Code optimization of algorithms on sparse grids

Niklas Heidenreich
Code optimization of algorithms on sparse grids

Niklas Heidenreich
Bachelorarbeit

Aufgabensteller: Prof. Dr. Christoph Pflaum
Betreuer: Riccarda Scherner, M. Sc.
Erklärung:

Ich versichere, dass ich die Arbeit ohne fremde Hilfe und ohne Benutzung anderer als der angegebenen Quellen angefertigt habe und dass die Arbeit in gleicher oder ähnlicher Form noch keiner anderen Prüfungsbehörde vorgelegen hat und von dieser als Teil einer Prüfungsleistung angenommen wurde. Alle Ausführungen, die wörtlich oder sinngemäß übernommen wurden, sind als solche gekennzeichnet.

Der Universität Erlangen-Nürnberg, vertreten durch den Lehrstuhl für Systemsimulation (Informatik 10), wird für Zwecke der Forschung und Lehre ein einfaches, kostenloses, zeitlich und örtlich unbeschränktes Nutzungsrecht an den Arbeitsergebnissen der Bachelorarbeit einschließlich etwaiger Schutzrechte und Urheberrechte eingeräumt.

Erlangen, den 24. November 2022

..............................................
Abstract

With the computational power of computers increasing year by year, so too do the complexity of the tasks calculated with them. For many research fields, however, more than a simple increase in raw performance is needed to satisfy the considerable runtime requirements of their calculations. An entirely new approach to solving those problems is necessary for these cases. One of these approaches comes in the form of sparse grid solvers. The goal is to reduce the enormous runtime growth connected with the exponential increase in data points needed for higher dimensional computations. Sparse grids allow for a significantly reduced number of grid points when dealing with high dimensions.

This paper aims to reduce the runtime of a prewavelet-based finite element library for the solution of partial differential equations on adaptive sparse grids called Expression Templates for Partial Differential Equations on Sparse Grids (ExPDESG) by Prof. Dr. Christoph Pflaum and Riccarda Scherner implemented in C++. Tested will be multiple optimization techniques, including adjusting the underlying data structure, rewriting functions to use more efficient assembly instructions, and parallelizing the algorithm to run on multiple nodes.
## Contents

1 Introduction ........................................ 7

2 Reference Systems ....................................... 7

3 Performance and efficiency ................................. 8

4 Main data structures of ExPDESG .......................... 8

5 Tree structure ........................................... 9

5.1 depth() .................................................. 11
  5.1.1 Using __builtin_clz() .............................. 11
  5.1.2 Bit-wise XOR instead of subtraction ................. 12
  5.1.3 C++20 ............................................... 13
  5.1.4 Performance ....................................... 14

5.2 nextLeft() ............................................. 15
  5.2.1 Removing redundant code ........................... 15
  5.2.2 Using __builtin_ctz() .............................. 15
  5.2.3 Using __builtin_ffs() .............................. 16
  5.2.4 C++20 ............................................... 16
  5.2.5 Performance ....................................... 17

5.3 nextRight() ............................................ 17
  5.3.1 Removing redundant code ........................... 18
  5.3.2 Using __builtin_ctz() .............................. 18
  5.3.3 Using __builtin_ffs() .............................. 19
  5.3.4 C++20 ............................................... 19
  5.3.5 Performance ....................................... 20

6 Basic grid data structure .................................. 20

6.1 Hash-Multimap Implementation ........................... 21

6.2 Multi-depth hash map ................................... 22

6.3 Restriction .............................................. 23
  6.3.1 Restriction with multi depth hash ................... 23
  6.3.2 Push instead of pull ................................ 24
  6.3.3 Push with inplace .................................. 24
  6.3.4 Performance ....................................... 25
  6.3.5 Further possible optimizations ..................... 26

6.4 Prolongation ............................................ 27
  6.4.1 Prolongation Inplace ................................. 28
  6.4.2 Prolongation using the multi depth hash .......... 29
  6.4.3 Inverted prolongation using the multi depth hash . 29
  6.4.4 Performance ....................................... 30

6.5 MultiLevelVector::setMultiLevelValues() .................. 31

6.6 Reordering .............................................. 32
  6.6.1 Reorder by depth ................................... 32
  6.6.2 Reorder by collision ................................ 33
  6.6.3 Performance ....................................... 34
  6.6.4 Problems .......................................... 34

7 MultiDimCompass ......................................... 34

8 Combined performance improvements ......................... 36
1 Introduction

Nearly all fields of study require more and more computationally intensive simulations. Although the capabilities of modern hardware are ever-increasing, there is a limit to what can be calculated in a reasonable amount of time. This becomes even more apparent with tasks requiring solutions to systems of higher dimensionality. Problems in higher dimensions need exponentially more data points to be calculated (i.e., the curse of dimensionality). This problem prevents full grids from being used for high-dimensional simulations without the runtime becoming unfeasible. Sparse grids use only a fraction of the grid points a full grid would need per dimension. This decrease in the number of required points carries over in every new dimension and, in return, allows the usage of more dimensions while still using a manageable amount of data points. The ExPDESG library uses the prewavelet-based algorithm on sparse grids by Prof. Dr. Christoph Pflaum and Dr. Rainer Hartmann [1] [2] and has been extended by Riccarda Scherner to allow for adaptive sparse grid computations.

This paper aims to improve the performance of this library without changing the underlying algorithms themselves. It will give an overview of the program’s main data structures, focusing on increasing the performance of the implementations. In order to reduce the runtime of the program, several optimization approaches have been applied.

The first optimizations presented in section 5 make use of special hardware instructions to improve the runtime of performance-critical sections. In section 6, a new data structure has been added and used to decrease memory access times and increase cache efficiency. Several functions have been rewritten to make use of this new data structure. Here the performance-intensive operations "restriction" and "prolongation" have been optimized to feature better runtime scaling when used in high dimensionalities. Reordering strategies for the underlying data structure have been tested, and the number of data accesses has been reduced. The section 7 discusses program runtime reduction by using precomputed stencil directions.

Afterward, in section 9, different parallelization techniques were used to allow for even faster calculations. These techniques include parallelization on shared memory systems, multi-node systems, and the Graphics Processing Unit (GPU), followed by parallel hybrid approaches to allow scaling on highly parallel systems.

2 Reference Systems

For measuring the runtime of the program and for performing the benchmarks, a reference system is needed. Table 1 contains a list of all reference systems used in this paper. Small micro-benchmarks involving only the use of a few functions with very small runtimes have been run on the Work system. For highly parallel workloads, the local HPC systems of the NHR@FAU (formerly known as RRZE) have been used.

<table>
<thead>
<tr>
<th>Name</th>
<th>CPU</th>
<th>GPU</th>
<th>misc.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Work</td>
<td>AMD Ryzen 7 2700X 8 cores</td>
<td>NVIDIA GeForce GTX 1080 Ti</td>
<td></td>
</tr>
<tr>
<td>Meggie</td>
<td>2x Intel Xeon E5-2630v4 10 cores</td>
<td>none</td>
<td>per Node ×728</td>
</tr>
<tr>
<td>TinyGPU</td>
<td>2x AMD Rome 7662 128 cores</td>
<td>4x NVIDIA A100 SXM4/Nvlink (40GB) per Node</td>
<td></td>
</tr>
<tr>
<td>Fritz</td>
<td>2x Intel Xeon Platinum 8360Y 36 cores</td>
<td>none</td>
<td>per Node ×992</td>
</tr>
</tbody>
</table>

Table 1: Specifications of the reference systems used in this paper.

Most of the optimizations have been tested with the help of Google’s micro benchmarking library [3]. The used compiler versions and runtime environments are listed in table 2.
<table>
<thead>
<tr>
<th>Name</th>
<th>OS</th>
<th>Compiler-Version</th>
<th>MPI-Version</th>
<th>CUDA-Version</th>
</tr>
</thead>
<tbody>
<tr>
<td>Work</td>
<td>Ubuntu 20.04</td>
<td>GCC 11.1.0</td>
<td>openmpi/4.1.2</td>
<td>none</td>
</tr>
<tr>
<td>Work with CUDA</td>
<td>Ubuntu 20.04</td>
<td>GCC 8.4.0</td>
<td>openmpi/4.0.3</td>
<td>cuda/10.1</td>
</tr>
<tr>
<td>Meggie</td>
<td>AlmaLinux8</td>
<td>GCC 12.1.0</td>
<td>openmpi/4.1.3</td>
<td>none</td>
</tr>
<tr>
<td>TinyGPU</td>
<td>Ubuntu 20.04</td>
<td>GCC 9.4.0</td>
<td>intelmpi/2019.8</td>
<td>cuda/11.6.1</td>
</tr>
<tr>
<td>Fritz</td>
<td>AlmaLinux8</td>
<td>Intel 2022.1.0</td>
<td>intelmpi/2021.7.0</td>
<td>none</td>
</tr>
</tbody>
</table>

Table 2: Specifications of the software used by the reference systems in this paper.

3 Performance and efficiency

This paper differentiates between performance and efficiency. Here we define efficiency as the amount of work that has to be done to complete a specific task and performance as the amount of time required to do this work.

Reducing the runtime by using an improved mathematical algorithm that computes the same or similar results while using fewer steps to do so will reduce the amount of work that has to be done at runtime and is therefore considered an improvement in efficiency. Using advanced Central Processing Unit (CPU) hardware instructions or executing calculations in parallel will not reduce the work but rather allow to complete it faster and is, therefore, an performance optimization. Note that because efficiency is unrelated to performance, a more efficient algorithm can also lead to an increased runtime if it has a worse performance than the less efficient algorithm.

While it is possible to mathematically prove that an algorithm is the most efficient way of solving a particular problem there is no most performant implementation in the general sense. Any algorithm will reach its most performant state if executed infinitely fast, resulting in a runtime of zero.

This paper aims to decrease the runtime of calculations of the ExPDESG library by improving their performance. This can be achieved by using more hardware-specific instructions, preventing unnecessary operations, restructuring the data, and parallelizing the calculations. This paper does not aim to rewrite the algorithmic part of this library.

4 Main data structures of ExPDESG

The main focus of this paper is to increase the program’s performance by optimizing calculations and data access. Therefore it is mandatory to understand the underlying data structures used. This section will give a small overview of some of the different data structures used by this implementation. The nature of an adaptive grid prevents simple array implementations. New points can be added at any time in the calculation, creating the need for an underlying data system that can find nearby points without having to spend a lot of time searching. The data access can be separated into three different structures:

Tree structure: The tree structure allows for easy point generation and is able to efficiently find neighboring points.

SparseGrid structure: The sparse grid saves every generated point inside a custom hashmap and acts as a mapping from the tree point to a memory index.

SparseVector structure: The sparse vector contains the actual values of each point. There can be multiple sparse vectors per sparse grid.
Figure 1: An example process of the retrieval of data from a neighboring point.

