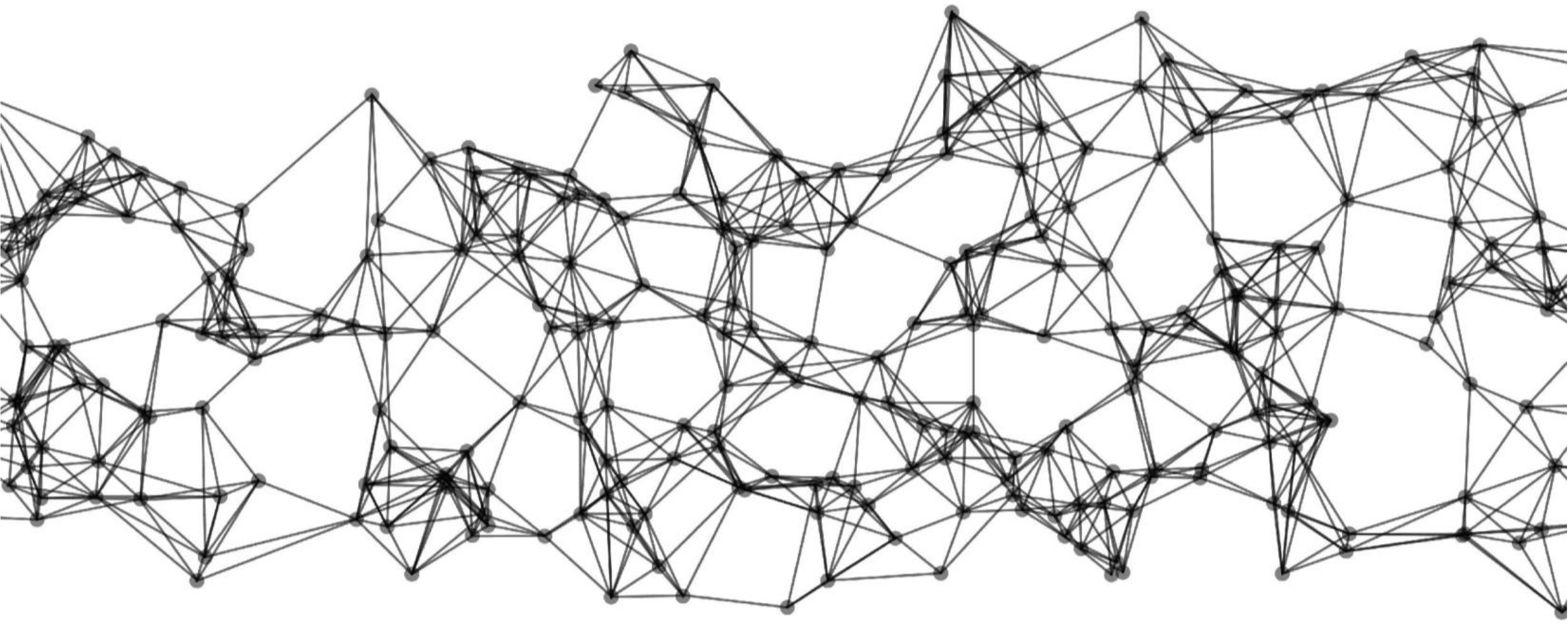
# Parallel Power Flow Algorithm for Large – Scale Distribution Network Topologies

Master's Thesis

Zurich, January 2017

| Author | Advisor | Professor |
|---|---|---|
| Patrick Lieberherr | Dr. Stephan Koch | Prof. Dr. Gabriela Hug-Glanzmann |
| pa.lieberherr@outlook.com | *CEO Adaptricity* | *Swiss Federal Institute of Technology* |
| | | *Power Systems Laboratory* |



# Abstract

The growing penetration of renewables in the electric power grid necessitates fast and accurate power flow algorithms. The calculation of big grids with thousands of nodes pushes well-known algorithms to their limits. In this thesis, an iterative method to solve big grids by splitting them into smaller subgrids is presented. The interaction with adjacent networks at the splitting points is accomplished by introducing grid elements that are temporarily added to the subgrids. As these elements remove data dependencies among the subgrids, full parallelization can be achieved using any traditionally sequential power flow algorithm in parallel on the individual subgrids. Experiments with a Java implementation showed that even few iterations yield very accurate results. Compared to a solver using the Sweeping algorithm, speedups of a factor of 9.25 were measured when simulating annual scenarios with medium sized topologies with a temporal resolution of 15 minutes. For bigger grids, even higher speedups can be measured. The calculation of a grid with 47’043 buses on a standard personal computer takes only 632 milliseconds with the parallel solver using the Sweeping algorithm as an underlying powerflow solver. In comparison, the regular sweeping solver requires 83 seconds to accomplish the same result. By choosing the degree of parallelization, either speed or memory consumption can be optimized. Making the tradeoff in favor of low memory consumption, reductions of memory footprint by factor 29 could be measured when computing large topologies.

# Contents

# Outline

This project aims to investigate performance gains of parallelizing power flow algorithms. The goal is to increase computation speed for big grids while maintaining the accuracy of the results. In this approach the grid model is split into distinct subgrids, each is repeatedly solved in parallel until the result converges. The project was conducted at Adaptricity, an ETH spin-off.

Chapter 1 explains why fast power flow algorithms will be needed in the future and gives an overview of the different power flow algorithms. Chapters 2 and 3 reveal how the computation was parallelized. Chapter 2 focuses on the concept in theory, whereas Chapter 3 shows an actual implementation in Java. Chapter 4 explains how the validation and performance testing of the parallelized algorithms were conducted, the results are presented in Chapter 5. Finally, Chapter 6 concludes the topic by discussing the results and suggesting next steps.

# 1 Introduction

## 1.1 Why do we need fast power flow algorithms?

To ensure a reliable supply of electrical energy, voltages and currents in an electrical network must be kept within certain limits. Electric power and transmission system operators use simulation tools to monitor their current network and to plan future extensions and modifications.

The computation times of the algorithms performing these simulations depend heavily on the size of the network. In practice, networks can be simulated up to a couple thousand nodes using conventional methods. Distribution system operators approached these limitations by simplifying the lowest grid level. This can be achieved by aggregating the power consumption of entire neighborhoods and attaching the aggregated load to a single node, which is connected to the substation that feeds the neighborhood. With this technique, the number of nodes in the simulation can be reduced significantly[1].

This simplification is accurate because the aggregated power consumption of residential customers is well understood. An increasing number of smart grid elements will complicate prediction. The fuzzy term "smart grid elements" refers to a variety of upcoming technologies like demand side managed household appliances, photovoltaic installations, domestic energy storage, electric cars whose batteries are part of virtual power plants providing ancillary services and so on. With a higher penetration of these technologies there will be a need to simulate the grids in higher detail to meet the requirements of the next generation of customers.

## 1.2 How are electric grids modelled?

Electric power networks are usually modelled as undirected graphs with nodes and edges. Depending on the voltage level, nodes and edges represent different things. Most of the nodes represent connection points to energy consumers, power plants or substations. Edges repre-

[1] Another reason why this simplification is often made is the fact that the grid operators do not have the grid data of the lowest voltage levels available digitally.

sent cables or overhead lines connecting these nodes. Nodes typically have loads or generators attached to them, representing the power injected or withdrawn from the network. Simulation tools like NEPLAN formulate a mathematical problem given the topology, the voltages, per-node power consumption, line and transformer parameters and so forth. The resulting set of equations are solved numerically using specialized algorithms. These algorithms calculate voltage magnitude and angles at every node. Other quantities like line currents and power flow can be derived from the node voltages.

## 1.3 Current Power Flow Algorithms

The main two algorithms used in power flow computations are Newton Raphson and Gauss Seidl. Both are well-known iterative methods to numerically solve sets of equations. They were developed in the 17th and 19th centuries. A more recent algorithm called Sweeping algorithm was published in 1990 by [1] and implemented in the year 2014 by F. Teng [2] for Adaptricity. All three algorithms can solve radial as well as meshed networks. Since they are iterative, a convergence criterion is required. For Newton Raphson and Gauss Seidl the termination tolerance on active and reactive power must be smaller than a certain predefined threshold. The Sweeping solver converges when successive bus voltage magnitudes (in per unit) are smaller than a threshold. The sweeping criterion applies to radial grids without PV generators as investigated in this project. The convergence criterion for other grids can be found in [2].

The main difference of the three algorithms is their performance. Consider a town of 8000 inhabitants in an urban area[2]. The city's electric power grid model consists of 2444 nodes, 2406 lines and 37 transformers. Table 1 shows the simulation performance with the three algorithms.

| **Algorithm** | **Computation Time** | **Iterations** |
|---|---:|---:|
| Newton Raphson | 130.421 s | 3 |
| Gauss Seidl | 3213.226 s | 3715 |
| Sweeping | 0.086 s | 3 |

**Table 1** *Execution times of the power flow algorithms for a grid with 1194 nodes*

Performance generally depends on the size and topology of the grid. The Sweeping solver usually performs best while Gauss Seidl is the slowest. This project will therefore focus on parallelizing the Sweeping algorithm.

[2] 200 inhabitants per km$^2$

## 1.4 Motivation to parallelize power flow algorithms

The Sweeping algorithm is already quicker than the other algorithms. The question arises what the benefit of even faster algorithms would be. Consider a much bigger area like the city of Zurich. Zurich has a population of roughly 400'000 inhabitants. A grid representing[3] Zurich has 5452 nodes, 4879 lines and 572 transformers. A single computation with the Sweeping algorithm takes 7.03 seconds on average.

Grids are often simulated over a period of time to simulate day/night effects, weather changes, different usage on weekends or even seasonal effects such as the higher energy consumption in winter. Usually, these simulations are performed with a temporal resolution of 15 minutes, meaning that the load and production are assumed to be static within 15 minutes, then they are changed and the grid is recomputed to simulate the following 15 minutes. This resolution requires 4 simulations per hour, 96 per day or 35'040 for an entire year. Hence, simulating Zurich for a year with the sweeping algorithm would then take several days. More renewables in the electric power grid may require an increase in temporal resolution to be able to simulate fast changes in wind and solar intensity more accurately. Faster algorithms would therefore be useful. Speed is not always the bottle neck when using power flow solvers. Memory consumption must be considered too. When simulating big grids the algorithms sometimes fail because the computer runs out of memory. When parallelizing the calculation, a solver would only need to compute much smaller grids. The memory consumption of all solvers together would still be high. But since there are no data dependencies between the subgrids, they can be computed one by one. Since the solvers for the subgrids can be deleted after calculation of the subgrid, parallelization can prevent the system running out of memory.

## 1.5 Previous Work

Previous attempts to parallelize power flow calculation focused on Newtons method. One attempt increased the efficiency of individual mathematical steps of the solver [3]. Speedups of factor 13 could be measured for large networks. Another approach introduced a distributed method allowing the Newton method to be executed on different computers, interconnected via Ethernet [3]. A speedup of five was achieved when using eight processors. Finally,

[3] It is a simulated grid, which corresponds to a city with the same population and population density as Zurich, not the actual topology of Zurich's electrical power grid.

there was an approach to transfer computationally intensive tasks to the GPU achieving speedups of around factor two [5].

In this approach the power flow problem was parallelized on a high level, independent of a specific solver or algorithm. Any algorithm can be used to solve the subgrids created by the parallelization method. Therefore, future power algorithms can also be parallelized with this method. Another advantage of this approach is that the individual subgrids could also be solved with different algorithms, depending on which algorithm performs best on the given subgrid topology.

## 1.6 Validation of Teng's solver

Newton Raphson and Gauss Seidl are well tested algorithms, since they have been around for a very long time. Teng's implementation of the backward/forward sweep algorithm was not tested extensively. Since Teng's solver was used in this project, it is validated again. Teng's solver was validated by comparing the calculation results of three different grids to another solver or algorithm. The grids vary in size, voltage levels and whether they are meshed or not. **Table 2** shows the grid characteristics. A visualization of Grids V1 and V2 can be found in Appendix B.

| | **Grid V1** | **Grid V2** | **Grid V3** |
|---|---|---|---|
| **Nodes** | 13 | 166 | 1364 |
| **Loads** | 3 | 123 | 1076 |
| **Generators** | 1 | 0 | 0 |
| **Lines** | 10 | 144 | 1219 |
| **Transformers** | 5 | 21 | 144 |
| **Voltage Levels** | 5.2; 8.5; 16; 65; 220 | 0.4; 10 | 0.4; 10; 100 |
| **Type** | meshed | radial | radial |
| **Validated against** | NEPLAN | Gauss Seidl | Newton Raphson |

**Table 2** *Grids used to validate the Sweeping algorithm*

The validation was performed against Newton Raphson, Gauss Seidl and NEPLAN [4], a network simulation tool. The "Extended Newton Raphson" method of NEPLAN version 5.5.7 was used. For the validations with Newton Raphson and Gauss Seidl, the solvers from the JPower framework [5] have been used. They are publicly available on the hosting platform "GitHub". All simulations were performed with a convergence criterion of 1E-12. All solvers converged. The bus voltage magnitude and angle errors of all nodes have been computed. The maximal occurred error per grid can be seen in Table 3. The errors of grids V2 and V3 are satisfactory considering the convergence criterion of 1E-12. Since NEPLAN has an accuracy limitation of five decimals, the results of grid V1 are accurate too. Altogether, the results of Teng's solver are correct.

| | Grid V1 | Grid V2 | Grid V3 |
|---|---|---|---|
| **Maximal Bus Voltage Error [kV]** | 3.987 E-6 | 2.846 E-9 | 7.521 E-11 |
| **Maximal Bus Voltage Error [pu]** | 7.913 E-7 | 4.895 E-10 | 1.880 E-10 |
| **Maximal Bus Angle Error [rad]** | 1.735 E-4 | 3.416 E-10 | 4.059 E-11 |

**Table 3** *Accuracy of the Sweeping algorithm*

# 2 Parallelize the Power Flow Problem

The first task was to parallelize the task of solving the power flow of a certain grid conceptually, i.e. finding a set of problems whose solution is identical to the original power flow problem. To be able to solve this set of problems in parallel it is important that there are no data dependencies.

