their descriptions are presented in the Score-P documentation[7].
Additionally, configuring a filter to exclude certain functions from being recorded can optimize the data collection, focusing on the most relevant parts of the application's source code. This helps to manage the volume of collected data and enhances analysis efficiency. An example filter file might look like this:

[7] https://scorepci.pages.jsc.fz-juelich.de/scorep-pipelines/docs/scorep-6.0/html/scorepmeasurementconfig.html (visited on May 2, 2024)

```
SCOREP_REGION_NAMES_BEGIN
    EXCLUDE
        max
        min
        debugPrint
        current_time_millis
```

To activate this filter, the corresponding environment variable must be set as shown:

```
export SCOREP_FILTERING_FILE=<filter-file>
```

While these steps form the foundation for basic performance analysis, Score-P also supports more advanced instrumentation options. These are not covered in this basic setup but can be explored in the documentation, particularly in the Application Instrumentation" section[8].

5.2 Measurement

Following the setup, the measurement process is simply executing the program with the desired parameters. It is crucial to not execute any computation on the login node. Instead, utilize the batch system for running computations. The command for executing an application using MPI is structured as follows:

```
$ mpirun -np <number-of-processes> <binary-file>
```

Upon execution, Score-P automatically creates a folder containing the tracing and profiling data. Due to the distributed nature of the system, the creation of the output files might be delayed. Therefore, it is advisable to wait a few seconds after the execution has finished until the files fully appear and ensure they are complete. Details on the execution setup, including specific commands and configurations, can be found in the run.sh script located in the GitHub repository[9].

5.3 Analysis

The analysis is structured into two main parts. The first part, the *Data Validation* phase, involves measurements taken using timing and *scorep-score*. This phase focuses on confirming that the data is complete and accurate, ensuring that all necessary data is accurate and complete. Such verification prevents the need to restart the analysis due to missing or inaccurate data.
The second part, *Visual Analysis and Interpretation*, uses Vampir to perform a detailed visual interpretation and analysis of the data.
This two-step approach makes the overall analysis process more efficient, ensuring an effective evaluation of the application without unnecessary duplication of work.

[8] `https://scorepci.pages.jsc.fz-juelich.de/scorep-pipelines/docs/scorep-6.0/html/instrumentation.html` (visited on May 2, 2024)

[9] `https://github.com/C1aas/SCAP-Seminar-MPI` (visited on May 2, 2024)

5.3.1 Data Validation

Before beginning with an in-depth analysis, it is advisable to first test the general execution times to ensure that the program is functioning as expected. It is expected that the multi-core version outperforms the single-core version in terms of speed. The execution times for the single-core version, as illustrated in Figure 5, show a quadratic increase in the time required for each iteration relative to the size of the grid. In scenarios with a grid size of 100,000x100,000, the duration for a single iteration is 252 seconds. Consequently, the time to complete one hundred iterations would require approximately seven hours. The extensive time demands highlight the inefficiency of the single-core implementation for handling large-scale computations, emphasizing the necessity for a more capable multi-core version.