5 Tree structure

The program uses a binary tree data structure to uniquely label each point of the grid. As well as create new points or find existing ones. Hereby a point represents an n-dimensional location inside the grid consisting of n indices where an index is simply a scalar value between 0 and 1. The tree structure allows fast traversal of such indices. For the adaptive sparse grid implementation, a reliable and fast way of increasing the point resolution of the grid is necessary. Multiple operations need to operate on points belonging to the same grid resolution requiring a data structure that allows for easy access to such points. The here used tree structure enables such behavior by doubling the precision of its entries at increasing depths. Figure 2 shows an example tree and its conversion into world position in 1-dimension.

The generated points are then passed to the sparse grid, where they are hashed and organized. To better visualize the labeling system, all the following numbers with an index of 2 are binary, and all numbers without an index represent decimal numbers.

Given an index, finding the left index of the next higher depth is represented by a simple shift of its memory representation to the left. Similarly, finding the right one is achieved by shifting the index once to the left and adding one.

Following \(\rightarrow\) represents a mapping from one number system to another. The numbers \(0\rightarrow 0\) and \(1\rightarrow 1\) are reserved for the left and right most points. The index \(10\rightarrow 0.5\). In the next depth \(100\rightarrow 0.25\) and \(101\rightarrow 0.75\) (see Table 3).

Figure 2: Example 1-dimensional tree structure of indices. Left the internal binary memory representation and right converted to decimal numbers representing world position.
<table>
<thead>
<tr>
<th>Binary</th>
<th>Decimal</th>
</tr>
</thead>
<tbody>
<tr>
<td>$0_2$</td>
<td>$0$</td>
</tr>
<tr>
<td>$1_2$</td>
<td>$1$</td>
</tr>
<tr>
<td>$10_2$</td>
<td>$0.5$</td>
</tr>
<tr>
<td>$100_2$</td>
<td>$0.25$</td>
</tr>
<tr>
<td>$101_2$</td>
<td>$0.75$</td>
</tr>
<tr>
<td>$1000_2$</td>
<td>$0.125$</td>
</tr>
<tr>
<td>$1001_2$</td>
<td>$0.375$</td>
</tr>
<tr>
<td>$1010_2$</td>
<td>$0.625$</td>
</tr>
<tr>
<td>$1011_2$</td>
<td>$0.875$</td>
</tr>
</tbody>
</table>

Table 3: Conversion table from binary representation to decimal world position

For d-dimensional calculations, a data point is uniquely identifiable by d different indices, one for every dimension.

In order to get elements within the tree structure, several methods are provided. These methods are used every time an index, or its neighbors are needed and are therefore called extremely often. That is why it is essential not to lose performance here unnecessarily. Methods like $\text{nextLeft}()$, $\text{nextRight}()$, and $\text{depth}()$ are all using loops to calculate the result. Utilizing specialized CPU-opcodes makes it possible to rewrite these methods without using a single loop and therefore improve the performance considerably. In the following sections, the runtime of three performance-critical methods is optimized. The optimizations for all three methods are similar. In order to prevent repetition, the exact description of the optimization of following similar methods will not be very detailed. But will nevertheless be presented for the sake of completeness.

**Level:** Some runtime tests feature a level value in their description. A grid of level $n$ contains all points that have a level equal to or less than $n$. Here the level of an index is equal to its depth. And the level of a point is defined as the $L_1$-Norm of the levels of its indices. Therefore, the grid shown in Figure 2 is a 1-dimensional grid of level 3. Such a grid does not feature any adaptive regions or points and is called a regular sparse grid. The grid shown in Figure 3 is a 2-dimensional grid of level 6. Its the center point is actually located at level 2. This is because it consists of the indices $(0010_2, 0010_2)$ that are both on level 1 making their $L_1$-Norm equal 2.

![Figure 3: Example of a regular sparse grid of level 6 consisting of 129 points.](image-url)
5.1 depth()

The depth() method is used to calculate the depth of the given index. This is used all around the code in performance-critical places. For example, it is used d times to calculate the depth of a d-dimensional point. This is done for every single point. Optimizing this method is, therefore, really important.

```c
unsigned depth(unsigned index) {
    if(index==0) return 0;
    unsigned de = 0;
    while(index!=1) {
        de = de + 1;
        index = index >> 1;
    }
    return de;
}
```

---

5.1.1 Using __builtin_clz()

The original depth() method uses a while loop to calculate the depth of the given index. This loop would be called n-times for indices of depth n. This means that this method has a growth rate of $O(n)$. By using special Assembly x86 (ASMx86) instructions, it is possible to rewrite this method to have a growth rate of $O(1)$ and reduce the number of needed operations considerably.

This can be achieved by using the fact that in order to calculate the depth of an index, it is only necessary to know the position of the first 1 bit in the index. Modern x86 CPUs have the built-in hardware instruction Count-Leading-Zeros (clz). C++ does not provide a way to use this instruction, but GCC provides __builtin_clz(). This built-in function will call the ASMx86 instruction clz that counts the leading zeros of the input. By using this function and then subtracting the result from 32 (number of bits of a 32bit int) we get the position of the first 1 bit. The depth of the index is then simply the position of the first 1 bit minus one. Thus, \( \text{depth} = 31 - \text{__builtin_clz(index)} \).

Instead of using the clz instruction, the position can be computed even faster by using Bit-Scan-Reverse (bsr), which gives the position of the first 1 bit in an int. This optimization is usually done automatically by the compiler. The actual depth can then be obtained by a call to bsr and simply adding 1. Unfortunately bsr is not callable as a C function. The compiler, however, notices this optimization itself and will use the bsr call instead of the call to clz (see Figure 5). The growth rate of the original function was $O(n)$; now, it is $O(1)$. This comes at the cost of this function only working on CPUs that support the now needed ASMx86 call to bsr and on unsigned integers with size 32bit.

```c
unsigned depth(unsigned index) {
    if(index==0) return 0;
    return 31 - __builtin_clz(index);
}
```

Figure 4: The original depth() method.

Figure 5: Using __builtin_clz(). Notice how the compiler automatically optimizes the call to clz to a call to bsr.
5.1.2 Bit-wise XOR instead of subtraction

Now the only operations left are the zero-check and the subtraction. The zero-check is necessary because \texttt{__builtin_clz()} is not defined for zero as input. The subtraction can still be optimized by replacing it with a bit-wise XOR operator. This operator will call the logical XOR for every bit of both numbers. If the number that gets subtracted from is a power of two minus one ($2^x - 1$), than all bits smaller than that power will be set to 1. Applying the bit-wise XOR with such a number will set all values set to 1 in both numbers to 0, effectively subtracting them.

**Definition.** We define the XOR operator $\oplus$ of two elements of $\{0, 1\}$ as

$$0 \oplus 0 := 0, \quad 0 \oplus 1 := 1, \quad 1 \oplus 0 := 1 \quad \text{and} \quad 1 \oplus 1 := 0.$$

This definition of the XOR-operator can be extended to arbitrary natural numbers in binary representation by

$$\sum_{k=0}^{\infty} a_k 2^k \oplus \sum_{k=0}^{\infty} b_k 2^k := \sum_{k=0}^{\infty} (a_k \oplus b_k) 2^k,$$

where $a_k, b_k \in \{0, 1\}$, for $k \in \mathbb{N}_0$.

**Proposition.** For $n, m \in \mathbb{N}_0$, one has $2^n - 1 - m = (2^n - 1) \oplus m$.

**Proof.** Given $n, m \in \mathbb{N}_0$ such that $m \leq 2^n - 1$ we want to show that

$$2^n - 1 - m = (2^n - 1) \oplus m.$$

One can rewrite $m$ in its binary representation $m_k$ given by

$$m = \sum_{k=0}^{n-1} m_k 2^k, \quad \text{where} \quad m_k \in \{0, 1\}.$$

Through geometric summation, one obtains

$$2^n - 1 = \sum_{k=0}^{n-1} 2^k.$$

Hence one has

$$2^n - 1 - m = \sum_{k=0}^{n-1} 2^k - \sum_{k=0}^{n-1} m_k 2^k = \sum_{k=0}^{n-1} (1 - m_k) 2^k$$

which together with $1 \oplus b = 1 - b$, for $b \in \{0, 1\}$, implies

$$(2^n - 1) \oplus m = \sum_{k=0}^{n-1} 2^k \oplus \sum_{k=0}^{n-1} m_k 2^k = \sum_{k=0}^{n-1} (1 \oplus m_k) 2^k = \sum_{k=0}^{n-1} (1 - m_k) 2^k.$$  

\[\square\]

A bitwise operation only has to compare the bits of both numbers with each other. These comparisons are independent and can therefore be calculated in parallel. Furthermore, because bitwise operations are so basic, they are implemented in hardware directly, making them extremely fast. The integer addition and subtraction instructions both use the same underlying hardware. Here after two bits are added, the result might contain a carry bit. Meaning the result of the next bit addition is dependent on the previous one. This makes bitwise operations faster than basic arithmetic addition or subtraction.
unsigned depth(unsigned index) {
    if(index==0) return 0;
    return __builtin_clz(index) ^ 31;
}

bsr edi, edi  // Sets ZF
mov eax, 0  // Checks ZF
cmovne eax, edi  // Checks ZF
ret

Figure 6: Using XOR instead of subtraction reduces the execution time even further. Here, in contrast to the previous optimization, the xor edx, edx call has been replaced with mov edx, 0. Both calls set edx to 0. While xor edx, edx is generally the faster operation, it is not used here because it would set the zero flag (ZF), which is needed for cmovne.

5.1.3 C++20

C++20 introduced a new function called std::bit_width() that returns the minimal number of bits necessary to store the given number. This function has the benefit of also working for non-32bit numbers.

The index’s depth depends only on its highest set 1 bit. Subtracting one from the result of this function will give us the depth of the given index. This will underflow for the input index 0, so an extra zero-check is required.