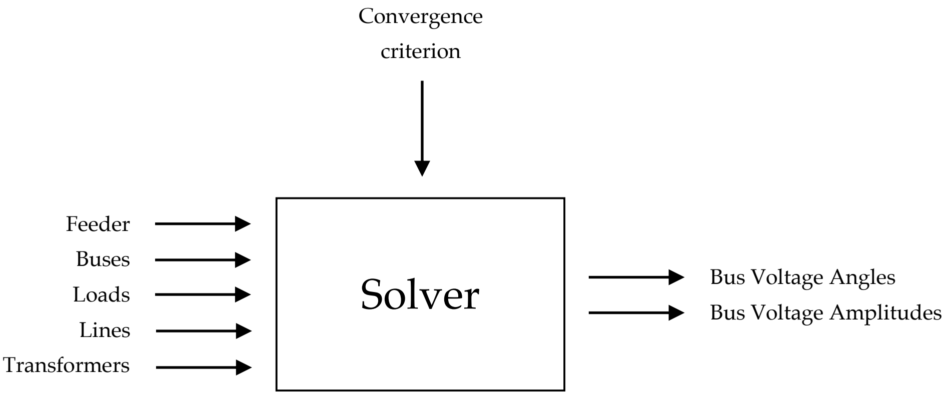


**Figure 1** *Solver input/output characteristics*

The solvers based on the algorithms presented in Chapter 1 have the same input/output characteristics, shown in Figure 1. The arrows on the left of the figure represent the grid including topological information as well as technical specification. Take for example the power line arrow. It must be specified which buses the line connects, the line resistance, reactance, length, a maximum current and whether the bus connections are closed. An example of a grid, represented in a textual form as XML-file together with its graphical representation can be found in Appendix A.

In this project, the approach was to split the grid in several subgrids. All of which are complete in the sense that they contain all information needed for the grid to be solved without knowledge of the original grid or adjacent subgrids. This characteristic is advantageous for two reasons: firstly, it allows the subgrids to be computed in parallel without data dependencies and secondly, the subgrids can be solved using any power flow solver.

To compensate for the removed connections between the subgrids extra buses, feeders[4] and loads are inserted into the subgrids. They simulate interactions with adjacent subgrids. These additional elements are referred to as *dummy elements*. The set of all dummy elements, belonging to the simulated interaction between two subgrids, is called *splitting point*. Every cut in the original grid leads to the creation of exactly one splitting point. The cuts are made along the voltage levels. Therefore, there number of cuts corresponds to the number of transformers.

## 2.1 Splitting a grid

The grid that should be split and solved is hereinafter called *original grid*. The grids that emerge from the splitting process are called *subgrids*. The original grid needs to be radial for the splitting strategy to work. The original grid is split by starting at the feeder, following the grid towards the low voltage ends and performing a cut at every transformer. At every cut, dummy elements are inserted to the resulting subgrids. A more formal explanation is given as follows: Represent the original grid as tree with the feeder being the root. Create an empty subgrid and add the feeder of the original grid. Perform RecursiveRoutine (subgrid, slackbus) with subgrid being the just created grid and slackbus being the bus connected to the feeder in the original grid. RecursiveRoutine is shown in the box of Figure 2.

When applying this splitting strategy, exactly $n_{\text{Transformer}} + 1$ subgrids are created, where $n_{\text{Transformer}}$ is the number of transformers in the original grid. All network elements of the original grid are in exactly one of the subgrids. An example of a small network being split can be seen in Figure 3. The Grid is split at Node 3. The dummy load in Subgrid 1 consumes exactly the amount of power that is injected by the dummy feeder of Subgrid 2. This power is usually positive since Subgrid 1 is hierarchically higher (closer to the original grid's feeder) than Subgrid 2. The power of the dummy bus can also be negative if Subgrid 2 produces excessive energy. In that case, it would be more intuitive to place a dummy generator instead of the dummy load in Subgrid 1. Since negative loads are mathematically equivalent to PQ

[4] The network feeder is a special kind of bus with a pre-specified constant voltage. A grid usually contains exactly one feeder. It represents the "rest of the network", which is not simulated. The feeder can consume or provide power such that the power flow problem has a solution without violating any constraints. Consider a network with two nodes, connected by a cable. Each node has a load of 1kW attached to it. There are no generators in the network. The power consumption of both loads would be violated due to the lack of a power supply in the network. That's where the feeder comes in. The power consumed by both loads as well as the cable losses come from the feeder, which can be connected to either one of the buses. The amount of power that the feeder can provide or consume is assumed to be infinite. The bus connected to the feeder is called slack bus.

generators this does not matter. Furthermore, the sign of the power delivered by the feeder is unknown prior to calculation.

```
RecursiveRoutine (currentSubgrid, currentBus)
  - Add currentBus to currentSubgrid
  - Add loads and generators connected to currentBus to currentSubgrid
  - Add lines connected to currentBus to currentSubgrid
  - For every added line
      - Call RecoursiveRoutine (currentSubgrid, busBeyondLine) with busBeyondLine being the bus on the other end of the line.
  - For each connected transformer
      - Add the transformer to currentSubgrid.
      - Create a dummy bus and a dummy load beyond the transformer and add them to the currentSubgrid
      - Connect the transformator to the dummy bus.
      - Create a new grid with a dummy feeder.
      - Call RecursiveRoutine (newSubgrid, originalBus) with originalBus being the bus to which the transformer was originally connected to.
```

**Figure 2**

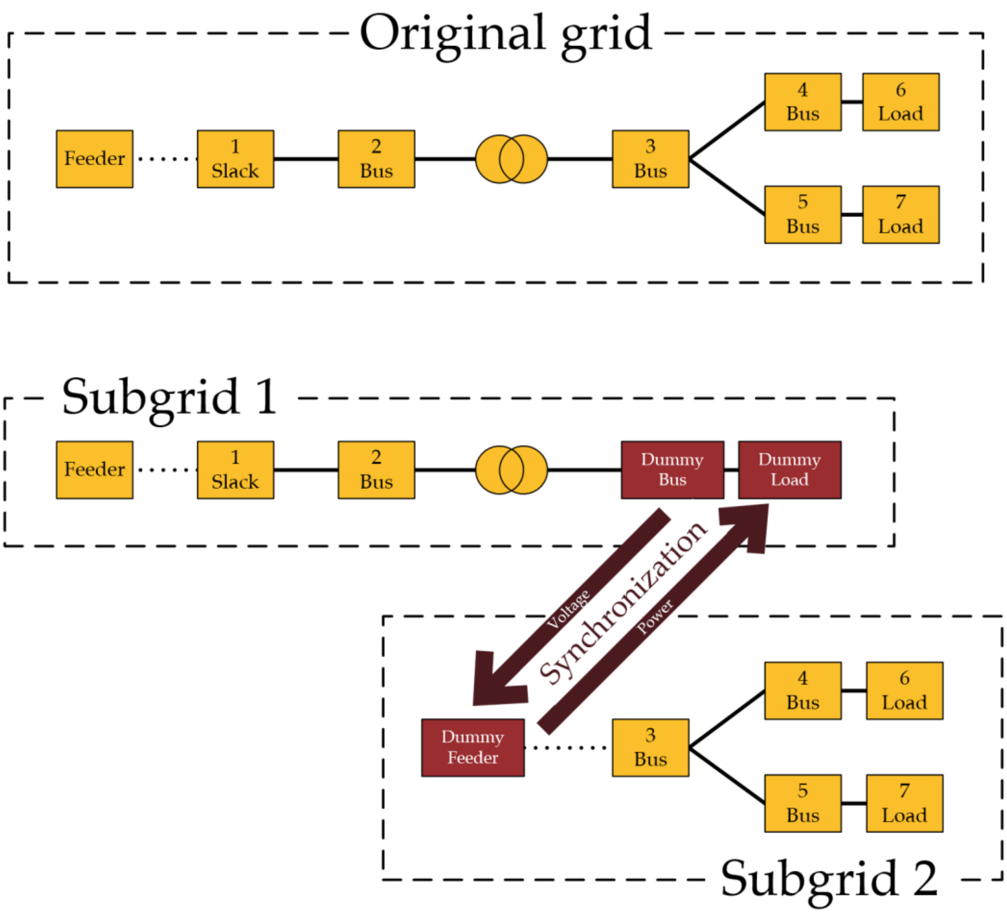


**Figure 3** *Visualization of the splitting strategy*

Note that meshed grids are not supported as they would lead to the creation of at least one subgrid with two feeders. Since the power flow solvers are limited to a single feeder per grid, such subgrids could not be solved. The restriction of only being able to split radial grid is considered acceptable as most grids are radial or only slightly meshed. The splitting strategy could be modified such that loops of meshed parts of the grid are placed in a single subgrid.

## 2.2 Synchronization

After all subgrids have been computed, the voltages of the dummy feeders and dummy loads have to be updated. This process is called *synchronization*. The first step in synchronization is to set the dummy bus's voltage to the voltage of the dummy feeder. This step is trivial since the bus voltages (magnitude and angle) are a direct result of the power flow solver. The second step is to set the dummy load's power to the feeder power, which is the power injected by the dummy feeder in the adjacent subgrid. To compute the feeder power, the power flow of all lines including losses connected to the feeder's slack bus can be calculated from the node voltages. The active power injected by the feeder is

$$P_{\text{Feeder}} = \sum_{\text{Lines}} P_{\text{Line}} + \sum_{\text{Loads}} P_{\text{Load}} - \sum_{\text{Gen}} P_{\text{Gen}}$$

Lines is the set of Lines connected to the feeder. $P_{\text{Line}}$ is defined such that power flowing from the slack to the other bus is positive. Loads is the set of loads and Gen the set of Generators connected to the slack bus. $P_{\text{Load}}$ and $P_{\text{Gen}}$ are positive. The line loadings are computed using the equations presented in Chapter 3.1 of [4]. Analogously to the active power, the reactive power is

$$Q_{\text{Feeder}} = \sum_{\text{Lines}} Q_{\text{Line}} + \sum_{\text{Loads}} Q_{\text{Load}} - \sum_{\text{Gen}} Q_{\text{Gen}}$$

## 2.3 Convergence

The process of computing all subgrids and synchronizing the dummies is referred to as a *parallel iteration*. In addition to the convergence criterion of the solvers, a second convergence criterion is required to terminate these parallel iterations. The latter will be called *parallel convergence criterion*. It depends on the voltage and power differences during synchronization. The following four criteria must all be satisfied for a single splitting point to have converged:

$$abs\left(\frac{|V_{t-1}|-|V_t|}{|V_t|}\right) < \varepsilon_{par} \qquad abs(P_{t-1} - P_t) < \varepsilon_{par}$$

$$abs(\angle(V_{t-1}) - \angle(V_t)) < \varepsilon_{par} \qquad abs(Q_{t-1} - Q_t) < \varepsilon_{par}$$

$V$ denotes feeder voltages, $P$ and $Q$ denote active and reactive power of the dummy load. The subscripts $t-1$ and $t$ indicate the value of the dummy element before and after synchronization during parallel iteration $t$. $\varepsilon_{par}$ is a predefined constant threshold, called *synchronization threshold*. The smaller $\varepsilon_{par}$, the more accurate the computation. In practice, $\varepsilon_{par}$ should be chosen between 1E-9 and 1E-3. The accuracy of the results, depending on $\varepsilon_{par}$, will be investigated in Chapter 5. The equations for power and voltage angle are not normalized due to the possibility of the denominator to be zero. Parallel iteration stops as soon as all splitting points have converged.

Observations have shown, that the number of iterations needed for convergence is proportional to the number of subgrids connected in series[5]. The following example is intended to illustrate why this effect occurs: Consider a topology with three voltage levels: 100, 20 and 0.4 kV. The splitting strategy will lead to a maximum of three[6] subgrids connected in series: one per voltage level. It takes three parallel iterations for the power consumption of the leaf subgrids to propagate up to the main feeder. After these three iterations convergence is not immediately given as small voltage changes at the splitting points change the voltages of the dummy feeders of the adjacent subgrids, leading to changes in cable losses in these subgrids. These loss changes in turn affect the power consumption of the subgrid, which needs to propagate up to the main feeder again. Because of this effect, it is reasonable to limit the number

[5] Subgrids are said to be connected in series if they build a chain, more formally: n subgrids are connected in series, if there are two subgrids such that the shortest path connecting these two, traverses all other subgrids. For example, Subgrid 1 and Subgrid 2 in **Figure 4** are connected in series.