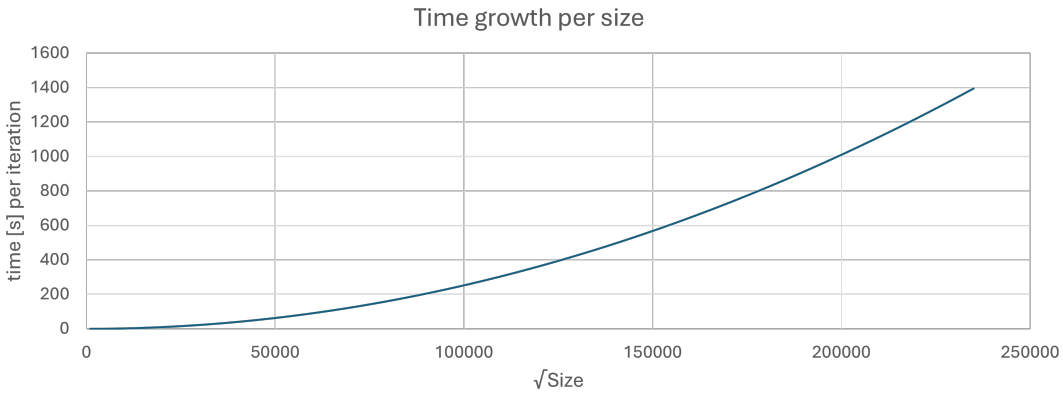


Figure 5: Game of Life - Time Per Iteration as a Function of Grid Size

Following that, the execution times from the multi-core version are measured and compared with the extrapolated times from the single-core version measurements. The measured and expected values are shown in Figure 6.

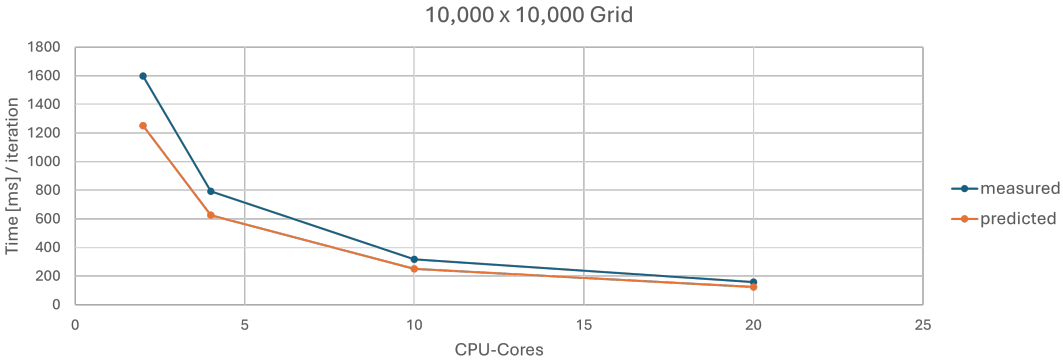


Figure 6: Game of Life - Multi-Core Scaling Efficiency by Number of Cores

Upon examination of the graph, it is observed that the multi-core version runs approximately 20% slower than predicted by extrapolation. This represents an overhead of about 20%. However, relying on simple time measurements in a heterogeneous system such as the SCC can be misleading, due to a large number of variables that influence these times. Nonetheless, such data can provide a general overview and can help in understanding the

performance of the application.

Following the timing analysis it is valuable to verify the recorded data using the command line tool *scorep-score*. An example of the output from this tool is illustrated in Figure 7.

type	max_buf[B]	visits	time[s]	time[%]	time/visit[us]	region
ALL	2,022,425	86,889	395.98	100.0	4557.29	ALL
MPI	1,501,864	44,732	67.49	17.0	1508.76	MPI
MEMORY	520,052	40,058	0.14	0.0	3.57	MEMORY
COM	2,756	1,066	0.15	0.0	143.90	COM
USR	2,626	1,022	328.19	82.9	321124.72	USR
SCOREP	41	11	0.00	0.0	280.25	SCOREP
MPI	890,000	11,800	0.04	0.0	3.38	MPI_Irecv
MPI	611,708	20,028	0.44	0.1	22.10	MPI_Send
MEMORY	260,026	20,029	0.09	0.0	4.39	malloc
MEMORY	260,026	20,029	0.06	0.0	2.75	free
MPI	61,183	10,028	20.35	5.1	2028.94	MPI_Recv
MPI	17,800	1,800	0.04	0.0	19.51	MPI_Isend
COM	2,600	1,000	0.01	0.0	12.34	sendandReceiveUpdatedGridRows
USR	2,600	1,000	326.26	82.4	326256.02	updateGridWithLimit
MPI	2,600	1,001	33.43	8.4	33392.94	MPI_Waitall
...						
...						