The need for a zero check and subtraction by one can be circumvented by a single right shift, reducing the std::bit_width() function’s result by one and the fact that for unsigned integers, the right-shift operator >> has these properties (1 >> 1) = (0 >> 1) = 0. The resulting function “C++20+Shift” will only do a single shift on the index and then call the std::bit_width() function, which is optimized really well by current compilers.

unsigned depth(unsigned index) {
    if(index==0) return 0;
    return std::bit_width(index)-1;
}

unsigned depth_shift(unsigned index){
    return std::bit_width(index>>1);
}

Figure 7: On the left the depth() function using the std::bit_width() function defined in C++20. And on the right the same function using the right shift operator in order to skip the zero check.
5.1.4 Performance

<table>
<thead>
<tr>
<th>Input range:</th>
<th>Name:</th>
<th>Runtime (in ns):</th>
<th>Runtime (relative):</th>
</tr>
</thead>
<tbody>
<tr>
<td>Small depths: [0, 2^5]</td>
<td>Nothing:</td>
<td>15.5</td>
<td>-</td>
</tr>
<tr>
<td></td>
<td>Original:</td>
<td>60.7</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td>CLZ:</td>
<td>40.4</td>
<td>0.67</td>
</tr>
<tr>
<td></td>
<td>XOR:</td>
<td>38.8</td>
<td>0.64</td>
</tr>
<tr>
<td></td>
<td>C++20:</td>
<td>40.9</td>
<td>0.67</td>
</tr>
<tr>
<td></td>
<td>C++20+shift:</td>
<td>46.7</td>
<td>0.77</td>
</tr>
<tr>
<td>High depths: [2^10, 2^30]</td>
<td>Nothing:</td>
<td>283,811,568</td>
<td>-</td>
</tr>
<tr>
<td></td>
<td>Original:</td>
<td>14,030,000,000</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td>CLZ:</td>
<td>1,135,422,039</td>
<td>0.08</td>
</tr>
<tr>
<td></td>
<td>XOR:</td>
<td>1,106,819,455</td>
<td>0.08</td>
</tr>
<tr>
<td></td>
<td>C++20:</td>
<td>1,147,326,142</td>
<td>0.08</td>
</tr>
<tr>
<td></td>
<td>C++20+shift:</td>
<td>1,378,375,951</td>
<td>0.09</td>
</tr>
<tr>
<td>Realistic range: [0, 2^{15}]</td>
<td>Nothing:</td>
<td>8,429</td>
<td>-</td>
</tr>
<tr>
<td></td>
<td>Original:</td>
<td>147,065</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td>CLZ:</td>
<td>38,549</td>
<td>0.26</td>
</tr>
<tr>
<td></td>
<td>XOR:</td>
<td>34,576</td>
<td>0.24</td>
</tr>
<tr>
<td></td>
<td>C++20:</td>
<td>36,036</td>
<td>0.25</td>
</tr>
<tr>
<td></td>
<td>C++20+shift:</td>
<td>51,580</td>
<td>0.35</td>
</tr>
</tbody>
</table>

Table 4: [Ref. system: Work (Table 1)] Runtime measurements of the different depth() implementations. The runtime of these functions is too small for accurate measurements. So the benchmarks were done by iterating over a number of indices given by the corresponding range. The “Nothing” benchmark times represent a for loop that does not call any depth() implementation. These “Nothing” times show that the execution times for the new depth() functions are so low that the for loop iteration itself takes a considerable percentage of runtime.

Looking at the runtime results from Table 4, the XOR-optimized version is the fastest by a small margin. With the new C++20 std::bit_width() functions offering similar performance. Interestingly the C++20 version using the shift operator performs significantly worse than its counterpart.

The growth rate of the new optimized functions is now O(1) instead of the originals’ O(n). This can easily be seen when looking at figure 8.

![Figure 8](image)

Figure 8: [Ref. system: Work (Table 1)] [Raw data: Table 24] Runtimes in ns of the different depth(index) implementations for different input indices. The runtime of these tests is measured for 2^{16} iterations. The original implementation suffered from rising runtime cost for higher input indices. This has been resolved in all new optimizations.
5.2 nextLeft()

The **nextLeft()** method returns the next index left of the given one. The returned index can have any depth smaller or equal to the depth of the given index. This method is used every time the values of neighbors are required, for example, in stencil calculation.

```c
unsigned nextLeft(unsigned index) {
    if (index==0)
        return 0;
    if ((index&1)==1)
        return (index>>1);
    while ((index&1)==0) {
        index = index >> 1;
    }
    return (index>>1);
}
```

![Figure 9: The original unoptimized nextLeft() function.](image)

The original method first checks if the given index is zero. If the index is not zero, it is checked if the trailing bit is set to 1. That means that the index is the right child of its parent. Looking at the tree structure (see Figure 2), it is clear that the parent has to be the index that has the next left world position. If that is the case, the index can just be shifted one to the right to get the value of the parent. The last case will shift the index one to the right while the last bit is set to 0.

5.2.1 Removing redundant code

The first optimization can be performed by removing redundant or duplicate code. Here the second if-clause, checking if the last bit of the index is 1, is entirely redundant. By removing that check, not only does the code become more readable, but the performance may also improve by a tiny amount.

```c
unsigned nextLeft(unsigned index) {
    if (index==0)
        return 0;
    while ((index&1)==0) {
        index = index >> 1;
    }
    return (index>>1);
}
```

![Figure 10: The cleaned up version of the original nextLeft() function.](image)

5.2.2 Using **__builtin_ctz()**

Looking at the tree structure (Figure 2) it can be seen, that algorithmically the actual purpose of the **nextLeft()** method is to shift a binary number to the right until its first 1-bit is shifted. The performance-intensive while-
loop can therefore be removed by using a specialized assembly instruction that returns the position of the least significant 1-bit of a number. By applying the same strategy as with the \texttt{depth()} method, it is possible to use the built-in compiler method \texttt{__builtin\_ctz()} that allows usage of the assembly instruction \texttt{Count-Trailing-Zeros (ctz)}. It returns the amount of trailing 0 bits of a 32bit int. Adding one to the result of \texttt{__builtin\_ctz()} should therefore result in the correct shift amount.

\texttt{__builtin\_ctz():} Returns the number of trailing 0-bits in x, starting at the least significant bit position. If x is 0, the result is undefined. [4, Chapter 6.59]

\begin{verbatim}
#include<iostream>

using namespace std;

unsigned nextLeft(unsigned index) {
  if(index==0 || index==(1<<31))
    return 0;
  unsigned firstRight;
  firstRight = __builtin\_ctz(index);
  return index >> (firstRight + 1);
}

int main() {
  unsigned index = 123456789;
  unsigned shiftAmount = nextLeft(index);
  cout << "The shift amount is: " << shiftAmount << endl;
  return 0;
}
\end{verbatim}

Figure 11: Using \texttt{__builtin\_ctz()}. The check for index==\(1\ll31\) has to be done because the result of \texttt{__builtin\_ctz(1\ll31)} is 31. That results in the line \texttt{return index \textgreater\textless 31 + 1} shifting a 32-bit unsigned integer by 32 bits, leading to undefined behavior.[5, Chapter 6.5.7]

5.2.3 Using \texttt{__builtin\_ffs()}

The \texttt{__builtin\_ctz()} version of the code needs to check if the given index is 0. It is possible to remove this zero check by using the built-in function \texttt{__builtin\_ffs()}. Short for Find-First-Set (ffs).

\texttt{__builtin\_ffs():} Returns one plus the index of the least significant 1-bit of x, or if x is zero, returns zero. [4, Chapter 6.59]

\begin{verbatim}
#include<iostream>

using namespace std;

unsigned nextLeft(unsigned index) {
  if(index==(1<<31))
    return 0;
  unsigned firstRight;
  firstRight = __builtin\_ffs(index);
  return index >> firstRight;
}

int main() {
  unsigned index = 123456789;
  unsigned shiftAmount = nextLeft(index);
  cout << "The shift amount is: " << shiftAmount << endl;
  return 0;
}
\end{verbatim}

Figure 12: Using \texttt{__builtin\_ffs()}. The \texttt{ffs} instruction allows skipping the zero check.

5.2.4 \texttt{C++20}

\texttt{C++20} introduces the function \texttt{std::countr\_zero()}. It returns the number of trailing zeros of the given argument. This also works for unsigned long datatypes. This new function also no longer needs the zero check.
unsigned nextLeft(unsigned index) {
    if (index == (1<<((sizeof(index)*8)-1)) )
        return 0;
    return index >> (std::countr_zero(index)+1);
}

unsigned nextLeft_shift(unsigned index) {
    if (index == (1<<((sizeof(index)*8)-1)) )
        return 0;
    return index >> (std::countr_zero(index<<1));
}

Figure 13: Using std::countr_zero(). The check for undefined shift behavior is still required.

5.2.5 Performance

<table>
<thead>
<tr>
<th>Input range:</th>
<th>Name:</th>
<th>Runtime (in ns):</th>
<th>Runtime (relative):</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Nothing:</td>
<td>15.0</td>
<td>-</td>
</tr>
<tr>
<td>Small depths: [0, 2^5]</td>
<td>Original:</td>
<td>35.2</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td>Cleaned:</td>
<td>30.4</td>
<td>0.86</td>
</tr>
<tr>
<td></td>
<td>CTZ:</td>
<td>27.0</td>
<td>0.77</td>
</tr>
<tr>
<td></td>
<td>FFS:</td>
<td>23.4</td>
<td>0.66</td>
</tr>
<tr>
<td></td>
<td>C++20:</td>
<td>26.8</td>
<td>0.76</td>
</tr>
<tr>
<td></td>
<td>C++20+shift:</td>
<td>23.7</td>
<td>0.67</td>
</tr>
<tr>
<td></td>
<td>Nothing:</td>
<td>274334724</td>
<td>-</td>
</tr>
<tr>
<td>High depths: [2^20, 2^30]</td>
<td>Original:</td>
<td>1679573529</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td>Cleaned:</td>
<td>1477744914</td>
<td>0.88</td>
</tr>
<tr>
<td></td>
<td>CTZ:</td>
<td>708432534</td>
<td>0.42</td>
</tr>
<tr>
<td></td>
<td>FFS:</td>
<td>705010180</td>
<td>0.42</td>
</tr>
<tr>
<td></td>
<td>C++20:</td>
<td>698514762</td>
<td>0.41</td>
</tr>
<tr>
<td></td>
<td>C++20+shift:</td>
<td>642727101</td>
<td>0.38</td>
</tr>
<tr>
<td></td>
<td>Nothing:</td>
<td>8392</td>
<td>-</td>
</tr>
<tr>
<td>Realistic range: [0, 2^15]</td>
<td>Original:</td>
<td>49092</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td>Cleaned:</td>
<td>50636</td>
<td>1.03</td>
</tr>
<tr>
<td></td>
<td>CTZ:</td>
<td>22038</td>
<td>0.45</td>
</tr>
<tr>
<td></td>
<td>FFS:</td>
<td>21518</td>
<td>0.44</td>
</tr>
<tr>
<td></td>
<td>C++20:</td>
<td>21413</td>
<td>0.44</td>
</tr>
<tr>
<td></td>
<td>C++20+shift:</td>
<td>17135</td>
<td>0.35</td>
</tr>
</tbody>
</table>

Table 5: [Ref. system: Work (Table 1)] Runtime measurements of the different nextLeft() implementations. The runtime of these function is so small that normal benchmarking no longer returns usable results. Therefore to function have been called repeatedly in a loop. Here the "Nothing" benchmarking times represent the time it took for a empty loop to execute. Looking at these times one can see that the new implementations are so fast that the loop execution in itself takes a large part of the total runtime.

Table 5 shows that the C++20 implementations feature similar performance as the optimized versions. With the C++20+shift version being noticeably faster than its non shift C++20 counterpart. The FFS version is faster than the CTZ version for small depths but looses most of its advantage for bigger numbers. All these performance numbers should be taken with caution as their execution time is so fast that the loop that calls these function repeatedly consumes a large amount of the total runtime.