[6] It might be argued that the longest path of connected subgrids is twice the amount of voltage levels when starting at a leaf subgrid, going to the top subgrid and back down to another leaf subgrid. As, however, the feeders' voltage is fixed, the power consumptions of subgrid "branches" do not affect each other.

of subgrids in series. In case of the here presented splitting strategy, this is the case as the number of voltage levels are usually small.

## 2.4 Calculation error caused by the splitting points

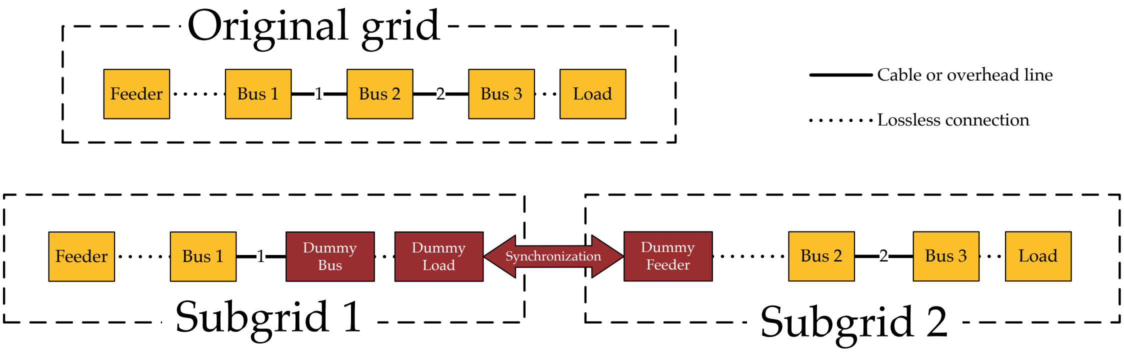


**Figure 4** *Grid to test a single splitting point*

To evaluate the influence of a splitting point on the calculation error of the original grid a small grid is being investigated. The main goals of this subchapter are to test if the parallel calculation converges to the same result as the sequential calculation and to quantify the remaining errors. The latter is of importance as these errors are aggregated along several splitting points in big grids. Consider a grid with three buses, two lines and one load, connected as shown in Figure 4. The feeder energizes the network with 1 kV. Since there are no transformers, the load is fed by the feeder. The grid is split[7] between buses 1 and 2. The main interest is to measure the bus voltage magnitude and angle difference between the network being computed in parallel using the presented splitting strategy and a regular computation as reference.

All computations performed in this subchapter are conducted with the Sweeping solver. For the reference computation, a convergence criterion of 1 E-30 is used. The subgrids in the parallel computation are computed with a convergence criterion of 1 E-30 as well. The termination criterion for the parallel computation is being varied between 1 E-1 and 1 E-30. The load is being varied between 10W and 10 kW. The value of the load might influence the accuracy of the line loading computation (which is needed to synchronize the subgrids). To minimize the possibility of having an error in the code the solver was simplified for this topology[8]. The results are shown in the heat maps of Figure 5. The figure shows that the parallel

[7] The splitting strategy would only split next to a transformer. For simplicity, no transformer was introduced for this experiment.

[8] The general parallel solver, which will be presented in Chapter 3, contains a lot of additional code to increase performance and handle edge cases correctly.

computation converges to the same result as the regular calculation. The errors converge to very small values. The remaining error after the investigated 15 iterations are smaller than $10^{-11}$ Volts (magnitude error) and $10^{-13}$ rad (angle error).

The speed of convergence depends on the line loading, which is controlled via the load. The error decreases faster with small line loadings (the middle columns of Figure 5). The reason for the slower convergence at high load powers is that the high line loadings increase the deviation of the nominal voltage along the network and therefore increase the effect described in Chapter 2.3. The slower convergence at the higher loadings raises concerns: Is the accuracy of the parallel algorithm for big grids with high line loadings satisfactory? Fortunately, the validation of the parallel algorithm in Chapter 5 shows that this is the case.

← Load Power →

| Iterations | -100 | -10 | -1 | -0.1 | 0.1 | 1 | 10 | 100 |
|---|---|---|---|---|---|---|---|---|
| 1 | -1.23 | -2.18 | -3.17 | -4.06 | -1.18 | -3.17 | -2.16 | -1.03 |
| 2 | -1.23 | -2.18 | -3.17 | -4.08 | -1.18 | -3.17 | -2.16 | -1.03 |
| 3 | -3.38 | -6.08 | -7.96 | -8.89 | -5.99 | -7.96 | -6.02 | -2.63 |
| 4 | -3.38 | -6.08 | -7.96 | -8.91 | -5.99 | -7.96 | -6.02 | -2.63 |
| 5 | -5.55 | -10.00 | -12.75 | -13.68 | -10.81 | -12.75 | -9.88 | -4.16 |
| 6 | -5.55 | -10.00 | -12.75 | -13.71 | -10.81 | -12.75 | -9.88 | -4.16 |
| 7 | -7.73 | -13.91 | -15.65 | -15.65 | -15.95 | -15.48 | -13.74 | -5.69 |
| 8 | -7.73 | -13.91 | -15.65 | -15.65 | -15.95 | -15.48 | -13.74 | -5.69 |
| 9 | -9.91 | -15.35 | -16.00 | -15.65 | -16.00 | -16.00 | -15.35 | -7.23 |
| 10 | -9.91 | -15.35 | -16.00 | -15.65 | -16.00 | -16.00 | -15.35 | -7.23 |
| 11 | -12.09 | -16.00 | -16.00 | -15.65 | -16.00 | -16.00 | -15.65 | -8.76 |
| 12 | -12.09 | -16.00 | -16.00 | -15.65 | -16.00 | -16.00 | -15.65 | -8.76 |
| 13 | -14.24 | -15.65 | -16.00 | -15.65 | -16.00 | -16.00 | -16.00 | -10.29 |
| 14 | -14.24 | -15.65 | -16.00 | -15.65 | -16.00 | -16.00 | -16.00 | -10.29 |
| 15 | -15.65 | -15.65 | -16.00 | -15.65 | -16.00 | -16.00 | -15.65 | -11.82 |

| Iterations | | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| 1 | -2.00 | -2.97 | -3.97 | -4.23 | -1.95 | -3.97 | -2.96 | -1.83 |
| 2 | -2.00 | -2.97 | -3.97 | -4.45 | -1.95 | -3.97 | -2.96 | -1.83 |
| 3 | -4.76 | -6.20 | -7.20 | -8.09 | -5.19 | -7.20 | -6.19 | -3.98 |
| 4 | -4.76 | -6.20 | -7.20 | -8.11 | -5.19 | -7.20 | -6.19 | -3.98 |
| 5 | -6.93 | -10.11 | -11.99 | -12.91 | -10.01 | -11.99 | -10.05 | -5.50 |
| 6 | -6.93 | -10.11 | -11.99 | -12.93 | -10.01 | -11.99 | -10.05 | -5.50 |
| 7 | -9.10 | -14.02 | -17.75 | -16.65 | -14.83 | -16.69 | -13.91 | -7.03 |
| 8 | -9.10 | -14.02 | -17.75 | -16.65 | -14.83 | -16.69 | -13.91 | -7.03 |
| 9 | -11.28 | -16.63 | -16.72 | -16.59 | -17.20 | -16.64 | -16.85 | -8.57 |
| 10 | -11.28 | -16.63 | -16.72 | -16.59 | -17.20 | -16.64 | -16.85 | -8.57 |
| 11 | -13.46 | -16.56 | -16.87 | -16.59 | -17.20 | -16.64 | -16.93 | -10.10 |
| 12 | -13.46 | -16.56 | -16.87 | -16.59 | -17.20 | -16.64 | -16.93 | -10.10 |
| 13 | -15.64 | -16.44 | -16.87 | -16.59 | -17.20 | -16.64 | -16.95 | -11.63 |
| 14 | -15.64 | -16.44 | -16.87 | -16.59 | -17.20 | -16.64 | -16.95 | -11.63 |
| 15 | -16.68 | -16.44 | -16.87 | -16.59 | -17.20 | -16.64 | -16.97 | -13.16 |

**Legend**

| | | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| -1.00 | -2.50 | -4.00 | -5.50 | -7.00 | -8.50 | -10.00 | -11.50 | -13.00 |
| ↑<br>inaccurate | | | | ↑<br>accurate | | | | ↑<br>very accurate |

**Figure 5** *Maximum bus voltage errors of parallel computation. The logarithm of the voltage magnitude error is shown in the upper heatmap, the logarithm of the angle error on the bottom. The map shows the logarithms of the errors. Therefore, a value of -6 stands for an error in the order of μV or μRad.*

# 3 Implementation

This chapter shows an implementation of the concept presented in Chapter 2. It contains insights for anyone who might be interested in using and/or changing this implementation. It does not contain any scientific results. The implementation was written in Java using the existing Sweeping solver to compute the individual subgrids.

## 3.1 Grid representation

To represent the grid this implementation uses a data structure used by DPG.sim. Every type of element in the grid is represented by a class. There are classes for busses, generators, lines, loads etc. The instances of these classes are hereinafter called *network objects*. The grid itself is represented by an object of the class Grid, which has references to the network objects. These references are stored in collections, mostly lists. A grid object is a complete topological representation of a grid including power production, consumption, line and transformer parameters. A power flow problem, represented by a grid object, can be solved with all presented algorithms and has a unique solution.

## 3.2 Structure of the Parallel Solver

The implementation of the parallel solver contains different classes. "ParallelSweepSolver" is the main class. All helper classes are inner classes of the ParallelSweepSolver class. The "Splitter" class divides the grid into multiple subgrids, adds dummy elements to the subgrids, keeps track of which subgrids are connected to each other and synchronizes the dummy elements between two parallel computations. At the end of all calculations, the splitter reverts all changes of grid object, so that the grid object can be re-used by another solver. The "SubgridComputator" class is responsible for repeatedly solving a single subgrid. It does not contain an actual solver. It uses an instance of SweepSolver[9] to solve the grids. For every subgrid the solvePowerFlow method creates an instance of SubgridComputator and executes all of them in parallel in a thread pool.

[9] The chosen solver to solve the subgrid can easily be changed. The Sweeping solver was optimized to perform well with the ParallelSweepSolver.

The "PowerInitializer" class initializes the power at the buses for a single grid. It caches additional topological grid information to perform well. This information includes data structures to efficiently retrieve all loads and generators connected to a certain node. This is critical for the performance of simulation series as the bus powers must be reinitialized before every computation. The PowerInitializer has both a memory and time complexity of $\mathrm{O}\left(n_B \cdot \frac{n_L+n_G}{n_B}\right) \Leftrightarrow O(n_L + n_G)$ with $n_B$ being the number of buses, $n_L$ the number of loads and $n_G$ the number of generators.

## 3.3 Procedure of computing a grid

### 3.3.1 Phase One: Split grid

In phase one the grid is subdivided into subgrids. This division is implemented in the Splitter class as specified in Chapter 2. To split a grid one instance of Splitter is required. Contrary to the description in Chapter 2.1 the splitting process is implemented using iteration instead of recursion as iteration is efficient in terms of speed and memory usage. During the partitioning process, a new instance of grid is created for every subgrid. The splitter does not create new instances of the gridObjects as this would lead to unnecessary additional memory usage. Instead it saves references to the existing network objects of the original grid. This has another advantage: if grid parameters, such as the power of loads, change between successive computations of the same grid, there is no necessity to update all gridObjects of the subgrids.

The splitting points created during the grid partitioning process are each represented by an instance of the splittingPoint class, an inner class of Splitter. The Splitter stores references of all splittingPoint objects. The splittingPoint object itself contains reference to all gridObjects involved in that splitting point: the transformer, the dummy bus connected to the transformer, the dummy load, the dummy feeder, the slackbus of the dummy feeder and both subgrids.

When the splitter has processed the entire grid, the subgrids are given to the main class for computation. An overview of the entire process can be seen in Figure 6.