Figure 7: Example Output of *scorep-score* for a 10K x 10K Grid over 100 Iterations

The simulation was run using ten workers, over one hundred iterations and with a grid size of 10,000 x 10,000. Therefore, it logically follows that the *'updateGrid'* function should be called 1,000 times. However, there were prior implementation versions where this was not the case. This highlights the usefulness of this tool for not only performance analysis but also for debugging. Additionally, the line labeled "MPI Region" in the table shows, that MPI took up 17% of the total runtime. This is roughly in line with the overhead of about 20% shown and calculated in Graph 6.
Furthermore, there were around 45,000 MPI calls compared to only 1,000 update calls, averaging 45 MPI calls for every update call. Even though the 45,000 calls include the initial distribution of the grid, this amount of message passing for each update call is excessive. It is likely an area that can be optimized.
Using the table from *scorep-score*, the recorded data can be further verified as shown in the examples.

5.3.2 Visual Analysis and Interpretation

After verifying the data, Vampir can be used for visual analysis. Vampir is a sophisticated and comprehensive software tool, which could be the subject of an entire master's thesis to explain its features. Therefore, only the basic functionalities will be discussed here. The Vampir manual[10] covers a detailed overview.
Figure 8 provides a screenshot of the Vampir GUI. The upper section of the interface consists of the *Chart Toolbar* and the *Zoom Toolbar*. The *Chart Toolbar* allows users to add custom graphs and access additional menus. The *Zoom Toolbar* enables users to adjust the timeframe displayed across the GUI, such as narrowing the view to the initial second of execution.

[10] `https://vampir.eu/tutorial/manual` (visited on May 5, 2024)