5.3 nextRight()

There are a lot of similarities between nextLeft() and nextRight(). Slight differences in both methods’ implementations lead to different possible optimizations. This section will give a small overview of possible optimizations and their performance.
Like the `nextLeft()` function, `nextRight()` returns the next index right of the given one. The returned index can have any depth smaller or equal to the depth of the given index. This method is used every time the values of neighbors are required, for example, in stencil calculation.

```c
unsigned nextRight(unsigned index) {
    if((index&1)==0) return (index>>1);
    index = index>>1;
    while((index&1)==1) {
        index = index >> 1;
    }
    return (index>>1);
}
```

Figure 14: The original unoptimized `nextRight()` function.

### 5.3.1 Removing redundant code

Like in the `nextLeft()` function, the first optimization can be performed by removing redundant or duplicate code. Here the second if-clause, checking if the last bit of the index is 1, is entirely redundant. By removing that check, the performance will improve by a tiny amount.

```c
unsigned nextRight(unsigned index) {
    while((index&1)==1) {
        index = index >> 1;
    }
    return (index>>1);
}
```

Figure 15: The cleaned up version of the original `nextRight()` function.

### 5.3.2 Using `__builtin_ctz()`

The `nextRight()` method uses the same algorithm as the `nextLeft()` one but with a bit-wise inverted input. This allows usage of the same optimization techniques as for the `nextLeft()` method. The check for an undefined shift if `__builtin_ctz()` returns 31 still has to be done. Because the index will be inverted before being passed to `__builtin_ctz()` instead of checking for `(1<<31)` it is necessary to check for its inverse `~(1<<31)`.

```c
unsigned nextRight(unsigned int):
    mov eax, edi
    shr eax
    and edi, 1
    jne .L12
    jmp .L1
    .L4:
    shr eax
    .L12:
    test al, 1
    jne .L4
    shr eax
    .L1:
    ret
```
unsigned nextRight(unsigned index) {
    if (index == ~(1<<31)) return 0;
    unsigned inverted = ~index;
    unsigned firstRight = __builtin_ctz(inverted);
    return index >> (firstRight + 1);
}

5.3.3 Using __builtin_ffs()

And just like with the nextLeft() method it is possible to use the ffs instruction to further improve performance. The built-in compiler function __builtin_ffs() returns the first set 1-bit in the given index. This also allows for the zero check to be skipped because __builtin_ffs() already supports zero as input.

unsigned nextRight(unsigned int) {
    if (index == -(1<<31)) return 0;
    unsigned inverted = ~index;
    unsigned int firstRight = __builtin_ffs(inverted);
    return index >> firstRight;
}

5.3.4 C++20

C++20 introduces the function std::countr_one(). It returns the number of trailing ones of the given argument. This also works for unsigned long datatypes. This new function also no longer needs the zero check.
```cpp
unsigned nextRight(unsigned index) {
    if (index == ~(1<<((sizeof(index)*8)-1)))
        return 0;
    return index >> (std::countr_one(index)+1);
}

unsigned nextRight_shift(unsigned index) {
    if (index == ~(1<<((sizeof(index)*8)-1)))
        return 0;
    return index >> (std::countr_one(index<<1));
}
```

Figure 18: Using `std::countr_one()`. The check for undefined shift behavior is still required.

### 5.3.5 Performance

<table>
<thead>
<tr>
<th>Input range:</th>
<th>Name:</th>
<th>Runtime (in ns):</th>
<th>Runtime (relative):</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Nothing:</td>
<td>15.2</td>
<td>-</td>
</tr>
<tr>
<td>Small depths: [0, 2^4]</td>
<td>Original:</td>
<td>45.6</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td>CTZ:</td>
<td>26.0</td>
<td>0.57</td>
</tr>
<tr>
<td></td>
<td>FFS:</td>
<td>25.1</td>
<td>0.55</td>
</tr>
<tr>
<td></td>
<td>C++20:</td>
<td>23.6</td>
<td>0.52</td>
</tr>
<tr>
<td></td>
<td>C++20+shift:</td>
<td>23.0</td>
<td>0.50</td>
</tr>
<tr>
<td>High depths: [2^20, 2^30]</td>
<td>Nothing:</td>
<td>277,226,902</td>
<td>-</td>
</tr>
<tr>
<td></td>
<td>Original:</td>
<td>1,684,060,055</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td>CTZ:</td>
<td>665,379,593</td>
<td>0.40</td>
</tr>
<tr>
<td></td>
<td>FFS:</td>
<td>666,680,432</td>
<td>0.40</td>
</tr>
<tr>
<td></td>
<td>C++20:</td>
<td>661,621,600</td>
<td>0.39</td>
</tr>
<tr>
<td></td>
<td>C++20+shift:</td>
<td>662,740,859</td>
<td>0.39</td>
</tr>
<tr>
<td>Realistic range: [0, 2^{15}]</td>
<td>Nothing:</td>
<td>8,358</td>
<td>-</td>
</tr>
<tr>
<td></td>
<td>Original:</td>
<td>51,718</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td>CTZ:</td>
<td>17,088</td>
<td>0.33</td>
</tr>
<tr>
<td></td>
<td>FFS:</td>
<td>17,034</td>
<td>0.33</td>
</tr>
<tr>
<td></td>
<td>C++20:</td>
<td>17,480</td>
<td>0.34</td>
</tr>
<tr>
<td></td>
<td>C++20+shift:</td>
<td>17,280</td>
<td>0.33</td>
</tr>
</tbody>
</table>

Table 6: [Ref. system: Work (Table 1)] Runtime measurements of the different `nextRight()` implementations. The "Nothing" category shows the runtime of the for-loop that calls the `nextRight()` implementations itself. One can see that the methods are so fast that this for loop takes a considerable amount of the benchmarks runtime.

As can bee seen in Table 6 all tested optimizations reach similar levels of performance. This can most likely be attributed to the compiler already optimizing the code heavily by itself. The real performance differences between the presented optimizations can better be shown by turning compiler optimizations off. As the goal is to get the fastest code, checking the performance without compiler optimizations would defeat the purpose when the program is always compiled with optimizations turned on.

### 6 Basic grid data structure

The library differentiates between the sparse grid and sparse vectors. The sparse grid contains a mapping from the tree structure point representation to a memory location. These memory locations can then be used to access the value of a point in a sparse vector.

In order to get values for points in the sparse grid, it is necessary for a mapping "point ↔ memory location" to exist. The adaptive sparse grid is a non-uniform adaptive grid where new points can be added at every time to increase precision. This feature makes simple multidimensional arrays as storage unfeasible. Although the number of points, and therefore the size of the array, are known, they can change over time. It is, therefore, not easily possible to map all points efficiently to unique and consecutive memory locations. Instead, a
multimap is used to store the point to memory mapping. A multimap allows data with the same hash to be stored in the same hash container and not overwrite each other. Each point is hashed by a dedicated hashing function and then compared to all other points that have received the same hash. Once the point has been found inside the HashMap, it is indexed and can be used to identify it uniquely. This index is then used to find the correct value inside a sparse vector. The multimap approach allows the usage of a hashing function that does not need to map uniquely. Internally the multimap can still be written to produce a consecutive memory mapping.

6.1 Hash-Multimap Implementation

The nature of the underlying adaptive sparse grid logic prevents the program from having a simple array as a data structure to store all points. The sparse grid makes accessing points in a typical multidimensional array fashion far more complex, and the adaptive part can create new points at any depth, making a structured array unfeasible.

The here-used multimap implementation consists of a primary and a secondary table. Given data will first be hashed by a hashing function into the primary table. The primary table stores at what location in the secondary table the first element with the same hash can be found. If there never was an element with the same hash, the primary table contains zero. In that case, the new data will be stored in the next available free space in the second table, and the location will be stored in the primary table. The secondary table contains the index of the next element in the secondary table that maps to the same location in the primary table. Suppose a hash was already taken in the primary table. In that case, the secondary table will be traversed until the last entry is found. Then the data will be mapped to the next free space in the secondary table, and the position will be written to the last found element.

Figure 19: The C++ multimap implementation featuring a primary and secondary table.

Figure 19 shows the used C++ implementation. Here the primary table contains the index of the secondary table + 1 or 0 if unused. The secondary table contains the index of the next entry with the same primary table mapping + 2, 0 if unused and 1 if it is the last element of this mapping.

Figure 20: Simplified and more abstract multimap representation featuring a primary and secondary table.

For the sake of simplicity all following figures will use the representation used in Figure 20. It shows the same internal state as Figure 19 but makes use of special keywords "-" and "END". Where an unused field contains "-" and the end of a hash group is represented by "END".

21
The size of the primary table is much smaller than the secondary table because the secondary table stores the actual data. In contrast, the primary table just stores the hash mapping. A bigger primary table will lead to more available hash mappings and less collision. This improves performance considerably at the cost of a high memory requirement. Changing the size of the primary table at runtime requires a complete rehash of all stored data. It is, therefore, very performance intensive. On the other hand, the size of the secondary table can be increased at will because the current implementation will fill the secondary table consecutively with no regard to its actual size.

### 6.2 Multi-depth hash map

A new multimap data structure was implemented that allows for space-efficient storage of data points at the cost of slow memory access. Points next to each other on the grid will most likely not be next to each other in the multimap resulting in cache misses on every grid access. A smaller map size may allow the whole map or at least significant parts to be loaded into cache, increasing the performance. The problem is that such a performance boost can only work if it is known in advance what subset of points will be used. Loading multiple smaller maps will result in the same cache problem as loading one giant map. Therefore, it is crucial to load only a handful of small maps each time. In ExPDESG, some memory-intensive methods require only grid points of a certain depth to be loaded. This means that creating a hash map for each depth allows only to have one small map. By the nature of adaptive sparse grids, many depths will have a relatively small number of points allowing the whole map to be loaded into cache. Figure 22 shows a read operation comparison between to original and the new multi-depth hash map implementations. There both hashmaps contains the same datapoints. The new alternative hash map implementation reduces the number of collisions and allows efficient access for algorithms that have iterate over all datapoints of a single depth.

A search for $0101_2$ with example hash 3 and depth D2 would have two collisions in the original mapping and none in the new per depth mapping.

![Figure 21: Example insertion of two more data packets into the multimap.](image)

![Figure 22: Comparison of the original hash grid structure and the new added Multi-depth hash map.](image)
6.3 Restriction

The restriction operator reduces the number of points of a grid. It gets a fine grid as input and returns a coarser grid. All coarse points will be interpolated with their adjacent points. The points that are used and their weight are determined by the stencil. The points with the highest depth are then removed.