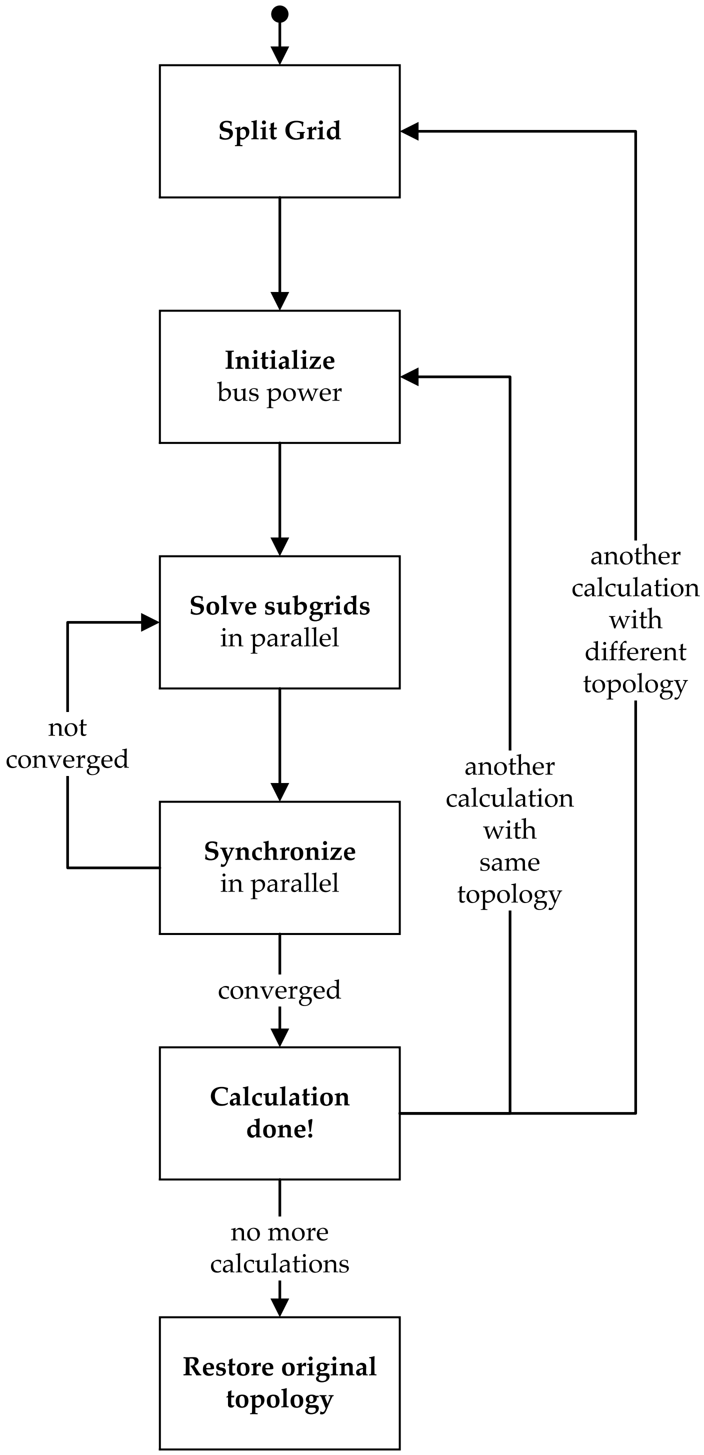

Split Grid
Initialize bus power
Solve subgrids in parallel
not converged
Synchronize in parallel
converged
Calculation done!
no more calculations
Restore original topology
another calculation with same topology
another calculation with different topology

**Figure 6** *Structure of parallel power flow solver*

### 3.3.2 Phase Two: Solve subgrids, synchronization

The solvePowerFlow method of the ParallelSweepSolver class solves all subgrids in parallel using the SubgridComputators. The threshold of convergence criterion to solve the subgrid is chosen to be $\min(0.1 \cdot \varepsilon_{par}, 10^{-6})$. $\varepsilon_{par}$ is the synchronization threshold, the threshold used to determine whether a splitting point has converged (see Chapter 2.3). Details on the parallelization are given in Chapter 3.6.

When all subgrids have been computed, synchronization starts. Synchronization is implemented in the splitter as specified in Chapter 2.2. Synchronization is performed in parallel. After writing the dummy bus voltage to the dummy feeder and assigning the feeders power consumption to the dummy load the dummy buses are initialized with the new power. This step avoids the necessity to re-initialize the power in all subgrids. Finally, the splitting method checks if the splitting point has converged with the criterion shown in Chapter 2.3. If one or more splitting points didn't converge, phase two is repeated.

### 3.3.3 Phase Three: restore original topology

When all splitting points have converged, the original grid is reverted to its original topology. During the splitting process one bus reference per transformer was replaced by the reference of the dummy bus of the splitting point. These changes are reverted by the restoreOriginalTopology method of the splittingPoint class.

## 3.4 Improvements of the Sweeping solver

The Sweeping solver has been optimized to enhance performance and usability. An overview of the modifications is given as follows:

**Feeder angles** Bus voltage initialization was adapted to handle grids with non-zero feeder voltages correctly.

**Result-Object** The solver was modified to store the computation results directly in the grid object. The solver returns void (instead of a data structure containing the node voltages).

**Fictitious Results** The (optional) result vector was adapted to exclude results of fictitious buses that were added by the solver[10].

[10] These buses are introduced by the Sweeping algorithm to handle loops.

**Warm start** If the grid is repeatedly computed without topology changes, the algorithm converges faster if the buses are initialized with voltages of previous computations as these voltages are generally closer to the new solution than with an initialization of 1 per unit magnitude and 0 rad angle (which corresponds to a flat start that is used for the first computation). The sweeping algorithm initializes the bus voltages with the previous results if Grid.warmStart is set to true.

**PQ vs PV generators** The sweeping solver originally implicitly treated all generators as PV[11]. As the other solvers implicitly treat them as PQ generators, the sweeping solver was adapted to do the same. A "generatorsArePQ" boolean was introduced to support both generator types in a later point in time. It is currently hardcoded to be set (true) for the parallel solver to behave consistently with the other solvers.

**Recompute grid** The solver structure was modified to match the constructors and methods of the ones of ParallelSweepSolver. See Chapter 3.7 for details.

## 3.5 Compatibility of grids with the parallel solver

There are many different elements that electric power grids may contain. This section shows which elements have been used to test the parallel solver and are therefore compatible.

Buses — Supported and tested during validation with Sweeping algorithm.

Loads — Supported with active and reactive load component. "Connected" flags are ignored.

Generators — Implicitly assumed to be PQ. Supported and tested for PQ generators with active and reactive component. Min/Max load constraints and connected flags are ignored.

Lines & Transformers — Supported and tested considering line length, resistance, reactance. Maximum current constraints and connected flags are ignored.

Feeders — Supported and tested for input grids with exactly one feeder. The splitter class ignores all feeders but the first one in the feeders list. Feeders' "operational" angle and voltage are considered.

[11] A PV generator is a generator that is controlled such that power (P) and voltage (V) are constant. A PQ generator on the other hand is controlled such that active (P) and reactive (Q) power are constant.

Topology		Only radial grids are supported. When given a meshed grid, the solver computes a false result or loops forever in the splitting procedure.

Other elements are generally ignored. There is no universal recipe to implement other grid elements to the be handled correctly by the parallel solver. Generally, one must add the elements to the subgrids at label[12] #02#, add a map with the bus with which the element interacts as key and the element itself as value at label #01# to be able to access it efficiently at label #02#. If the new grid element is connected to a bus, which may be split (buses with transformers connected to them) it must be handled by the splittingPoint class (label #03#) and at the synchronize method (label #04#). If the added element is modified during the splitting process, the changes should be reverted at the end of the computation (at label #05# and/or #06#)

## 3.6 Parallelization

Time-consuming parts of the solver are parallelized, the rest runs in the main thread. Subgrid computation and the synchronization of the dummy elements are the most CPU-intensive parts of the solver. They have been parallelized using Java's ExecutorService, which is an implementation of a thread pool. A thread pool is a collection of threads executed in parallel. In the application of the parallel solver, every thread solves a subgrid or synchronizes a splitting point. The number of threads, also called the size of the thread pool, is constant. When a thread is finished solving a subgrid it is given a new one to solve. When all subgrids are solved the thread-pool is shut down. Measurements show that using a thread pool decreases computation time by 34% and roughly doubles CPU utilization compared to running one thread per subgrid. The latter means that for big grids, hundreds of threads are created and started simultaneously. The frequent context switches between the threads are inefficient. Using a thread pool is much more efficient as there are fewer context switches and less memory consumption. The question arises of how big to choose the thread pool. Table 4 shows the dependency of the performance on the size of the thread pool. The computer on which the tests were conducted, has a Quadcore CPU with hyper threading enabled, leading to a total number of 8 logical CPUs. All hardware details are listed in Chapter 4.4. Best performance is achieved when choosing the thread pool size likewise.

[12] These labels can be found as comments in the ParallelSweepSolver class.

| Method | Time | peak Memory usage | avg. CPU usage |
|---|---|---|---|
| Individual Threads | 100 s | 1637 MB | 18.94 % |
| Thread pool with 4 Threads | 85 s | 1716 MB | 33.65 % |
| Thread pool with 8 Threads | 66 s | 1640 MB | 38.14 % |
| Thread pool with 16 Threads | 72 s | 1711 MB | 40.73 % |

***Table 4*** *Performance of different parallelization implementations*

## 3.7 Usage example

The constructor of the parallelSweepSolver class takes three inputs: A grid object, the synchronization threshold and the maximal amount of iterations. The solvePowerFlow method then takes a single input: a boolean indicating whether the topology of the grid has changed since the last calculation with the solver. The solvePowerFlow method does not have an output. The calculation results are stored in the grid object, more precisely in the PowerFlowDataBus objects of the Bus objects.

**Solve a grid with the parallel sweeping solver:**

```
grid.initialize();
ParallelSweepSolver parallelSweepSolver = new ParallelSweepSolver(grid,1E-6,1000);
parallelSweepSolver.solvePowerFlow(false);
parallelSweepSolver.restoreOriginalTopology();

// access convergence info
boolean hasConverged = sweepSolver.hasConverged();
int iterations = sweepSolver.getIterations();
```

**Solve a grid with the sweeping solver:**

```
grid.initialize();
new PowerInitializer(grid).initializePower(); // replaces initPower(grid)
SweepSolver sweepSolver = new SweepSolver(grid,1E-9,1000);
sweepSolver.solvePowerFlow(false);

// access convergence info
boolean hasConverged = sweepSolver.hasConverged();
int iterations = sweepSolver.getIterations();
```

**Access the results**

```
for (Bus bus:grid.getBuses()){
    int busID = bus.getID();
    double voltage_pu = bus.retrievePowerFlowData().getVoltageInPerUnit();
    double voltage_kV = bus.retrievePowerFlowData().getVoltageInKilovolt();
    double angle_rad = bus.retrievePowerFlowData().getAngleInRadians();
}
```

### Simulation Series with sweeping solver

```
grid.initialize();
PowerInitializer powerInitializer = new PowerInitializer(grid);
powerInitializer.initializePower();
SweepSolver sweepSolver = new SweepSolver(grid,1E-9,1000);

// series without topology change
for (int i=1;i<10;i++){
    // modification of the power of loads and generators...
    powerInitializer.initializePower();
    sweepSolver.solvePowerFlow(false); // not a topology change! --> false
}

// series with topology change
for (int i=1;i<10;i++){
    // modification of the network topology...
    grid.initialize();
    powerInitializer = new PowerInitializer(grid);
    powerInitializer.initializePower();
    sweepSolver.solvePowerFlow(true); // topology change --> true
}
```

### Simulation Series with parallel sweeping solver

```
grid.initialize();
ParallelSweepSolver parallelSweepSolver = new ParallelSweepSolver(grid,1E-6,1000);

// series without topology change
for (int i=1;i<10;i++){
    // modificatiopn of the power of loads and generators...
    parallelSweepSolver.solvePowerFlow(false);
}

// series with topology change
for (int i=1;i<10;i++){
    parallelSweepSolver.restoreOriginalTopology();
    // modification of the network topology...
    grid.initialize();
    parallelSweepSolver.solvePowerFlow(true);
}
parallelSweepSolver.restoreOriginalTopology();
```

# 4 Testing

## 4.1 Grid creation tools

As parallelization comes with the computational overhead of splitting the grid in subgrids, parallelization makes only sense for grids of a certain size. Originally, it was intended to use GridBuilder to build large grids for performance testing. GridBuilder is a MATLAB tool developed by Viktor Lenz in 2014 for Adaptricity [5]. It builds grids taking several inputs, the most important of which are the size in square kilometers and the number of inhabitants of the area that should be modelled. After specifying these two parameters it is possible to draw a population density map using the graphical user interface. The tool then generates realistic radial low voltage (LV) grids that are connect by a medium voltage (MV) network in an open ring configuration to the network feeder. The LV grids are 0.4kV, the MV grids are 10kV. The tool internally uses a spanning tree algorithm to connected the LV loads. Due to performance limitations of this algorithm grids generated by GridBuilder are limited in size to a few thousand nodes, which is not enough to push the parallel solver to its limit.

To be able to generate larger grids a Java tool called CityGridBuilder (CGB) was developed. It generates a big grid by assembling existing small grids created by GridBuilder[13]. CGB takes two arguments, equivalent to those of GridBuilder: number of inhabitants and population density. The program then chooses GridBuilder-Grids from a given database matching the desired population density. The program repeatedly adds GridBuilder grids to the network until the desired number of inhabitants is reached. The GridBuilder grids are connected in an open-ring configuration on the MV level. If the grid network exceeds the threshold of 10 GridBuilder Grids, an additional HV grid is created to connect the different GridBuilder groups among each other, again in an open-ring configuration. The voltage of the HV grid level is 100kV. Using CGB, it is possible to create grids, that represent areas with several million of inhabitants. The created grids are radial and homogeneous with respect to population density.

[13] The XML export of GridBuilder was modified to include some statistical information about the grid in the XML file. This is then used by CityGridBuilder to choose grids that match the required population and density as well as possible.