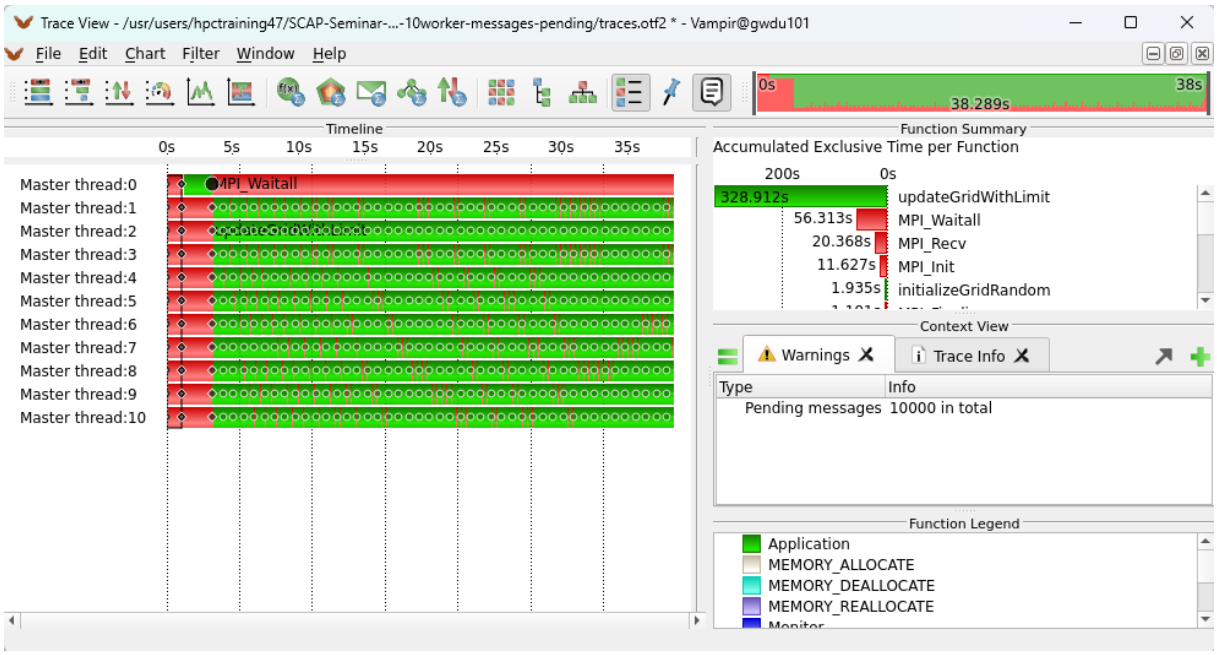


Figure 8: Vampir - GUI Overview

On the left side of the interface is the *Timeline*, which displays MPI calls in red and application code in green for each thread. This area also supports zooming capabilities and illustrates the communication flow among processes. On the right, the *Function Summary* panel lists the accumulated time spent in each function. Below this, the *Context View* provides trace information and warnings, among other potential data windows. This particular screenshot originates from a version of the application that contained a bug; it failed to wait for all messages to be received but did not crash. Consequently, this feature is especially valuable for debugging to ensure the application functions correctly. In the lower right corner, the Function Legend clarifies which color represents each function type.

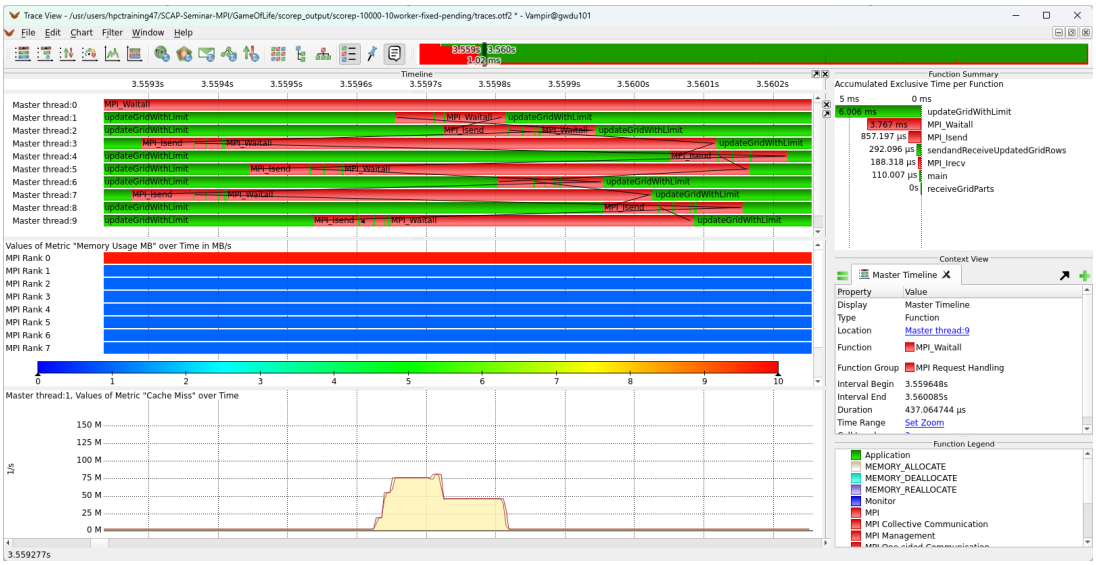


Figure 9: Vampir - Example with Memory Usage and Cache Misses

Figure 9 presents a refined view, with custom graphs, focusing on a timeframe of one

millisecond, that was adjusted using the *Zoom Toolbar*. The *Timeline* provides a detailed view of the communication following each *updateGrid* call, represented by black lines. It's essential to verify that the communication works as expected.
Beneath the Timeline, two graphs are inserted to illustrate specific performance metrics. The first graph displays the memory usage across the processes. Logically, the main process uses significantly more memory with approximately 10 MB, as it is responsible for maintaining the entire grid, whereas the worker processes each utilize approximately 1 MB of memory.
The second graph below tracks the cache misses in both L1 and L2 caches over time for the first worker process. The baseline cache miss rate is at 2 million misses. However, this rate increases to over 75 million during communication with adjacent worker processes. This spike is expected due to the processor's need to reload newly transmitted data into the cache. Nevertheless, a base cache miss rate of 2 million, with peaks reaching 75 million, indicates inefficiency and highlights an opportunity for optimization.

5.4 Optimization

Due to the technical difficulties and the associated time delays, as highlighted in the introduction and further discussed in Section 6, this subsection is concise. It presents potential optimizations without their actual implementation.