![Diagram of restriction]

Restriction works by iterating through every point smaller than a given depth D. For each point, the next left and right neighbors of depth D are found and then used to update the points value with the stencil $[0.5, 1, 0.5]$ resulting in this formula

$$\text{value} = \text{oldValue} + 0.5 \cdot (\text{leftNeighborValue} + \text{rightNeighborValue}).$$

```java
void restriction1D(Vector coarse, Vector fine, int t, int d) {
    grids = getDepthOrderedSubgrids()
    for(grid : grids){
        depth = grid.getDepth();
        if(depth[d] > t) continue;
        for(point : grid){
            nLeft = grid.find(point.nextLeft(t+1))
            nRight = grid.find(point.nextRight(t+1))
            coarse[point] = fine[point] + 0.5 * ( fine[nLeft] + fine[nRight] )
        }
    }
}
```

Figure 24: Simplified pseudocode of a basic restriction1D implementation.

6.3.1 Restriction with multi depth hash

The left and the right neighbors are always points of depth D. This method can therefore be optimized by accessing the multi-depth hash map and loading the hashmap for depth D.
void restriction1D_hash(Vector coarse, Vector fine, int t, int d) {
    grids = getDepthOrderedSubgrids()
    for(grid : grids){
        depth = grid.getDepth();
        if(depth[d] > t) continue;
        depthGrid = getDepthGrid(point.nextLeft(t+1).getDepth());
        for(point : grid){
            nLeft = depthGrid.find(point.nextLeft(t+1))
            nRight = depthGrid.find(point.nextRight(t+1))
            coarse[point] = fine[point] + 0.5 * (fine[nLeft] + fine[nRight])
        }
    }
}

Figure 25: Simplified pseudocode using the multi depth hash. Now it is possible to make all memory location accesses on the multi-depth hash grid instead of the complete grid. This improves cache efficiency and therefore memory access times.

6.3.2 Push instead of pull

In the original version, the program iterates over all coarse points. Every one of these coarse points then pulls the values from the fine points in order to update itself. A more efficient solution would be to instead iterate over all fine grid points and let each one push its value to the corresponding coarse grid points. This reduces the number of grid points that have to be iterated over considerably. The values are now no longer pulled by the coarse grid points that need them but are pushed from the fine points to the coarse ones.

A pull restriction on a regular 3-dimensional sparse grid of depth 10 has to iterate over 20481 + 18434 + 16388 + 14344 + 12304 + 10272 = 92223 points (see Table 25). The push version only over 8256 points.

void restriction1D_inverted(Vector coarse, Vector fine, int t, int d) {
    grids = getDepthOrderedSubgrids()
    for(grid : grids){
        depth = grid.getDepth();
        if(depth[d] != t+1) continue;
        for(point : grid){
            nLeft = grid.find(point.nextLeft)
            nRight = grid.find(point.nextRight)
            value = fine[point]
            coarse[nLeft] += 0.5 * value;
            coarse[nRight] += 0.5 * value;
        }
    }
}

Figure 26: Simplified pseudocode of the push version of the code. Instead of using the neighbors to calculate the new value of the current point, it now uses the current point to calculate the values of its neighbors. Note that this method assumes that the coarse grid is initialized to zero. As otherwise, the += operation on uninitialized memory can lead to undefined behavior.

6.3.3 Push with inplace

The restriction can further be optimized by making it in place. Instead of restricting from a coarse grid into a fine grid, the restriction can happen in the same grid. In ExPDSG, sparse vectors always are the same size as the grid. A coarse vector will, therefore, actually reserves the same amount of space as a fine vector. Performing the restriction inplace prevents unnecessary time loss caused by reserving memory for a new vector and initializing it.
```python
void restriction1D_inplace(Vector vec, int t, int d) {
    grids = getDepthOrderedSubgrids()
    for (grid : grids){
        depth = grid.getDepth();
        if (depth[d] != t+1) continue;
        for (point : grid){
            nLeft = grid.find(point.nextLeft)
            nRight = grid.find(point.nextRight)
            value = vec[point]
            vec[nLeft] += 0.5 * value;
            vec[nRight] += 0.5 * value;
        }
    }
}
```

Figure 27: Simplified pseudocode of the inplace push version of the code. The undefined behavior problem of the inverted version can be circumvented by just using one vector. This is possible because the addition will never overwrite the values of points that can appear on the right site of the `+=` operator.

This optimization makes sense if the calling function does not need the old fine vector any more. In that case a lot of unnecessary instantiation time can be saved.

### 6.3.4 Performance

<table>
<thead>
<tr>
<th>Level 13: 114,687 Points</th>
<th>Name</th>
<th>Runtime (in ns)</th>
<th>Runtime (relative)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Restrict depth 2</td>
<td>Original</td>
<td>1,885,954</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td>Hash</td>
<td>1,825,784</td>
<td>0.96</td>
</tr>
<tr>
<td></td>
<td>Inverted</td>
<td>620,028</td>
<td>0.32</td>
</tr>
<tr>
<td></td>
<td>Inplace</td>
<td>648,434</td>
<td>0.34</td>
</tr>
<tr>
<td>Restrict depth 10</td>
<td>Original</td>
<td>4,729,773</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td>Hash</td>
<td>4,430,363</td>
<td>0.94</td>
</tr>
<tr>
<td></td>
<td>Inverted</td>
<td>25,901</td>
<td>0.005</td>
</tr>
<tr>
<td></td>
<td>Inplace</td>
<td>26,743</td>
<td>0.006</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Level 15: 647,167 Points</th>
<th>Name</th>
<th>Runtime (in ns)</th>
<th>Runtime (relative)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Restrict depth 2</td>
<td>Original</td>
<td>35,519,065</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td>Hash</td>
<td>12,596,360</td>
<td>0.35</td>
</tr>
<tr>
<td></td>
<td>Inverted</td>
<td>14,409,207</td>
<td>0.41</td>
</tr>
<tr>
<td></td>
<td>Inplace</td>
<td>15,288,504</td>
<td>0.43</td>
</tr>
<tr>
<td>Restrict depth 12</td>
<td>Original</td>
<td>142,126,350</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td>Hash</td>
<td>26,600,665</td>
<td>0.19</td>
</tr>
<tr>
<td></td>
<td>Inverted</td>
<td>214,797</td>
<td>0.0015</td>
</tr>
<tr>
<td></td>
<td>Inplace</td>
<td>225,402</td>
<td>0.0016</td>
</tr>
</tbody>
</table>

Table 7: [Ref. system: Work (Table 1)] Runtimes in ns of the different `restriction1D(t,d=0)` implementations on a 3-dimensional grid of given level.

Table 7 shows the runtime differences for the lowest and highest possible input depth. One can see that the inverted and inplace versions generally reduce the runtime considerably. These optimizations can reach a performance increase of nearly 700 times for high depths. For low depths, however, the achieved speedup reduces to 3 times for the level 13 grid and even falls short of the performance of the hash optimization for the level 15 grid.

This leads to the question of when the hash and when the inverted/inplace optimizations should be used and how their runtime behavior look like. In figure 28 one can see that the original functions runtime scales with the input parameter t. The hash version of the method has the same scaling behavior as the original but seems to reach its equilibrium at around t = 6. After that, the runtime does not increase further.
be because, for a small number of points, the added runtime induced by the extra hash map requests, that are needed for the new depth hash maps, offsets the improvements in access speed these hash maps provide. The new inverted and inplace optimizations now scale inversely with t. Meaning they actually reduce their runtime for higher values of t. This scaling behavior gets even more apparent when looking at the runtime for 9-dimensional caclulations shown in figure 29. There inverse scaling of the inverted and inplace methods both lead to dramatically improved runtimes for every depth except for very small t.

![Graph showing runtime vs depth for different methods](image1)

**Figure 28:** [Ref. system: Work (Table 1)][Raw data: Table 27] Runtimes in ns of the different restriction1D(t,d=0) implementations on a 3-dimensional level 15 grid.

![Graph showing runtime vs depth for different methods](image2)

**Figure 29:** [Ref. system: Work (Table 1)][Raw data: Table 28] Runtimes in ns of the different restriction1D(t,d=0) implementations on a 9-dimensional level 15 grid.

### 6.3.5 Further possible optimizations

The optimized method restriction1D() is a 1-dimensional restriction. This method has to be called multiple times for restricting in multiple dimensions. In the current program version, the calling method iterates through every dimension d and calls this method to restrict in the given d. Further performance improvements could therefore be achieved by restricting in all dimensions at the same time, removing the need to iterate through every point of depth D for every dimension. Theoretically, this could allow boundary and neighbor checks to be done in parallel using Single Instruction Multiple Data (SIMD). However, this optimization gets complicated, considering that the current calling method does not always iterate over
every single dimension but skips a few depending on its input parameters. Therefore, the SIMD implementation would have to accommodate these cases while also allowing an arbitrary amount of dimensions to be loaded in parallel. Currently, the caller uses the results of each 1-dimensional restriction. This optimization, therefore, would require extensive rework of the basic index access methods and the callers’ calculation loop.

6.4 Prolongation

![Diagram](image)

Figure 30: Example of a 1-dimensional prolongation to depth 3.

Prolongation works by iterating through every point of depth D. For each point, the next left and right neighbors of depth smaller D are found and then used to update the points value with the stencil $[0.5, 1, 0.5]$ resulting in this formula

$$\text{value} = \text{oldValue} + 0.5 \cdot (\text{leftNeighborValue} + \text{rightNeighborValue}).$$

```c
void prolongation(Vector coarse, Vector fine, Depth dCoarse, Depth dFine) {
    fine = coarse;
    int tcoarse;
    int tfine;
    for (int d = 0; d < DimensionSparseGrid; d++) {
        tcoarse = dCoarse.at(d) + 1;
        tfine = dFine.at(d);
        for (int t = tcoarse; t <= tfine; ++t) {
            prolongation1D(coarse, fine, t, d);
        }
    }
}
```

Figure 31: Simplified pseudocode of a basic prolongation() implementation. This method accepts an coarse and fine target depth and will call prolongation1D() repeatedly until it has prolongated to the targeted fine depth.
void prolongation1D(Vector coarse, Vector fine, int t, int d) {
    grids = getDepthOrderedSubgrids();
    for (grid : grids) {
        depth = grid.getDepth();
        if (depth[d] != t) continue;
        for (point : grid) {
            nLeft = grid.find(point.nextLeft);
            nRight = grid.find(point.nextRight);
            coarse[point] = fine[point] + 0.5 * (fine[nLeft] + fine[nRight]);
        }
    }
}

Figure 32: Simplified pseudocode of a basic prolongation1D() implementation.