To be able to investigate the optimal size of the subgrids another grid creation tool was created: SyntheticGridBuilder. The tool is given the exact size and numbers of subgrids to create. The topology is very simple. All subgrids are directly connected to the slackbus. Every subgrid is a binary tree. Each bus in that subgrid has either one or two children, chosen randomly. The subgrid is grown until the required size is reached. A grid with two subgrids and 5 nodes each is shown in **Figure 7**. SyntheticGridBuilder will be used in Chapter 5.5 to investigate how the performance of the parallel solver changes with changing subgrid size.

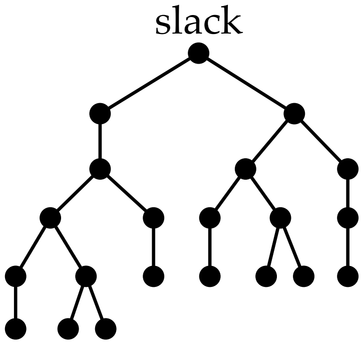


**Figure 7** *Grid with two subgrids and ten buses per subgrid. Created with SyntheticGridBuilder*

## 4.2 Tested Grids

Throughout the result chapter, the same five grids are used to measure and compare the performance of the parallel solver. The grids are shown in Table 5.

| | Grid A | Grid B1 | Grid B2 | Grid C1 | Grid C2 |
|---:|---|---|---|---|---|
| **Buses** | 42 | 655 | 663 | 12’753 | 47’043 |
| **Loads** | 14 | 636 | 510 | 12’343 | 45’514 |
| **Generators** | 0 | 0 | 0 | 2’475 | 9’197 |
| **Lines** | 38 | 645 | 586 | 12’558 | 46’310 |
| **Transformers** | 3 | 9 | 76 | 194 | 723 |
| **Voltage Levels [kV]** | 0.4; 20 | 0.4; 10 | 0.4; 10 | 0.4; 10; 100 | 0.4; 10; 100 |
| **Origin** | DPG.sim | GridBuilder | GridBuilder | CGB | CGB |
| **Population density [Inh/km2]** | - | 200 | 800 | 200 | 200 |
| **Inhabitants** | - | 2’000 | 48'000 | 42'000 | 153'000 |
| **Prosumer Density** | 0% | 0% | 0% | 20% | 20% |

**Table 5** *Test grids*

Grid A is a small Benchmarking grid taken from DPG.sim, it is called «Benchmark LV Microgrid». Grids B1 and B2 roughly equal in size[14] and bigger than grid A. Grid B1 represents a rural area with 2000 inhabitants and a low population density, grid B2 represents a more urban are with a higher number of inhabitants and a higher population density. The grids are considered equal in size due to the almost same number of buses. This is possible because a single connection point serves much more households in areas with a high population density. The difference is the number of transformers. Since the splitting strategy splits the grid along the voltage levels, for every transformer a splitting point is generated. The splitting of grid B2 results in 77 subgrids with an average size of 8.6 while B1 is only split into 10 subgrids with an average size of 65.5.

Grids C1 and C2 are much larger than the B-grids. Grids of these sizes cannot be solved by Newton Raphson and Gauss Seidl. The solvers would use more memory then offered by the test computer. Grids C1 and C2 correspond to the cantons of Nidwalden and Schwyz in terms of population density and number of inhabitants. Recall that grids created by CityGridBuilder are generic, they therefore do not represent the actual topology of Nidwalden and Schwyz. Currently, it's unusual to simulate a grid of that size. The purpose of the C-grids is to test the limits of the algorithm.

The subgrids in CGB are chosen randomly. To be able to reproduce a CGB-grid, a seed needs to be given to the CGB's constructor. The seed to build grids C1 and C2 is «6718254L».

## 4.3 Java Virtual Machine

A crucial part of this project was to be able to compare the performance of the parallel algorithm with the current algorithms. At first, the difficulty associated with Java Benchmarking was heavily underestimated. The trivial approach to measure code performance is to repeatedly execute the same task, measure elapsed time, and compute the average. When doing that something unfavorable can be observed: For each repetition computation time decreases. Not only for the first few computations, even after thousands of repetitions computation time decreases. Some of the measurements showed outliers, which occurred periodically with equal height. 100 computations of the same grid are shown in Figure 8.

[14] Grid size is defined as the number of buses.

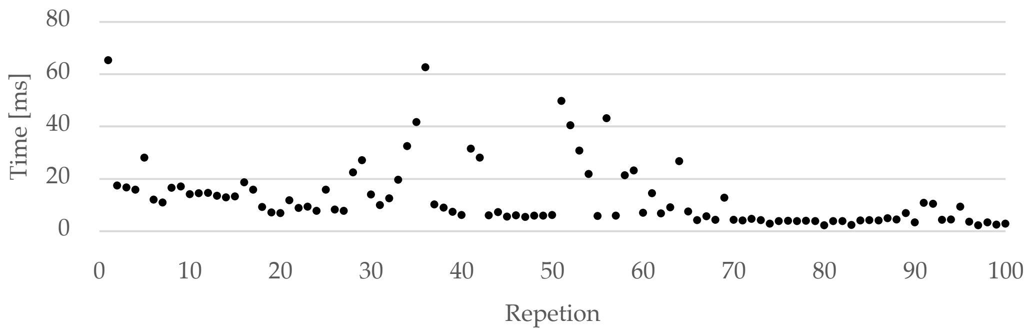


**Figure 8** *Execution times of repeated calculation of the same grid*

Throughout the analysis the following effects were observed frequently:

1. The first few computation take longer.
2. Periodic outliers. They are similar in height and occur periodically. These can be explained by the JVM's garbage collector. While some garbage collectors work in parallel to the execution of the program, others stall program execution during garbage collection causing such outliers.
3. Non-periodic outliers. They are most-likely caused by the operating system's thread scheduler, which allocates CPU time to the different threads of the running programs.
4. Periodic Fluctuation followed by a significant increase in speed.

Plots showing the mentioned effects can be found in Figures Figure 9 to Figure 12. All figures show the runtime of 1000 repetitions of the exact same computation. The repetitions are shown on the x-axis and computation time on the y-axis. Effects 1, 3 and 4 can be explained by the way the JVM compiles Java code to machine code. It uses a so called Just-In-Time-Compilers (JITs). The JIT compiles the Java code at runtime into machine code "just in time" to be executed. During the execution of the program the JIT monitors threads and identifies sections of the code that are executed frequently. These parts are then optimized and recompiled in the background. The optimization and recompilation slows down program execution, after the optimized code is deployed, program execution speeds up.

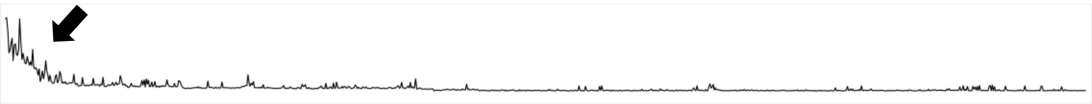


**Figure 9** *Effect 1: higher execution times for the first few executions*

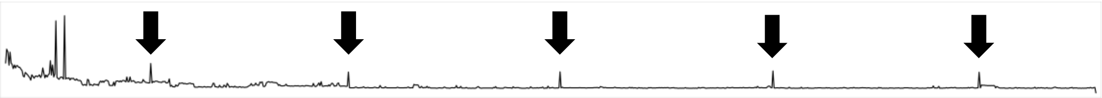


**Figure 10** *Effect 2: Periodic outliers due to garbage collector activity*

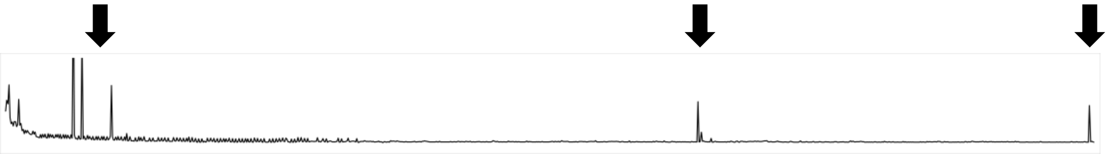


**Figure 11** *Effect 3: Non-periodic outliers, executed on EULER cluster*

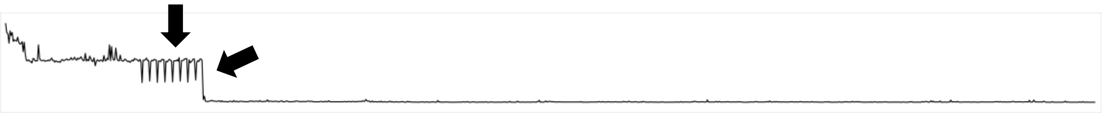


**Figure 12** *Effect 4: periodic fluctuations followed by a significant increase in speed*

As JIT is responsible for most of the fluctuation it was considered to turn it off. Turning JIT compilation off in fact results in a significant reduction of fluctuation, but increases computation time dramatically. Figure 13 shows the computation times without JIT.

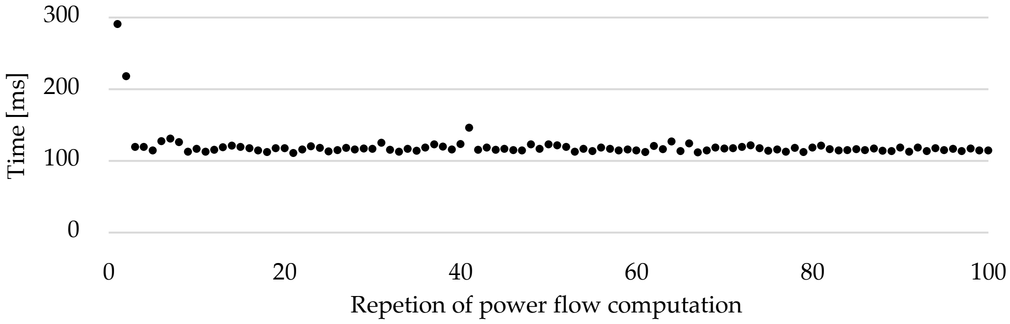


**Figure 13** *Computation times of 100 measurements without Just-In-Time compilation*

There is, however, a problem when turning off JIT compilation: Not all algorithms are slowed down by the same factor. While Newton Raphson is only slowed down by factor 2, Gauss Seidl is slowed down by factor 18. Therefore, it was decided against turning JIT compilation off.  To reduce the effects of observation 3, the EULER cluster[15] was considered to run the performance tests on. The amount of background activity on the computers of the EULER cluster was hoped to be smaller. As this was not the case, the usage of EULER was discarded as well. The following measures were taken instead:

- Before every performance analysis all algorithms are run for 30 seconds with different grids to complete the optimizations of the JIT compiler with the highest impacts.
- Before every computation the garbage collector is called to reduce the impact of Observation 2. After calling the garbage collector the main thread is set to sleep for five seconds to allow garbage collection to be completed before the benchmark starts.
- For single measurements: Every measurement is performed a certain number of times[16]. Outliers are discarded and the average of the remaining measurement is calculated.

Outliers are identified using the methodology shown in [6]. Values outside of boundaries given by the acceptance interval are considered outliers and discarded. The interval boundaries are

$$t_{min} = Q_{12} - 1.5 \cdot abs(Q_{34} - Q_{12})$$
$$t_{max} = Q_{34} + 1.5 \cdot abs(Q_{34} - Q_{12})$$

with $Q_{12}$ being the border between the first and second quartile[17] and $Q_{34}$ the border between the third and fourth quartile. The acceptance interval is $[t_{min}, \ t_{max}]$.

The chosen benchmarking strategy has shown to generate reproducible values. The computed benchmark times are shown in **Figure *14***. Every point corresponds to 100 computations, averaged while discarding outliers. The accuracy of the benchmarking strategy is considered high relative to the execution times of the solvers, which often differ in orders of magnitude.

[15] a high-performance computer-cluster run by ETH

[16] Depending on the grid size, at least 10 times

[17] The first quartile contains smallest 25% of the values of a data set, the second quartile the next 25% and so forth.

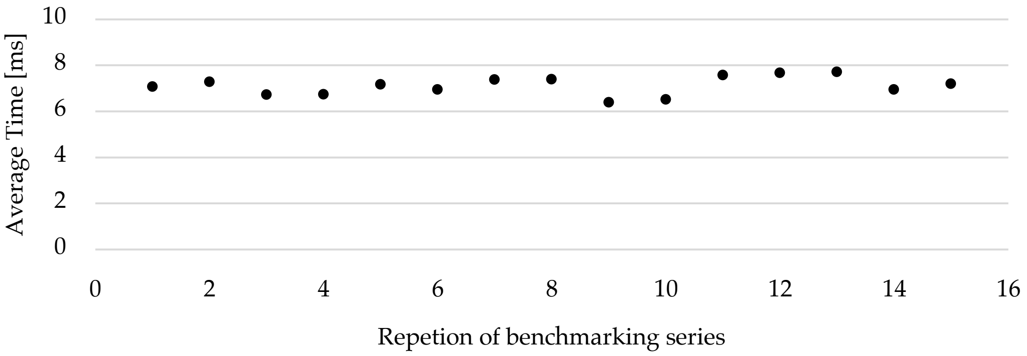


**Figure 14** *Averaged computation times with outliers being discarded*

Concluding this subchapter, it shall be noted that performance testing in Java is a tricky topic with a lot of stumbling blocks. The measures taken mitigate the biggest unwanted effects. Some effects are ignored as there is no possibility to prevent them. An example is the increased garbage collector activity for larger grids. The given memory of 14GB is more than enough to solve a small grid like grid A, therefore the garbage collector may never run at all. For bigger grids, such as C1 and C2, the garbage collection runs frequently to prevent the JVM to run out of memory.