1 Bit Cell Representation: Currently, each cell uses 1 byte or 8 bits. However, since a cell's state — dead or alive — can theoretically be represented by just one bit, it would be valuable to explore if packing multiple cells into a single byte could enhance performance. This approach would reduce memory usage and likely decrease cache misses. It could also introduce overhead due to the need for unpacking bits.

Only Store Life Cells: Storing only living cells might improve the efficiency, as these are the only cells that can cause a change. Excluding dead cells reduces data volume, but positional information for each living cell must then be recorded. Investigating whether this strategy increases or decreases performance could provide valuable insights.

Enhancing Communication Efficiency: To improve the efficiency of communication, it can be beneficial to implement the following strategies:

- *Selective Messaging*: Include only living cells in transmitted messages, excluding dead cells. This reduces the size of the messages, making the communication process more streamlined.
- *Conditional Messaging*: Send messages to neighboring cells only when there are updates or changes to report. This ensures that communication is relevant, avoiding the transmission of redundant information. In cases where there are no changes, a simple acknowledgment signal, or 'ping', can be sent to inform neighbors that there are no updates.

These enhancements aim to optimize the use of resources and improve the overall efficiency of the communication system while maintaining simplicity in implementation.

Implementation of Statistics: The current algorithm iterates over each cell and checks all of its neighbors. However, in areas with large aggregations of "dead" cells, these checks

are redundant as "dead" cells cannot alter without "live" cells around. In theory, only living cells and their neighbors need to be checked, as only live cells can introduce a change. Eliminating these unnecessary checks, the CPU load could be significantly reduced. It is also likely that this will reduce the number of cache misses.

Dynamic Load Balancing: Reducing CPU usage might result in uneven workload distribution across CPU cores. Implementing dynamic load balancing or a similar mechanism could help in distributing the load more evenly. It would be interesting to investigate whether such balancing introduces overhead and whether the potential benefits justify this cost. Exploring different strategies for dynamic load balancing could also be very insightful.

5.5 Results

The performance analysis methodology used ensured data quality and also proved valuable for debugging purposes. Utilizing tools such as Score-P and Vampir, deeper insights into the application were gained. Several bugs, such as one where messages were not being delivered correctly, were identified and resolved. Through analysis using Vampir the need for optimized RAM usage and reduction of cache misses arose. Additionally, the amount of messages sent via MPI appeared excessive.
These insights highlighted areas requiring optimization and several promising strategies were identified that could significantly enhance the application's performance.
For future work, it would be beneficial to implement these optimizations. Continuing with the "Performance Engineering Life Cycle", the next phase should involve evaluating the effectiveness of the optimizations to assess their impact. This would validate the theoretical findings and help in fine-tuning the application for optimal performance.

6 Challenges

In 2023, the SCC received an update to its software and modules. However, it was not verified that all tools were functioning properly after the update. In addition, the documentation for Score-P remained outdated. This led to several issues with the tools used for this practical on the cluster. Specifically, the Score-P tool did not work due to multiple libraries not being linked and incorrect paths in the documentation. This affected dependent tools like Vampir and Scalasca, which rely on the data generated by Score-P. Furthermore, Scalasca did also miss a lot of library and path links and the Cube-GUI, used for visualization, was entirely missing on the SCC.
After reporting the issue, the Cube-GUI was installed. To resolve the problems with Score-P and Scalasca, a temporary fix was to manually find and link the required libraries until no more errors occurred. Due to limited knowledge of the SCC and its software, significant effort was required to manually locate the correct paths. In response to these issues, a support request was submitted with detailed information on the situation.
The support team responded with a short explanation and a solution. The main problem on the SCC is that no module files are generated for dependent packages, resulting in them not being loaded. This will be fixed on the new Software stack that is coming to the SCC later in the year.
As an interim solution, the support team manually created the modules for the dependent libraries needed for Score-P, Scalasca and Vampir. With these adjustments, all three tools are now functioning correctly. This is especially useful for the course "Practical Course on High-Performance Computing" in which students also need to test and analyze their applications.