6.4.1 Prolongation Inplace

Prolongation takes as parameters a fine and a coarse vector. The coarse vector will be used to get all the values for the coarse points, which will then be used to calculate the new values of the fine vector. The performance problem lies in the fact that both the coarse and the fine vectors are actually the same size. Meaning the coarse vector also contains all fine points; they are simply not used. The fine vector then has to be filled with all values from the coarse vector plus the new fine values. In most cases, the calling function does not actually need to have a new separate vector as a result. In that case, the new fine values can just be written into the coarse vector. This prevents the necessity of having to initialize a fine vector by copying all coarse values from the coarse vector.

void prolongation_inplace(Vector vec, Depth dCoarse, Depth dFine) {
    int tcoarse;
    int tfine;
    for (int d = 0; d < DimensionSparseGrid; d++) {
        tcoarse = dCoarse.at(d) + 1;
        tfine = dFine.at(d);
        for (int t = tcoarse; t <= tfine; ++t) {
            prolongation1D(vec, t, d);
        }
    }
}

Figure 33: Simplified pseudocode of an inplace version of the prolongation() implementation. In contrast to the original prolongation() method this one does not require a copy of the coarse grid to be made (see Figure 31, line 2)
```cpp
void prolongation1D_inplace(Vector vec, int t, int d) {
    grids = getDepthOrderedSubgrids()
    for (grid : grids){
        depth = grid.getDepth();
        if (depth[d] != t) continue;
        for (point : grid){
            nLeft = grid.find(point.nextLeft)
            nRight = grid.find(point.nextRight)
            vec[point] = vec[point] + 0.5 * ( vec[nLeft] + vec[nRight] )
        }
    }
}
```

Figure 34: Simplified pseudocode of the inplace prolongation1D implementation. This version only needs a single input Vector and therefore allows its caller prolongation() (Figure 33) to no longer need to copy one of its input vectors.

6.4.2 Prolongation using the multi depth hash

The multi-depth hash version of the prolongation code does not need to find data points of a single depth but of multiple. Therefore the usage of the multi-depth hash is more complicated than in the inverted prolongation case. Before doing any calculation, it is necessary to find all depth hashes for every single depth that we will need to find data points in. For the prolongation in direction d, it is enough to create an array containing every depth that has a smaller depth in the given direction d than the current depth has. This results in only having to load a handful of depth hashes in advance. In contrast to the original prolongation inplace optimization, this one scales very well with a high gridpoint count at the cost of being a bit slower for grids of smaller sizes. This happens because for small grids the cost of finding the corresponding depth hashes and searching the points there takes longer than just searching them in the normal grid.

```cpp
void prolongation1D_hash(Vector vec, int t, int d) {
    grids = getDepthOrderedSubgrids()
    for (grid : grids){
        depth = grid.getDepth();
        if (depth[d] != t) continue;
        depthGrids = getGridsOfDepthInDirection(depth,d,t);
        for (point : grid){
            nLeft = depthGrids.find(point.nextLeft)
            nRight = depthGrids.find(point.nextRight)
            vec[point] = vec[point] + 0.5 * ( vec[nLeft] + vec[nRight] )
        }
    }
}
```

Figure 35: Simplified pseudocode of a inverted prolongation1D implementation. This version uses the multi depth hash to find the memory mapping of the left and right neighbor points. Allowing for faster and more cache efficient memory access of the smaller depth grids.

6.4.3 Inverted prolongation using the multi depth hash

The prolongation method iterates over all points of depth D. For each point, it finds the corresponding neighbors of depth smaller than D. In contrast to restriction, the prolongation method does not iterate over all points of depth smaller than D but over all points of depth D. In short: restriction pushes its values, and prolongation pulls (see images Figure 23 and Figure 30). Because of internal memory organization, the value of points that need to be searched is very costly to access. The idea is to rewrite the prolongation method to also push values by iterating over all points of depth smaller D and pushing to all points of depth D. The search for points of depth D can then further be speed up by using the multi depth grid for the single depth D.
```cpp
void prolongation1D_inverted(Vector vec, int t, int d) {
    grids = getGridsOfSmallerDepth();
    for (grid : grids) {
        depth = grid.getDepth();
        if (depth[d] >= t) continue;
        depthGrid = getDepthGrid(depth);
        for (point : grid) {
            nLeft = depthGrid.find(point.nextLeft);
            nRight = depthGrid.find(point.nextRight);
            vec[nLeft] += 0.5 * vec[point];
            vec[nRight] += 0.5 * vec[point];
        }
    }
}
```

Figure 36: Simplified pseudocode of an inverted prolongation1D implementation using the multi-depth hash map. The depth if-condition now checks for depth smaller than \( t \). That means that instead of iterating over all points of depth \( t \) we now iterate over all points with a depth smaller than \( t \) and push the found values into the points of depth \( t \). It is then possible to use the multi-depth hash map to speed up the memory access for the required points of depth \( t \).

### 6.4.4 Performance

<table>
<thead>
<tr>
<th>Level 13: 114,687 Points</th>
<th>Name: Runtime (in ns): Runtime (relative):</th>
</tr>
</thead>
<tbody>
<tr>
<td>Worst Case: Depth ((1,1,1) \to (10,2,1))</td>
<td>Original: 7,838,904 1</td>
</tr>
<tr>
<td></td>
<td>Inplace: 4,982,184 0.64</td>
</tr>
<tr>
<td></td>
<td>Hash: 5,589,350 0.71</td>
</tr>
<tr>
<td></td>
<td>Inverted: 6,747,645 0.86</td>
</tr>
</tbody>
</table>

| Best Case: Depth \((1,1,1) \to (2,1,1)\) | Original: 3,704,806 1 |
|                               | Inplace: 705,137 0.19 |
|                               | Hash: 766,122 0.21 |
|                               | Inverted: 637,086 0.17 |

<table>
<thead>
<tr>
<th>Level 15: 647,167 Points</th>
<th>Name: Runtime (in ns): Runtime (relative):</th>
</tr>
</thead>
<tbody>
<tr>
<td>Worst Case: Depth ((1,1,1) \to (12,2,1))</td>
<td>Original: 170,222,311 1</td>
</tr>
<tr>
<td></td>
<td>Inplace: 104,206,862 0.61</td>
</tr>
<tr>
<td></td>
<td>Hash: 43,655,222 0.26</td>
</tr>
<tr>
<td></td>
<td>Inverted: 68,788,431 0.40</td>
</tr>
</tbody>
</table>

| Best Case: Depth \((1,1,1) \to (2,1,1)\) | Original: 81,039,827 1 |
|                                 | Inplace: 17,837,132 0.22 |
|                                 | Hash: 4,703,301 0.06 |
|                                 | Inverted: 5,235,581 0.06 |

Table 8: [Ref. system: Work (Table 1)] Benchmarks for regular sparse grids of given level in 3 dimensions. These runtimes are for the prolongation methods (prolongation() Figure 31 and prolongation_inplace() Figure 33) that call the underlying prolongation1D() methods.

In table 8 it is easy to see that the performance gain of all these different optimizations is substantial. The standard inplace version has a slight performance advantage when using a smaller amount of gridpoints. For level 13 all three optimizations feature similar performance with the inplace version winning by a small margin for the worst case. On level 15 the inverted and hash optimizations far outperform the original and inplace versions. This leads to the inplace version actually being the fastest one for smaller calculations. For calculations involving a high number of points the inverted and hash versions both scale far better than the original or inplace versions. In order to find the best-performing version for a given calculation, these performance benchmarks have to be redone for the calculations expected grid size and dimension. Figure 37 shows the runtime behavior for these optimizations on a 3 dimensional grid. There the runtimes of...
all the methods are quite similar, with the normal inplace version generally performing the best. The original versions calling method `prolongation()` needs to additionally create a new output vector. This extra runtime cost is not taken into account in these measurements. Interestingly in 9 dimensions, the runtime behavior of all these implementations changes completely, as seen in Figure 38. In contrast to the 3-dimensional version (see Figure 37), the normal inplace optimization no longer performs the best. Here the improved cache efficiency of the multi-depth hash grid used in the hash and inverted implementations shows its true potential. Both optimizations are substantially faster than the original or inplace one.

![Figure 37](image1.png)

**Figure 37:** [Ref. system: Work (Table 1)][Raw data: Table 29] Runtimes in ns of the different `prolongation1D(t,d=0)` implementations on a 3-dimensional level 13 grid.

![Figure 38](image2.png)

**Figure 38:** [Ref. system: Work (Table 1)][Raw data: Table 30] Runtimes in ns of the different `prolongation1D(t,d=0)` implementations on a 9-dimensional level 15 grid.

### 6.5 `MultiLevelVector::setMultiLevelValues()`

This method is used to set the values of an `MultiLevelVector`. There only the values of gridpoints are loaded that have a depth smaller or equal to the given depth. The original version of this method iterates over all points of the grid and checks the depth for each and every one of them. Using the `DepthMap` it is possible to reduce the number of points that have to be iterated. When loading a `DepthMap` for a certain `Depth d` it is no longer necessary to check if a point contained in this map has the correct depth. Also points that don't
have the correct depth will never be iterated over. Reducing the runtime considerably when only checking a small fraction of all points.

<table>
<thead>
<tr>
<th>Level 13: 114,687 Points</th>
<th>Name: Runtime (in ns): Runtime (relative):</th>
</tr>
</thead>
<tbody>
<tr>
<td>Worst Case: 1534 Points</td>
<td>Original: 517,557 15.94</td>
</tr>
<tr>
<td></td>
<td>Optimized: 32464 1</td>
</tr>
<tr>
<td>Best Case: 1 Point</td>
<td>Original: 429,688 4314.13</td>
</tr>
<tr>
<td></td>
<td>Optimized: 99.6 1</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Level 15: 647,167 Points</th>
<th>Name: Runtime (in ns): Runtime (relative):</th>
</tr>
</thead>
<tbody>
<tr>
<td>Worst Case: 5632 Points</td>
<td>Original: 9,891,468 1.7</td>
</tr>
<tr>
<td></td>
<td>Optimized: 5,809,663 1</td>
</tr>
<tr>
<td>Best Case: 1 Point</td>
<td>Original: 2,463,580 24,635.80</td>
</tr>
<tr>
<td></td>
<td>Optimized: 100 1</td>
</tr>
</tbody>
</table>

Table 9: [Ref. system: Work (Table 1)] Benchmarks for regular sparse grids of given level in 3 dimensions.

Table 9 shows the runtimes of the original and optimized versions. The best case is here defined as having to just find and copy one single gridpoint to the MultiLevelVector. This will happen for the center point at Depth(1,1,1). The worst case is having to copy the maximum number of grid points. For level 13 this would be Depth(10, 2, 1) and for level 15 Depth(4, 2, 9). These depths have been determined by checking for all possible depth inputs. The method setMultiLevelValues() will only copy the data points of a single given depth. So the maximum number of points is equal to the maximum number of points found in a single depth of a regular non-adaptive sparse grid. Here the cache efficiency of the now-used depth map shows its potential. With the main reason for the speed up actually being the now limited number of points that have to be iterated.

### 6.6 Reordering

In this hashmap implementation a lot of performance is lost through cache misses. By changing the position of the data points it should be possible to decrease the number of cache misses and therefore increase performance. There are multiple reordering strategies.