## 4.4 Test environment

Details regarding the hard- and software of the test computer on which all computations have been executed on can be found in Table 6.

| | |
|---|---|
| **CPU** | Intel Core i7, Quad Core, 3.1GHz, Hyper-Threading enabled, 8 logical CPUs |
| **Operating System** | Windows 10 Build 14393, 64 bit |
| **Java Version** | 1.8.0_101 |
| **Java Virtual Machine** | HotSpot (build 25.101-b13) |
| **Garbage Collectors** | PS Scavenge, PS MarkSweep |
| **Memory allocated to JVM** | 14 GB |

**Table 6** *Hard- and software used for performance measurements*

# 5 Results

This Chapter outlines the performance of the parallel algorithm and compares it with the existing algorithms in terms of accuracy and speed. The accuracy of the algorithm depending on the chosen convergence threshold is analyzed in Chapter 5.1. Chapters 5.2 and 5.3 focus on comparing the performance of the algorithms. While Chapter 5.2 focuses on single calculations, Chapter 5.3 investigates performance when simulating an entire year. The memory and CPU utilization of the algorithms are compared in Chapter 5.4. Chapter 5.5 investigates the subgrid size that maximizes performance of the parallel solver. Finally, Chapter 5.5 shows the overhead of parallelization.

## 5.1 Accuracy

The convergence criteria of the four algorithms are defined differently. Therefore, the accuracy of the result differs when the same threshold for the convergence criteria are taken. The main purpose of this chapter is to answer the questions: "Is the parallel solver accurate?" and "Which threshold of the parallel solver should be chosen to obtain equally accurate results when comparing with a given threshold of the sweeping solver?".

Typical values for the convergence threshold of the power flow algorithm are between 1E-3 and 1E-9. The accuracy analysis is conducted with the sweeping algorithm (SW), the parallelized sweeping (PSW) algorithm as well as with Newton Raphson (NR) and Gauss Seidl (GS). The computations were performed with twelve different convergence thresholds. The main interest was to measure the highest errors of bus voltage magnitude and angle and how they depend on the convergence threshold. The results are validated against a calculation with the SW[18] solver with a very small convergence threshold of 1E-16. Grid B2 was used as test grid. The voltage magnitude and angle errors are shown in Figure 15. The calculation times of the algorithms are included to give a rough overview of the differences in performance. In the following chapters, performance will be discussed in-depth. All plots of Figure 15 are presented in double logarithmic scale.

[18] The choice of the SW algorithm for validation could seem to be in favour of the sweeping-based algorithms. Validations against NR with the same threshold showed qualitatively the same results.

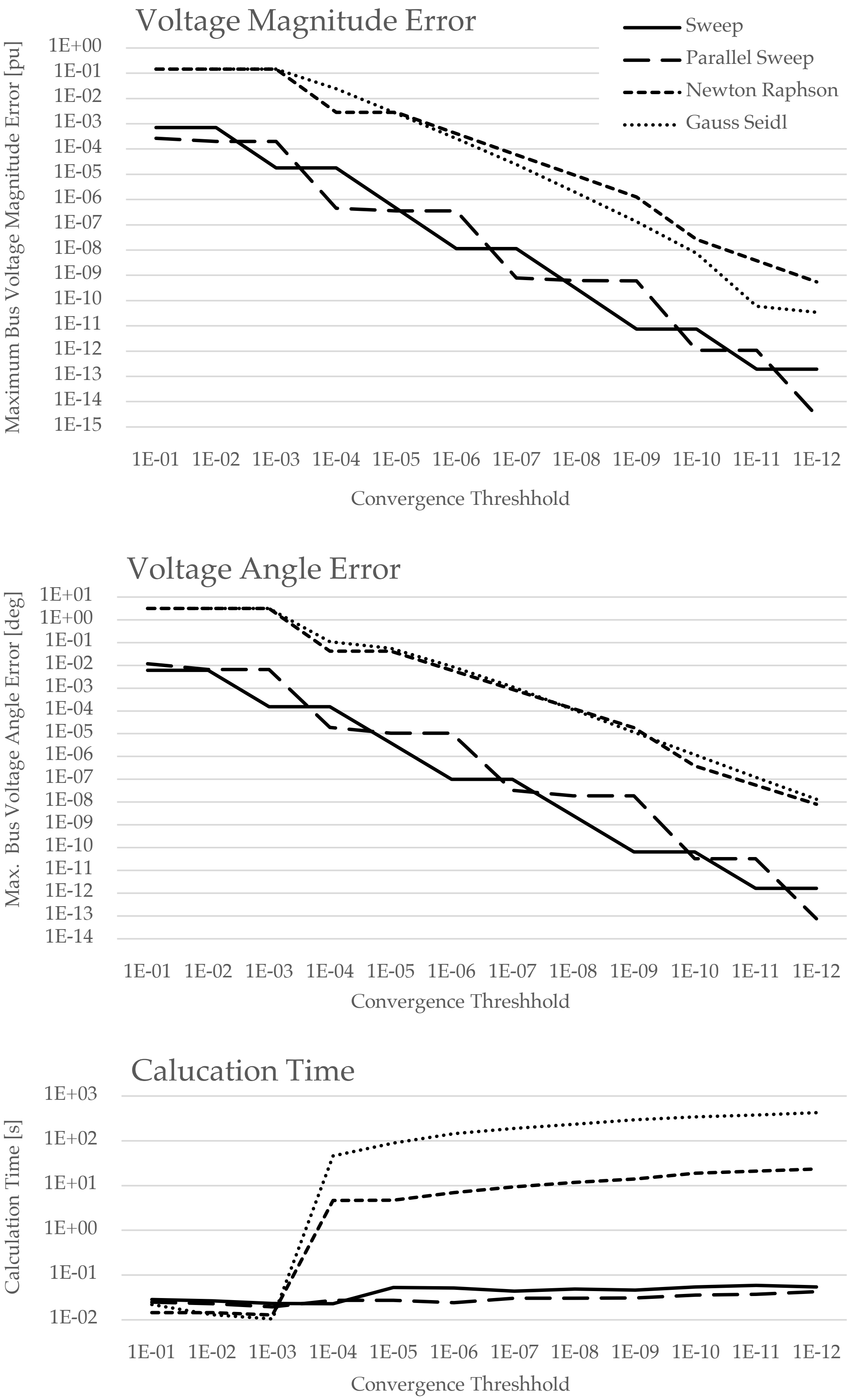


**Figure 15** *Comparison of bus voltage magnitude (top) and angle (middle) accuracy and speed (bottom) of different algorithms for difference convergence thresholds.*

The plots show that the calculation errors decrease with decreasing thresholds. The relation is roughly linear. The PSW solver is as accurate as the SW solver. The errors of SW and PSW are smaller for every convergence threshold compared to NR and GS. Therefore, it is inadequate to compare the speed of the algorithms using the same convergence thresholds. While the threshold dependencies are generally different for all algorithms, two groups stand out: One containing SW and PSW, the other one NR and GS.

The curves of the sweeping algorithms are choppy. This is due to the fact that the amounts of iterations are small, sometimes even identical for different thresholds. Consider for example the flat parts of the parallel Sweeping algorithm. In both error plots, there is a flat area ranging from a convergence threshold of 1E-7 to 1E-9. Thresholds within this range all lead to seven parallel iterations. Due to the choppy nature of these curves it is hard to find a conversion factor to compute an equivalent convergence threshold for the sweeping-based algorithms given a threshold used for Newton Raphson and Gauss Seidl.

Attempts to use curve fitting to approximate the choppy curves to obtain smoother functions did not lead to satisfactory results. Therefore, another approach was used. By defining a maximal tolerated voltage error one can find a convergence error for each algorithm. The maximal error is chosen to be 1E-5, meaning than no bus voltage magnitude error exceeds 1E-5 per unit and no angle error exceeds 1E-5 radians. Using this threshold in a 100kV grid no voltage error would exceed 1V, which is considered accurate.

The analysis of the errors of the five test grids as well as additional 18 grids created using the grid creation tools yielded the thresholds showed in Table 7. For all algorithms, the shown number is the highest threshold satisfying the maximum error condition. The performance comparison of the algorithms in the next chapters will be conducted using these thresholds.

| **Algorithm** | **Threshold** |
|---|---|
| Sweep | 1E-4 |
| Parallel Sweep | 1E-5 |
| Newton Raphson | 1E-8 |
| Gauss Seidl | 1E-8 |

**Table 7** *Equivalent thresholds regarding a maximum voltage magnitude and angle error of 1E-6*

## 5.2 Speed of a single calculation

In this subchapter, the algorithm performance is compared when performing a single calculation. All of five test grids are computed with all algorithms, except for grids C1 and C2, which are only computed with SW and PSW. NR and GS would require more memory than what was offered by the test computer. Measurement series with significantly smaller grids than C1 and C2 already yield in calculation times of hours. The calculation times of the grids are shown logarithmically in Figure 16 and in absolute values in Table 8.

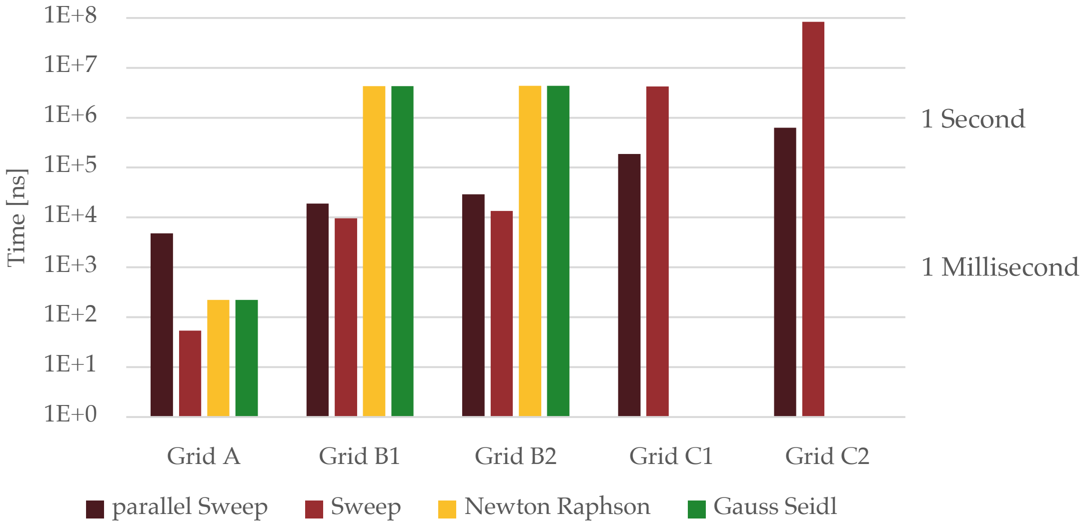


**Figure 16** *Calculation times, shown logarithmically*

| | Grid A | Grid B1 | Grid B2 | Grid C1 | Grid C2 |
|---|---|---|---|---|---|
| **parallel Sweep** | 4.829 | 18.858 | 29.228 | 187.519 | 632.076 |
| **Sweep** | 0.054 | 9.668 | 13.550 | 4’203.992 | 83’647.301 |
| **Newton Raphson** | 0.223 | 4’310.069 | 4’328.519 | - | - |
| **Gauss Seidl** | 0.123 | 286’346.807 | 237’270.710 | - | - |

**Table 8** *Calculation times in milliseconds of a single calculation*

When comparing the classical solvers NR and GS one can easily see that they both perform in the same order of magnitude. They are both significantly slower than the SW solver. That said, the focus is to compare the SW solver with its parallel counterpart, PSW.

Grid A is not suited for parallelization as it is very small, computation takes 89 times longer with PSW compared to the SW solver. The comparison of the B-grids show that the performance of both algorithms is similar with a slight advantage of the SW solver. When computing these grids for a single time parallelization doesn’t make sense as it is slower. Recalling that B1 and B2 are similar in size and primarily differ in the number of transformers, as expected, the overhead of B2 is larger due to its smaller subgrids. More subgrids lead to more splitting points and higher synchronization time. For the C-grids, the parallel algorithm is significantly faster. Grid C1 is computed 21 times faster, grid C2 even 132 times faster by the PSW solver.

## 5.3 Speed of calculation series

As already mentioned in Chapter 1, grids are often simulated over a period of time with a temporal resolution of 15 minutes. Between two individual calculations, node power changes reflect changes in electric power consumption and production. Topological changes occur when for instance switches are opened or closed. While topology usually remains the same between two calculations, node power typically changes every time. In case of no topology change the performance of PSW higher because the grid does not have to be re-split for every calculation. The bigger the grid, the greater the effect. For the smallest grid A, the time to split the grid is only roughly 2% of the total computation time. For the biggest grid C2, it's 44%[19]. Other solvers perform better in repeated grid calculation without topology change as well. The goal of this subchapter is to quantify these performance benefits.