6.1 Personal Recommendations

Based on these experiences, I recommend that supervisors test the software that students rely on for their research in advance. While I understand that not every potential issue can be tested, ensuring basic functionality especially, after updates, can prevent disruptions to research activities. This approach would help in maintaining a reliable environment on the cluster and also support the academic and research objectives of both students and employees of the GWDG.

7 Conclusion

In this report, a procedure for performance analysis with Score-P and Vampir was developed, which is easy to understand and comprehensive, especially for beginners. In addition, the cluster's software system has been improved, resulting in an improved computational environment for academic and research activities.
Within this report, a methodology for performance analysis utilizing Score-P and Vampir has been shown. This methodology is both accessible and detailed, making it particularly suitable for novices in the field. Furthermore, enhancements to the software system of the SCC have been implemented, improving the infrastructure for academic and research purposes.

The next and final Chapter concludes by outlining potential areas for future research and further development opportunities that can follow this report.

7.1 Outlook

There are several promising approaches as to how this project can be continued, each outlined below.

Usage of Scalasca: Now that Scalasca is working properly, it would be interesting to compare Scalasca with Vampir. It would be valuable to explore the advantages of using Scalasca and whether combining Scalasca and Vampir could enhance performance analysis.

Source Code Optimizations: There are several improvements to the source code that can be made. Such as reducing memory usage, cache misses and the amount of data sent. But also inducing statistics to only update cells that can change. With that one might need to use dynamic load balancing to evenly distribute the data during the simulation. As section 5.4 shows the optimizations it would be interesting to implement them and continue with the performance engineering cycle to assess the effectiveness of those optimizations.

Utilizing GPUs: Another possible and interesting approach is using GPUs to perform the simulation. They can handle many operations at once and could significantly speed up the calculation of the simulation. This would allow for even larger and more complex simulations.

In summary, the mentioned approaches can enhance the application and can achieve faster simulation speed. After all, *"Computation time is money"*. However, it is crucial to balance optimization efforts; as Donald Knuth wisely stated, *'If you optimize everything, you will always be unhappy'*.

References

[Gar70] Martin Gardner. “Mathematical Games - The fantastic combinations of John Conway’s new solitaire game"life"”. In: *Stanford University Course Material* (1970). Accessed: 2024-04-27, pp. 120–123. URL: https://web.stanford.edu/class/sts145/Library/life.pdf.

[Hat+23] Ian A. Hatton et al. “The human cell count and size distribution”. In: *Proceedings of the National Academy of Sciences* 120.39 (2023), e2303077120. DOI: 10.1073/pnas.2303077120. eprint: https://www.pnas.org/doi/pdf/10.1073/pnas.2303077120. URL: https://www.pnas.org/doi/abs/10.1073/pnas.2303077120.

A Code samples

```
void updateGrid(unsigned char** grid, int height, int width) {
    unsigned char** newGrid = copy(grid);
    for (int i = 0; i < height; i++) {
        for (int j = 0; j < width; j++) {
            int liveNeighbors = 0;
            // Check all eight neighbors and count them
            for (int y = -1; y <= 1; y++) {
                for (int x = -1; x <= 1; x++) {
                    // Skip the cell itself
                    if (y == 0 && x == 0) continue;
                    int ni = i + y;
                    int nj = j + x;
                    //check bounds
                    if(not in bounds) continue;
                    if(grid[ni][nj] == 1) {
                        liveNeighbors ++;
                    }
                }
            }
            // Apply the Game of Life rules
            if (grid[i][j] == 1) { // Currently alive
                if (liveNeighbors < 2 || liveNeighbors > 3) {
                    newGrid[i][j] = 0; //dies
                } else {
                    newGrid[i][j] = 1; //stays alive
                }
            } else { // Currently dead
                if (liveNeighbors == 3) {
                    newGrid[i][j] = 1; // Dead cell becoming alive
                }
            }
        }
    }
}
```

Listing 4: Single-Core - Simplified Code for the updateGrid Function