![Secondary Table mapping](image)

Figure 39: A sample input of the hashmap. Here the sample input order was chosen in the same order the expdesg program would have created these points.

#### 6.6.1 Reorder by depth

Sorting by the depth of the inserted points essentially tries to recreate the data structure of the multi-depth hash optimization. Data points that have the same depth are then stored near each other. This increases the number of cache hits for tasks that need to access a lot of data points from the same depth, therefore, improving performance. The problem is that, unlike in the multi-depth hash, the number of collisions stays the same. In order to find a gridpoint, there will still be many collisions which will mainly result in cache misses because other points with the same hash can have any arbitrary depth and are, therefore, not loaded into cache.
In figure 40 one can see the hashmap example from figure 39 reordered by depth. Datapoints of the same depth are now located next to each other improving cache hit rate for multiple following searches of data of the same depth. Note that in the case of a hash collision, this structure may require table accesses at the wrong depth leading to cache misses. Here searching for 1000_{3} of depth 3 will result in having to access 0010_{2} of depth 1 and 0101_{2} of depth 2.

Having to access entries of the wrong depth can be avoided by knowing at what index the secondary table contains entries of the correct depth. While this further optimization could lead to better performance, it also comes at the cost of having to sort the secondary table every time a point gets added. When changing the ordering of the entries in the secondary table all already existing vectors that use this grid have to be reordered too. The adaptive property and the resulting possible point additions during runtime make this a hard-to-justify trade-off.

6.6.2 Reorder by collision

To make the map more collision friendly, it is also possible to reorder by hash mapping. Datapoints that received the same hash for the primary table will be stored next to each other in the secondary table. This leads to collisions always being loaded into cache. This allows us to access data points with multiple collisions more easily. Figure 41 shows the example hashmap from Figure 39 reordered by collision. Now all data points that share the same hash in the primary table are grouped into a block of consecutive memory in the secondary table. This allows most members of such a hash group to be loaded into cache together, reducing the number of cache misses.

**Figure 40**: Reordered figure 39 by depth. See Figure 2 for the depth values of the different input binary numbers.

**Figure 41**: Reordered Figure 39 by collision mapping. All colliding datapoints are now located next to eachother increasing cache hit rate.
6.6.3 Performance

<table>
<thead>
<tr>
<th>Reordering</th>
<th>Runtime (in s)</th>
<th>Runtime (relative)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Original:</td>
<td>285.64</td>
<td>1</td>
</tr>
<tr>
<td>Depth:</td>
<td>181.34</td>
<td>0.63</td>
</tr>
<tr>
<td>Collision:</td>
<td>295.68</td>
<td>1.04</td>
</tr>
</tbody>
</table>

DIM 3, Level 15: 647, 167 Points

<table>
<thead>
<tr>
<th>Reordering</th>
<th>Runtime (in s)</th>
<th>Runtime (relative)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Original:</td>
<td>134.75</td>
<td>1</td>
</tr>
<tr>
<td>Depth:</td>
<td>138.29</td>
<td>1.03</td>
</tr>
<tr>
<td>Collision:</td>
<td>131.59</td>
<td>0.98</td>
</tr>
</tbody>
</table>

DIM 9, Level 12: 1,519 Points

Table 10: [Ref. system: Work (Table 1)] Runtime measurements for a simple example program doing one matrix multiplication on a non-adaptive grid of a given size.

As seen in table 10 at 3 dimensions and approximately 650,000 grid points, the depth reorder significantly improves the program’s performance. While the collision reorder does not significantly change the program runtime. At 9 dimensions and approximately 1,500 grid points, both reordering strategies do not significantly change the program runtime. A possible reason could be the small number of data points allowing the standard hashmap implementation to already fit into the cache.

6.6.4 Problems

The sparse vectors save the values of each gridpoint at the same array index the grid saves its gridpoint in the secondary table. Reordering the table at runtime, therefore, comes at the significant performance cost of having to reorder all sparse vectors the same way. This could also easily lead to bugs in the program resulting from not reordered sparse vectors. Therefore the current implementation only supports the reordering step to happen before the actual calculation begins. All at a later time, adaptively added points will be appended to the sparse grid in the same order they were created. Not making use of any reordering performance improvements. While Table 10 shows that a correct reordering technique has the potential to improve the program’s runtime significantly, this test was only done on a non-adaptive grid. The usage of an adaptive grid would require reordering the grid every time points get added, which would most likely diminish most performance gains. For higher-dimensional calculations, the performance gains for both tested reordering strategies are negligible or non-existent. This shows that the best reordering strategy most likely varies on a case-to-case basis.

7 MultiDimCompass

The MultiDimCompass class is used to iterate over the different directions of a multidimensional stencil. This simplifies the task of finding the next correct gridpoint for a stencil calculation. The original version of the code had to calculate the new direction every time the getDirection() method was called. This design features great performance if calls to getDirection() are few and far between as it prevents the MultiDimCompass from calculating directions that are not required. The problem is that the actual use case of the MultiDimCompass is to iterate over every possible direction for every dimension. This means that most of the time, all directions are accessed, contradicting the original design approach. The MultiDimCompass can be optimized for this kind of usage by calculating all possible directions at the incrementation step (operator++()). This optimization further uses the fact that a single incrementation of the MultiDimCompass will, most of the time only change the value of one direction. Making the runtime of the new operator++() equal to the old getDirection(), with the runtime of the new getDirection() just being a single array access and, therefore, negligible.
class MultiDimCompass {
    public:
        MultiDimCompass() {
            shiftNumber = 0;
        }

        void operator++() {
            ++shiftNumber;
        }

        // calculate on call to getDirection()
        inline Direction getDirection (unsigned int d) const {
            unsigned int num = shiftNumber;
            for (int s = 0; s < d; ++s) num = num / 3;
            return (Direction) (num % 3);
        }

        bool hasNext() { return shiftNumber < maxShift; }

    private:
        unsigned int shiftNumber;
        static unsigned int maxShift;
};

Figure 42: Simplified C++ implementation of the original MultiDimCompass. This original implementation calculates the new directions on the call to getDirection().
class MultiDimCompass {
    public:
        MultiDimCompass() {
            shiftNumber = 0;
            for (size_t i = 0; i < DimensionSparseGrid; i++)
            {
                directions[i]=0;
            }
        }

        // calculate on call to ++()
        void operator++() {
            ++shiftNumber;
            for (size_t i = 0; i < DimensionSparseGrid; i++)
            {
                directions[i]=directions[i]+1;
                if (directions[i]==3)
                {
                    directions[i] = 0;
                    continue;
                }
                break;
            }
        }

        inline Direction getDirection (unsigned int d) const {
            return (Direction) directions[d];
        }

        bool hasNext() { return shiftNumber < maxShift; }
    }

private:
    unsigned int directions[DimensionSparseGrid];
    unsigned int shiftNumber;
    static unsigned int maxShift;
};

Figure 43: Simplified C++ implementation of the optimized MultiDimCompass. The new implementation calculates the new directions on the incrementation call operator++(). The actual directions only have to be calculated once this way. Moreover, the new way of calculating the directions does not use the expensive division operation on repeat.

<table>
<thead>
<tr>
<th>DIM 3, Level 15: 647,167 Points</th>
<th>Reordering:</th>
<th>Runtime (in s):</th>
<th>Runtime (relative):</th>
</tr>
</thead>
<tbody>
<tr>
<td>Original: 208.77</td>
<td>1.06</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Optimized: 196.21</td>
<td>1</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>DIM 9, Level 12: 1,519 Points</th>
<th>Reordering:</th>
<th>Runtime (in s):</th>
<th>Runtime (relative):</th>
</tr>
</thead>
<tbody>
<tr>
<td>Original: 190.04</td>
<td>1.36</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Collision: 139.96</td>
<td>1</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Table 11: [Ref. system: Work (Table 1)] Runtime of an single multiplication using sparse vectors of given size and dimensionality.

This optimization improves performance by preventing unnecessary calculations of directions for each dimension. In table 11 it is easy to see that this only shows its true potential if the dimensionality is relatively high. The 3-dimensional calculation barely improved in performance. While for 9 dimensions, the whole calculations runtime improved by roughly 30% just by using a more efficient way to calculate the next direction.

8 Combined performance improvements

The in sections 5, 6 and 7 presented optimizations decrease the runtime of the program when executed on a single core. Before we dive into the different parallelization techniques described in section 9 it is interesting
to see how much all previous optimizations combined can decrease the program runtime.

<table>
<thead>
<tr>
<th>Level 13: 114,687 Points</th>
<th>Name:</th>
<th>Runtime (in s):</th>
<th>Runtime (relative):</th>
</tr>
</thead>
<tbody>
<tr>
<td>Original:</td>
<td>42.7</td>
<td>4.5</td>
<td></td>
</tr>
<tr>
<td>Optimized:</td>
<td>9.48</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>Level 15: 647,167 Points</td>
<td>Original:</td>
<td>863</td>
<td>4.69</td>
</tr>
<tr>
<td>Optimized:</td>
<td>184</td>
<td>1</td>
<td></td>
</tr>
</tbody>
</table>

Table 12: [Ref. system: Work (Table 1)] Final single core speedup for a single multiplication using sparse vectors of given size in 3 dimensions.

For some of the presented optimizations, different implementations featured different performances depending on the dimensionality and size of the problem. An algorithm deciding the best implementation to use at runtime depending on the given inputs has yet to be created. Therefore the performance data was collected by choosing one possible implementation beforehand. Table 12 shows a roughly 4.5 times performance increase compared to the original implementation.

### 9 Parallelization

For most scientific calculations, it is not enough to only use a single core. Today nearly all home desktop computers have CPUs with multiple cores. For average consumers, these reach from 4 cores all the way up to 16 cores. Moreover, for workstations and high-end computing, this number gets even higher. Even running the program in parallel on the average home computer could lead to substantial performance gains. Problems of high enough complexity can only be calculated in parallel in a feasible amount of time. In order to use supercomputers effectively, it is required to parallelize a program so that it is able to run on multiple cores or even nodes while still achieving a decent speedup.