The five test grids are simulated for an entire year with changing consumption and production but without changes in topology. For all algorithms, warm start is enabled. For the PSW this means that both warm start between parallel iterations as well as between different calculation of the entire grid are enabled. All algorithms run for at least five minutes, repeatedly solving one of the test grids. Between the different computations load and generator powers are changed randomly. The exact durations of the running times of the solver are logged. The NR, GS and SW solvers require node power initialization. PSW does that internally. The initialization runtime is added to the solver runtime. After five minutes the test program waits for the termination of the current calculation, then extrapolates the runtimes to match 35'040 calculations, which is the amount of calculations needed when recalculation the network every fifteen minutes for a year. The results are shown logarithmically in Figure 17 and in absolute values in Table 9.

[19] More details will follow in Chapter 5.5

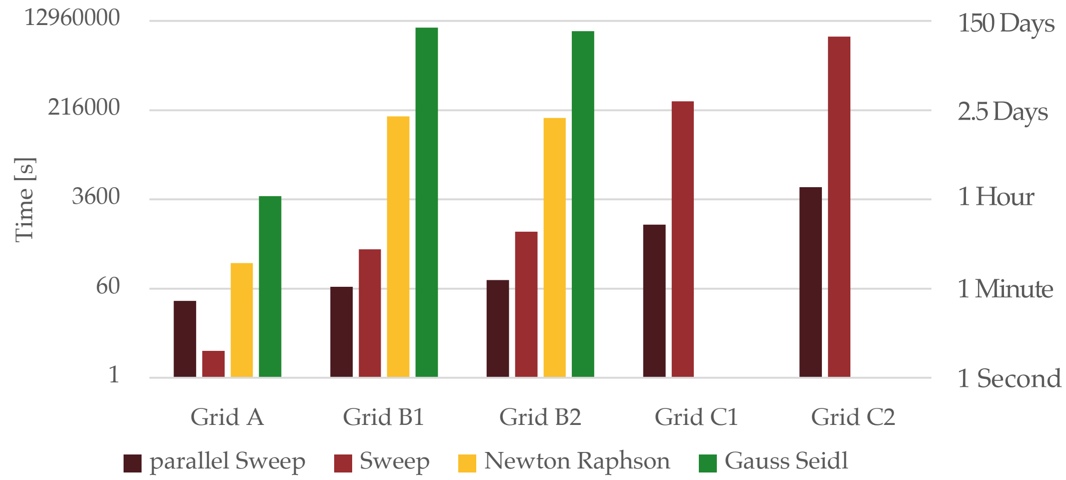


**Figure 17** *Calculation times when simulating a year, shown logarithmically*

| | Grid A | Grid B1 | Grid B2 | Grid C1 | Grid C2 |
|---|---|---|---|---|---|
| **parallel Sweep** | 34.382 | 65.689 | 88.655 | 1124.299 | 6285.764 |
| **Sweep** | 3.436 | 363.412 | 820.179 | 322680.513 | 6261171.937 |
| **Newton Raphson** | 193.031 | 161855.784 | 150336.652 | - | - |
| **Gauss Seidl** | 4148.531 | 9558070.516 | 8029059.288 | - | - |

**Table 9** *Calculation times in seconds when simulating a year*

Generally, PSW is the fastest algorithm, followed by SW, NR and GS. Grid A is the only exception. As in case of the single calculation, the parallelization overhead is too large for topologies with a size of grid A. The minimal size of a network for which parallelization is faster will be found in Chapter 5.5. When looking at the B-grids the tide has turned. Contrary to the algorithm ranking for a single computation, the PSW solver is faster for simulation series. PSW outperforms SW by a factor of 5.5 (grid B1) and 9.25 (B2). The smaller subgrid size of B2 still slows down the performance of PSW but the effect is small than for single calculations in the previous chapter. The performance of the SW solver suffers from the high number of transformers in grid B2 too. Regarding the C-grids, the benefit of parallelization is obvious: PSW outperforms SW by factors of 287 (grid C1) and 996 (C2).

## 5.4 Memory and CPU usage

With increasing grid size, the memory consumption of the solvers increase too. When using the PSW, a trade-off between speed and memory usage can be made. The difference is whether to create a new solver for every iteration of the subgrids or to store and re-use the solvers among different iterations and computations. The former is faster but requires more memory, especially for big grids such as C2. This might cause problems at some point. The latter is slower but uses less memory. As the performance benefit of re-using the solvers showed to be rather small, the re-use property is disabled by default. All results of the previous chapters were measured without re-using the solvers. Unless stated otherwise, all statements refer to the PSWs default behavior. With solver re-usage disabled, memory consumption can further be reduced by decreasing the size of the thread pool. Latter however reduces performance. Halving the thread pool size roughly doubles the solvers' runtime.

Memory consumption was measured by a separate Java thread logging the runtime's memory usage every millisecond. The logger was designed very lightweight so that the logger's own memory consumption is very small. It only keeps the highest measured value, not a full log. The peak memory usage of the solvers for the five subgrids are shown in Table 10. The first line shows the memory consumption of the grid object in Java. It is equal for all solvers.

| | **Grid A** | **Grid B1** | **Grid B2** | **Grid C1** | **Grid C2** |
|---|---:|---:|---:|---:|---:|
| **Initialized Grid-Object** | 0.65 | 9.75 | 9.09 | 280.41 | 319.06 |
| **SW** | 1.95 | 4.02 | 4.33 | 640.97 | 8705.49 |
| **PSW default** | 12.06 | 16.46 | 28.69 | 645.50 | 816.35 |
| **PSW high speed** | 24.32 | 33.28 | 55.04 | 816.99 | 4016.35 |
| **PSW low memory** | 3.93 | 7.18 | 9.96 | 64.04 | 299.88 |
| **NR** | 10.52 | 140.12 | 211.91 | - | - |
| **GS** | 65.00 | 1066.50 | 774.32 | - | - |

**Table 10** *Memory consumption in Megabytes*

Let's first discuss the PSW with default configuration. Memory consumption seems reasonable. PSW's memory consumption is smaller than the consumption of NR and GS, except for grid A. When compared to the SW solver, the PSW solver is better for big grids (C1, C2) while the SW solver is better for smaller grids (A, B1, B2). The turnover is around grid C1 for PSW with default configuration. The PSW solvers memory consumption of grid C2 is below 10% of the memory consumption of the SW solver. As before when comparing speed, the PSW performs best for the very large grids (C1, C2).

The memory consumption of the PSW solver with default configuration depends on the hardware of the system it is running on. Since the thread pool size is chosen to match the number of logical CPU's, the number of simultaneously used solvers is equal to that number. Therefore, memory consumption increases with increasing number of CPUs.

By re-using the subgrid solvers, speed is maximized. This option is denoted "PSW high speed" in the table. Compared to the default configuration, memory consumption rises by at least factor 1.26 (grid C1) up to 4.92 (C2). As the performance benefit of re-using the solvers is rather small (4.6% faster for grid B1), the re-use property is disabled by default.

By disabling re-usage and choosing the thread pool to have a size of one the memory consumption of the solver is minimized. This configuration is denoted "PSW low memory" in the table. Across all grids, memory consumption is reduced significantly. The memory usage is between 10% (grid C1) and 44% (B1) of the default configuration's usage. When comparing the memory saving PSW with the regular SW algorithm, PSW's memory usage is still higher for the smaller grids A, B1 and B2. The practical importance of this option is however limited as parallelization is de facto turned off with a thread pool containing a single thread. Temporal performance is accordingly bad. It is however useful when memory is very scarce and would not suffice to solve the grid using the solvers default configuration.

For the sake of completeness, the table also shows NR and GS' memory usage. It is, except for grid A, higher than SW or any configuration of PSW.

CPU load was measured using Windows' "Performance Monitor". The average usage of 60 seconds was logged while the solvers repeatedly solved grids. The values were averaged over all logical CPU and over the logged duration of 60 seconds. The results are shown in Table 11.

| | **Average CPU usage** | **tested on grid[20]** |
|---|---:|---:|
| **SW Solver** | 15.43% | C2 |
| **PSW Solver (default configuration)** | 92.37% | C2 |
| **NR Solver** | 28.53% | B2 |
| **GS Solver** | 20.24% | B2 |

**Table 11** *Average CPU usage*

The high CPU usage of the PSW shows that parallelization works well and the non-parallelized parts of the solver (the code that runs on the main thread while the thread-pools are idle) account for a small fraction of the CPU-time.

[20] The solvers are tested with different grids such that they preferably do not terminate within the test duration of 60 seconds. If this is not possible, the solver computes the grid over and over.

## 5.5 Optimal subgrid size

Regarding the development of alternative splitting strategies is interesting to evaluate at which subgrid size parallelization is most effective. This will be investigated in this subchapter. The grids used to test the optimal subgrid size are created using the SyntheticGridBuilder introduced in Chapter 4.1. Grids with sizes of 100 up to 4000 buses are created. Every grid is created in ten different versions with varying number of transformers such that the resulting subgrid sizes range from 40 to 400 nodes. A total number of 400 configurations were tested (40 different grid sized, each with ten different subgrid sizes). These configurations are repeatedly computed during two minutes. Between every computation the node powers are changed randomly. The simulation is conducted with the SW and with PSW algorithm. Analogous to Chapter 5.3 the runtime is extrapolated to 35′040 individual calculations, representing the simulation of a year. The results are shown in the heatmap in **Figure *18***. The shown numbers are the runtime of the PSW normalized with the runtime of the SW algorithm for the same configuration. Values smaller than 1 mean that the PSL was faster and vice versa.

For grids up to a size of 800 nodes, the SW solver is clearly better than PSW in its default configuration. Starting at a grid size of 1000 nodes, PSW starts to outperform SW with when choosing a subgrid size around 160. For grids with more than 1800 nodes, PSW outperforms SW almost anytime regardless of the size of the subgrids.

Subgrid size →

Grid size ↓

| | 40 | 80 | 120 | 160 | 200 | 240 | 280 | 320 | 360 | 400 |
|---|---|---|---|---|---|---|---|---|---|---|
| 100 | 3676% | | | | | | | | | |
| 200 | 1389% | 1532% | | | | | | | | |
| 300 | 1168% | 893% | 1404% | | | | | | | |
| 400 | 1727% | 2127% | 1017% | 895% | 641% | | | | | |
| 500 | 710% | 275% | 530% | 926% | 568% | 587% | | | | |
| 600 | 567% | 250% | 540% | 487% | 388% | 389% | 542% | | | |
| 700 | 330% | 170% | 135% | 435% | 379% | 361% | 364% | 400% | | |
| 800 | 432% | 160% | 122% | 127% | 354% | 346% | 322% | 284% | 362% | 409% |
| 900 | 236% | 207% | 92% | 222% | 258% | 205% | 394% | 227% | 313% | 348% |
| 1000 | 170% | 113% | 114% | 82% | 282% | 215% | 254% | 210% | 230% | 345% |
| 1100 | 122% | 85% | 56% | 57% | 52% | 202% | 193% | 226% | 219% | 165% |
| 1200 | 152% | 116% | 78% | 64% | 144% | 147% | 188% | 151% | 205% | 150% |
| 1300 | 127% | 55% | 43% | 46% | 47% | 167% | 189% | 181% | 126% | 277% |
| 1400 | 97% | 79% | 78% | 56% | 116% | 97% | 141% | 169% | 130% | 225% |
| 1500 | 67% | 76% | 77% | 48% | 50% | 127% | 106% | 168% | 112% | 100% |
| 1600 | 64% | 62% | 44% | 36% | 34% | 54% | 92% | 122% | 97% | 100% |
| 1700 | 59% | 42% | 54% | 49% | 42% | 56% | 74% | 103% | 85% | 191% |
| 1800 | 33% | 47% | 28% | 64% | 31% | 43% | 26% | 59% | 64% | 100% |
| 1900 | 31% | 22% | 18% | 16% | 17% | 13% | 33% | 47% | 57% | 56% |
| 2000 | 29% | 27% | 25% | 23% | 24% | 25% | 34% | 30% | 52% | 60% |
| 2100 | 22% | 15% | 22% | 23% | 18% | 14% | 13% | 32% | 47% | 54% |
| 2200 | 28% | 20% | 16% | 13% | 13% | 12% | 11% | 12% | 49% | 42% |
| 2300 | 24% | 16% | 19% | 28% | 20% | 27% | 20% | 15% | 19% | 46% |
| 2400 | 61% | 32% | 50% | 50% | 46% | 30% | 28% | 72% | 90% | 182% |
| 2500 | 110% | 65% | 61% | 67% | 46% | 34% | 34% | 39% | 119% | 99% |
| 2600 | 58% | 67% | 45% | 35% | 49% | 71% | 63% | 46% | 40% | 56% |
| 2700 | 29% | 28% | 36% | 58% | 93% | 52% | 58% | 48% | 103% | 126% |
| 2800 | 82% | 36% | 36% | 60% | 37% | 84% | 77% | 65% | 50% | 37% |
| 2900 | 24% | 30% | 46% | 45% | 45% | 69% | 58% | 34% | 35% | 32% |
| 3000 | 32% | 45% | 30% | 44% | 45% | 45% | 58% | 46% | 41% | 31% |
| 3100 | 19% | 52% | 35% | 36% | 27% | 26% | 21% | 25% | 24% | 84% |
| 3200 | 46% | 41% | 32% | 32% | 28% | 21% | 71% | 44% | 41% | 35% |
| 3300 | 27% | 59% | 28% | 37% | 59% | 45% | 47% | 36% | 29% | 33% |
| 3400 | 27% | 39% | 31% | 26% | 18% | 43% | 55% | 48% | 27% | 26% |
| 3500 | 16% | 37% | 10% | 29% | 29% | 24% | 42% | 48% | 27% | 81% |
| 3600 | 22% | 30% | 42% | 44% | 31% | 36% | 30% | 62% | 26% | 24% |
| 3700 | 14% | 20% | 43% | 22% | 19% | 33% | 39% | 45% | 27% | 41% |
| 3800 | 20% | 17% | 21% | 12% | 13% | 26% | 22% | 18% | 27% | 33% |
| 3900 | 16% | 19% | 29% | 27% | 20% | 35% | 22% | 20% | 16% | 57% |
| 4000 | 29% | 16% | 12% | 18% | 12% | 40% | 29% | 24% | 33% | 21% |

| Legend | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|
| | 20% | 40% | 60% | 80% | 100% | 200% | 300% | 400% | 500% |
| | ↑<br>PSW 5x faster | | | | ↑<br>equal speed | | | | ↑<br>SW 5x faster |

**Figure 18** *Speed comparison of PSW and SW as a function of grid and subgrid size*

## 5.6 Computational Overhead

The computation of a grid in parallel comes with an overhead. This overhead depends on the size of the grid and the amount of splitting points generated, hence the number of transformers in the network. Figure 19 shows the execution times of the five benchmark grids executed on the test computer (Chapter 4.4). Elapsed time is divided into six categories: Splitting, Subgrid Initialization, Node Power Initialization, Calculation, Synchronization and Topology Restoration. A thread pool size of 8 was chosen for the parallelized parts: the subgrid calculations (shown in yellow) and synchronization (shown in red). All categories except for the actual calculation (yellow) can be considered as an overhead of parallelization as these do not arise in the non-parallel solver. Despite the overhead, PSW might be quicker than the sequential SW solver as the grid calculation (yellow) is much quicker as it is executed in parallel.

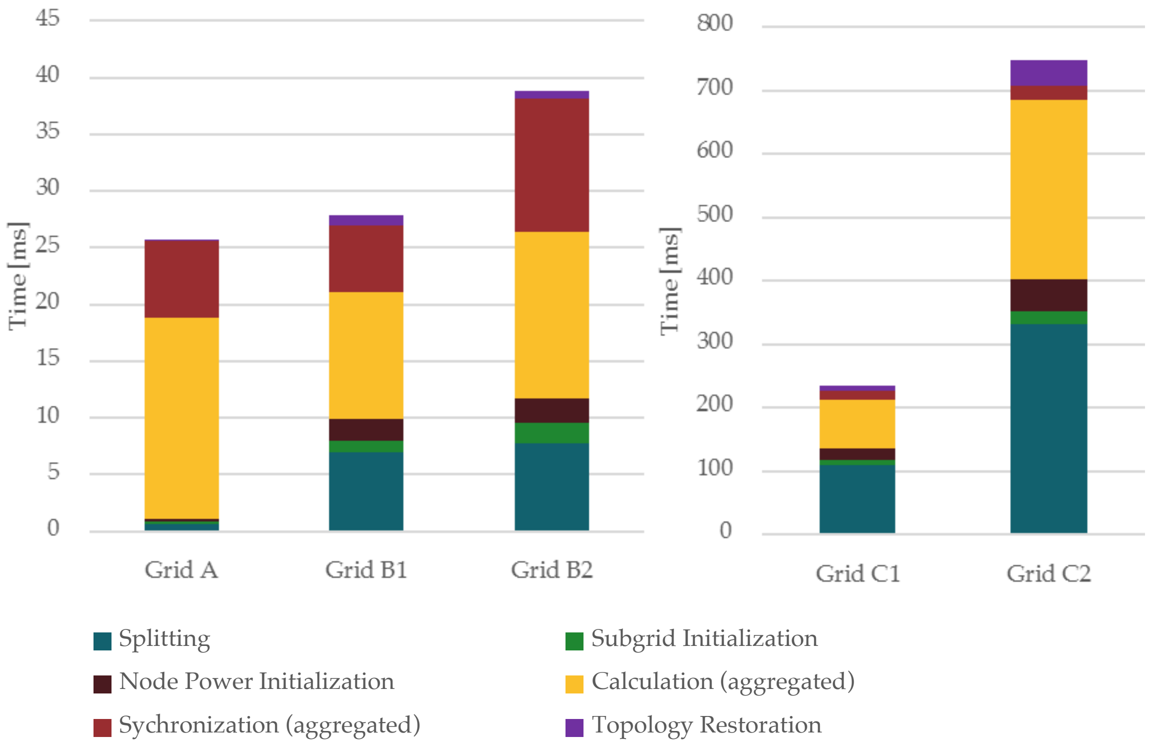


**Figure 19** *Overheads of the parallel algorithm. The shown values are the average of ten computations with outliers filtered out.*

The higher subgrid computation time can be explained by the fact that the SW solver has a constant overhead as well. With a higher number of subgrids this constant overhead occurs more often. The comparison of the grids B1 and B2 shows that the overhead of the PSW solver depends on the size of the subgrids.

When doing a series of computations without topology changes, the splitting and subgrids initialization overheads occur only at the first computation, for all the following computations they are zero. If lines or transformers are connected or disconnected at the substation the splitting and initialization must be repeated.

The computation of the subgrids and synchronization must be performed once per iteration of the parallel solver. The bars in Figure 19 show the aggregated overheads for a computation with a synchronization threshold $\varepsilon_{par}$ of 1E-9. Node power is re-initialized whenever topology changes or changes in the power of loads or generator have occur, which is usually every time. Due to the efficient implementation of node power initialization, its overhead is very small.

# 6 Conclusions & Outlook

In this thesis, a high-level method to parallelize power flow computation has been presented. Experiments with an implementation of the concept showed that the method yields results as accurate as the sequential solvers. A list of convergence thresholds for the investigated solver leading to equivalently accurate results has been presented.

The performance of the parallel implementation was compared to three other solvers. Medium sized networks of around 600 nodes perform best with the parallel solver, when performing a high number of computations without changing the topology. A speedup of 9.25 could be measured when simulating a 600-node grid for an entire year without topology change. However, the sequential solver is faster when doing a single computation. The same applies to calculation series with frequent changes in topology. The parallel solver shows its strength when calculating huge networks with tens of thousands of nodes. The speedup of a grid with 12'000 nodes is 22 for a single calculation and 286 for a simulation series. The bigger the grid, the bigger the speedup. A single calculation of a grid with 47'000 nodes runs 132 times faster using the parallel solver. A simulation series with the same grid runs even 996 times faster with the parallel solver. Small grids, however, do not perform well with the parallel solver due to the big parallelization overhead compared to calculation time. As a rule of thumb one can consider the breakeven point to be between 1000 and 2000 nodes, depending on the topology of the network, in particular on the size of the subgrids.

Parallelization degree is a trade-off that can be chosen to either maximize performance or minimize memory consumption. The former is chosen by default. By choosing the latter, a grid with 47'000 nodes can be solved using as little as 300 megabytes of memory. In the performance configuration, 816 megabytes are needed. Both are low compared to the sequential sweeping solver that requires 8.7 gigabytes to solve the same grid. The memory tests also show that the sweeping solver uses significantly less memory than either Newton Raphson or Gauss Seidl.

The strength of the parallel solver lies in the calculation of big networks. Small networks are solved quickly using the sequential sweeping solver. None of the tested grids perform best with Newton Raphson or Gauss Seidl. A powerful general purpose solver would be a combination of the sequential and parallel implementation of the sweeping algorithm. The solver would dynamically choose either the sequential or parallel implementation depending on the grid size and the expected number of calculations.

Further research should be aimed at improving performance of the parallelized solver. First, not all parts of the solver are parallelized. Splitting for example, is not. Implementing a parallelized splitting would increase the performance for individual computations as well as calculation series with frequent topology changes. Secondly, new splitting strategies should be investigated[21]. A focus could be aiming for optimal subgrid sizes as found in Chapter 5.5. Another approach could be to split meshed grids such that loops are placed within a subgrid. That way, meshed grids could also be solved using the parallel solver.

Furthermore, one could implement the thread pool size to be chosen and changed dynamically depending on the available memory on the computer. Whenever the Java virtual machine risks to run out of memory, the thread pool is decreased to prevent the program from being aborted.

[21] The splitting was implemented in an own class. The advantage of the design is that the splitter can be replaced easily or other splitters can be added as an alternative.

# 7 Bibliography

[1] G. X. Luo and A. Semlyen, “Efficient load flow for large weakly meshed networks,” *IEEE Trans. Power Syst.*, vol. 5, no. 4, pp. 1309–1316, 1990.

[2] F. Teng, “Implementation of a Voltage Sweep Power Flow Method and Comparison with Other Power Flow Techniques,” 2014.

[3] T. Feng and A. J. Flueck, “A message-passing distributed-memory Newton-GMRES parallel power flow algorithm,” *Power Eng. Soc. Summer Meet. 2002 IEEE*, vol. 3, no. c, pp. 1477–1482 vol.3, 2002.

[4] G. Andersson, Lecture notes of Power System Analysis course, 2015.

[5] V. Lenz, “Generation of Realistic Distribution Grid Topologies Based on Spatial Load Maps,” 2015.

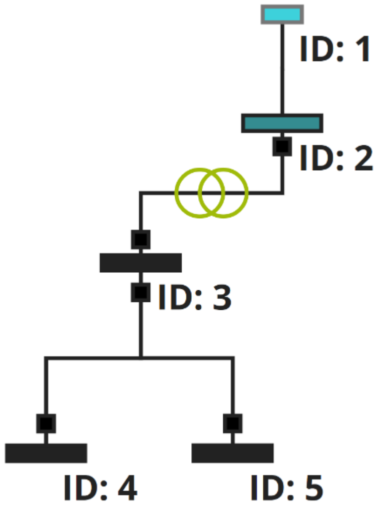
[6] V. Hodge and J. Austin, “A Survey of Outlier Detection Methodologies,” *Artif. Intell. Rev.*, vol. 22, no. 1969, pp. 85–126, 2004.

# Appendix

## A Visual and textual representation of a small grid

Note that the visual representation doesn't contain loads and generators.

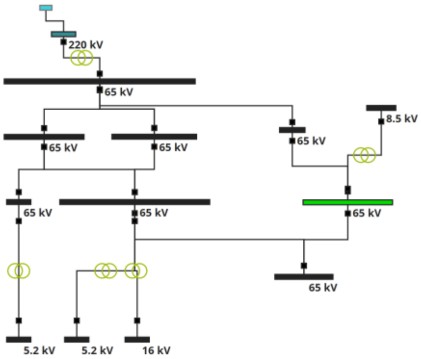
```
<?xml version="1.0"?>
<GRID>
   <BUSBAR_NODE>
      <Node ID="2"  BaseVoltageInKilovolt="100"/>
      <Node ID="3"  BaseVoltageInKilovolt="10"/>
      <Node ID="4"  BaseVoltageInKilovolt="10"/>
      <Node ID="5"  BaseVoltageInKilovolt="10"/>
   </BUSBAR_NODE>
   <LINE>
      <Line ID="6" Bus1ID="3" Bus2ID="4" ResistanceInOhmPerKilome-
      ter="0.74439" ReactanceInOhmPerKilometer="0.00000" LengthInKilome-
      ter="1" MaximumCurrentInAmpere="100.00" ShuntCapacitanceInMicrofar-
      adPerKilometer="0.0" ConnectedAtBus1="true" ConnectedAtBus2="true" />
      <Line ID="7" Bus1ID="3" Bus2ID="5" ResistanceInOhmPerKilome-
      ter="0.74439" ReactanceInOhmPerKilometer="0.00000" LengthInKilome-
      ter="1" MaximumCurrentInAmpere="100.00" ShuntCapacitanceInMicrofar-
      adPerKilometer="0.0" ConnectedAtBus1="true" ConnectedAtBus2="true" />
   </LINE>
   <LOAD>
      <Load ID="9" Bus1ID="4" ActiveLoadInMegawatt="0.5" Reactive-
      LoadInMegavar="0.0" Name="Load1" Connected="true"/>
      <Load ID="10" Bus1ID="5" ActiveLoadInMegawatt="0.25" Reactive-
      LoadInMegavar="0.0" Name="Load2" Connected="true"/>
   </LOAD>
   <TRANSFORMER>
      <Transformer ID="8" Bus1ID="2" Bus2ID="3"
      TransformerRatingInMegavoltampere="0.1"
      MaximumApparentPowerInMegavoltampere="0"
      ShortCircuitVoltageInPercent="4.00" Cop-
      perLossesInPercent="1.00" ConnectedAt-
      Bus1="true" ConnectedAtBus2="true"/>
   </TRANSFORMER>
   <FEEDER>
      <Feeder ID="1" Bus1ID="2" BaseVoltageIn-
      Kilovolt="100.0" AngleInRadians="0.0"
      Name="Feeder" Connected="true"/>
   </FEEDER>
</GRID>
```



# B Grids used to validate Sweeping algorithm

Note that the visual representation doesn't contain loads and generators.

### Grid V1



### Grid V2