**Parallel speedup:** The parallel speedup is a measure of how much the parallelized programs runtime has improved in reference to the original single core runtime.

\[ S_n = \frac{T_1}{T_n}, \text{ with } T_n \text{ being the runtime of the program using } n \text{ worker.} \]

**Parallel efficiency:** The parallel efficiency shows how efficiently the parallelization is in terms of its scaling. A program that halves its runtime when doubling the amount of parallel workers has an efficiency of one.

\[ \varepsilon_n = \frac{S_n}{n}, \text{ with } n \text{ being the number of workers.} \]

**Amdahl’s Law:** In order to efficiently parallelize the code, it is required to have calculations that are independent of each other and can be computed in parallel. Even small serial parts of the code will limit the theoretical speedup significantly. This is described in Amdahl’s law

\[ S_n(p) = \frac{1}{(1 - p) + \frac{p}{n}}. \]

Here \( S \) denotes the maximal theoretical speedup of the program, \( p \) is the percent of the program that can be run in parallel, and \( n \) is the number of parallel threads. The maximum achievable parallel speedup using an unlimited number of cores \( S_{\infty} \) is therefore

\[ S_{\infty}(p) = \lim_{x \to \infty} (S) = \frac{1}{1 - p}. \]

Using this formula, it is possible to calculate the maximum speedup of a given program. A perfect parallel program with 100% parallelization will be able to infinitely scale on parallel systems \( \lim_{p \to \infty} S_{\infty}(p) = \infty \). However a program with 80% parallelization will only be able to reach a maximum speedup of \( S_{\infty}(0.8) = 5 \) using an infinite number of processors.
9.1 Reference calculation

For the reference calculation of the parallelization, an adaptive Poisson-Neumann solver is used. The dimensionality is set to 6 dimensions. The reference program solves the Poisson equation with Neumann boundary conditions on an adaptive grid. After converging to a small enough error, it will adaptively refine the grid and restart the iterative solver. Every such refinement step increases the Adaptive Refinement-Level (AR-Level). Therefore, the AR-Level denotes the number of adaptive refinements to the grid and is only required for repeatability given the project’s source code. Table 13 shows the number of Degrees of Freedom (DoF) of the reference calculations and the number of iterations that were done. Here "Iterations" means the number of times the iterative solver had to execute the underlying multiplication to converge. In order to get the runtime for a single iteration, the runtimes given in the subsequent tables have to be divided by this amount.

<table>
<thead>
<tr>
<th>AR-Level</th>
<th>Iterations</th>
<th>Active Nodes/DoF</th>
<th>Hanging Nodes</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>2</td>
<td>729</td>
<td>0</td>
</tr>
<tr>
<td>2</td>
<td>5</td>
<td>3645</td>
<td>0</td>
</tr>
</tbody>
</table>

Table 13: Specifications of the here used 6-dimensional Poisson Neumann solver for a given AR-Level.

9.2 Distributive computing using MPI

Message Passing Interface (MPI) is a messaging standard designed for highly parallel multi-node architectures. It allows for fast and reliable communication between programs running on CPUs that are located on different computers. In the context of high-performance computing, these different computers are called nodes. A supercomputer can consist of thousands such nodes. Usual parallelization techniques using multi-threading often only work on a single CPU or do not consider the massive latency penalty when accessing data stored on another node in the system. With MPI, the programmer can safely manage all resources and data transfers between Nodes. Instead of launching a host program that creates multiple threads, a program using MPI has to be started once for every CPU core that should be used. These multiple running programs all get their unique MPI-id called MPI-rank and can communicate with each other.

The adaptive sparse grid algorithm is capable of calculating in an arbitrary number of dimensions. Depending on the dimensionality $d$ the algorithm calculates $2^d$ cases [1]. All these cases can be calculated independently and take a lot of time to compute. They are, therefore, a prime target for parallelization. Given a problem in high enough dimensions, the program can theoretically be parallelized on an arbitrary number of cores. In reality, even a 6-dimensional problem only allows for $2^6 = 64$ cases and, therefore, MPI tasks. In today’s day and age, there already exist single CPUs that have 64 or more cores making the multinode computing ability of MPI somewhat redundant. A possible solution to this problem will later be discussed in section 9.5.

The dimensionality $d$ of the program is known at compile-time and allows the program to be efficiently distributed. Each MPI process will calculate its assigned number of cases before combining them and sending them to the other MPI tasks.
void calcCases() {
    unsigned numCases = 1 << DIM;
    SparseVector Ax = 0.0;
    for(size_t i = 0; i < numCases; i++){
        Ax_neu = calcCase(i);
        Ax += Ax_neu;
    }
}

void calcCasesMPI(int world_rank, int world_rank) {
    unsigned numCases = 1 << DIM;
    SparseVector Ax = 0.0;
    for(size_t i = 0; i < numCases; i++){
        if (world_rank == (i % world_size)) {
            Ax_neu = calcCase(i);
            Ax += Ax_neu;
        }
    }
    MPI_Allreduce(Ax);
}

Figure 44: Simplified pseudo-code versions of the case function. Left the original and right the MPI parallelized one. The most notable differences in the MPI version (right) are the extra check in line 5 and the Allreduce call in line 10. Line 5 checks if this MPI task should calculate the current case. (This if-clause here exists for clarification. To save some unnecessary branches, it is more efficient to increment the index i by world_size.) Line 10 adds up all the different vectors Ax of every MPI task together.

The only remaining non-parallel part is now the addition of the result of all cases. Here every process adds all cases it calculates together and then uses MPI_Allreduce() to parallelize the addition between all processes.

<table>
<thead>
<tr>
<th>Nodes</th>
<th>MPI-Tasks</th>
<th>AR-Level 1 in sec</th>
<th>Speedup</th>
<th>Efficiency</th>
<th>AR-Level 2 in sec</th>
<th>Speedup</th>
<th>Efficiency</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>159.83</td>
<td>1</td>
<td>1</td>
<td>3366.38</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>20</td>
<td>22.99</td>
<td>6.95</td>
<td>0.35</td>
<td>441.88</td>
<td>7.62</td>
<td>0.38</td>
</tr>
<tr>
<td>4</td>
<td>40</td>
<td>16.93</td>
<td>9.44</td>
<td>0.24</td>
<td>311.08</td>
<td>10.82</td>
<td>0.27</td>
</tr>
<tr>
<td>3</td>
<td>60</td>
<td>14.03</td>
<td>11.39</td>
<td>0.19</td>
<td>256.60</td>
<td>13.12</td>
<td>0.22</td>
</tr>
<tr>
<td>4</td>
<td>80</td>
<td>12.44</td>
<td>12.85</td>
<td>0.16</td>
<td>218.60</td>
<td>15.40</td>
<td>0.19</td>
</tr>
<tr>
<td>5</td>
<td>100</td>
<td>12.42</td>
<td>12.86</td>
<td>0.13</td>
<td>218.28</td>
<td>15.42</td>
<td>0.15</td>
</tr>
</tbody>
</table>

Table 14: [Ref. system: Meggie (Table 1)][6D. Solver: see Table 13] Runtime measurements on the Meggie cluster using multiple nodes.

<table>
<thead>
<tr>
<th>MPI-Tasks</th>
<th>Runtime AR-Level 1 in sec</th>
<th>Speedup</th>
<th>Efficiency</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>171.15</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>107.35</td>
<td>1.59</td>
<td>0.80</td>
</tr>
<tr>
<td>4</td>
<td>66.92</td>
<td>2.56</td>
<td>0.64</td>
</tr>
<tr>
<td>8</td>
<td>41.44</td>
<td>4.13</td>
<td>0.52</td>
</tr>
<tr>
<td>16</td>
<td>25.84</td>
<td>6.62</td>
<td>0.41</td>
</tr>
<tr>
<td>32</td>
<td>16.27</td>
<td>10.52</td>
<td>0.33</td>
</tr>
<tr>
<td>64</td>
<td>10.13</td>
<td>16.90</td>
<td>0.26</td>
</tr>
</tbody>
</table>

Table 15: [Ref. system: TinyGPU (Table 1)][6D. Solver: see Table 13] Runtime measurement on the TinyGPU cluster using a single 128 core CPU.
Table 14 shows that this MPI parallelization allows for speedups around 13 times when using 64 MPI-tasks on a 6-dimensional problem. The speedup changes depending on the size of the grid and reaches 16 times for AR-Level 2. In Table 15 one can see that the speedup is even greater when calculated on a single CPU instead of multiple Nodes. Now the speedup is able to reach 17 times. The resulting speedup and efficiency curve can be seen in figure 45.

9.3 Multithreading using OMP

OpenMP (Open Multi-Processing) allows for easy-to-use multithreading. In contrast to MPI, there exists only one program that gets executed. The program then spawns multiple OMP threads that can execute code in parallel. This allows threads to work on the same data and share their memory. This, in turn, removes the need for expensive communication between different tasks. All this comes at the expense of only really being efficient when used on a single CPU. Also, if one is not careful, using OMP can lead to race conditions or false sharing.

**Race conditions** can happen when two or more threads try to modify or read from the same memory location. A simple C++ incrementation like `A++;` may look like a single instruction, but once compiled, it can actually represent a series of instructions. For simplicity’s sake consisting of a read of A, an incrementation, and a write back to A. If multiple threads execute this code in parallel, they might read A simultaneously. Both then increment the read value and write the result back to A, leading to a wrongly incremented A. This programming error then leads to unexpected wrong results or undefined program states.

**False sharing** happens when two or more processes change the data of the same cache line. A cache line is a line of data, usually 64 bytes, that gets loaded into the cache. If different CPU cores change the data in the same line, then this cache line has to be updated for both cores by cache coherency protocols. This can lead to unnecessary performance degradation. In contrast to race conditions, false sharing will not result in wrong or undefined program states but will silently reduce the program’s performance.
The stencil calculation uses a lot of runtime, and each calculation is independent of the last one. This allows for a quick OMP parallelization by adding a `#pragma omp for` (see Figure 46, right, line 4) above the for loop.

### Table 16: [Ref. system: Meggie (Table 1)][6D. Solver: see Table 13] OMP performance on the Meggie cluster. The Meggie cluster features 20 cores per Node.

<table>
<thead>
<tr>
<th>OMP-Threads</th>
<th>Runtime AR-Level 1 in sec</th>
<th>Speedup</th>
<th>Efficiency</th>
<th>AR-Level 2</th>
<th>Speedup</th>
<th>Efficiency</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>159.83</td>
<td>1</td>
<td>1</td>
<td>3366.38</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>20</td>
<td>20.585165</td>
<td>7.97</td>
<td>0.40</td>
<td>375.075193</td>
<td>0.90</td>
<td>0.45</td>
</tr>
</tbody>
</table>

### Table 17: [Ref. system: TinyGPU (Table 1)][6D. Solver: see Table 13]

<table>
<thead>
<tr>
<th>OMP-Threads</th>
<th>Runtime AR-Level 1 in sec</th>
<th>Speedup</th>
<th>Efficiency</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>171.15</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>87.38</td>
<td>1.96</td>
<td>0.98</td>
</tr>
<tr>
<td>4</td>
<td>46.98</td>
<td>3.64</td>
<td>0.91</td>
</tr>
<tr>
<td>8</td>
<td>26.63</td>
<td>6.43</td>
<td>0.80</td>
</tr>
<tr>
<td>16</td>
<td>16.61</td>
<td>10.30</td>
<td>0.64</td>
</tr>
<tr>
<td>32</td>
<td>11.53</td>
<td>14.84</td>
<td>0.46</td>
</tr>
<tr>
<td>64</td>
<td>8.88</td>
<td>19.27</td>
<td>0.30</td>
</tr>
</tbody>
</table>
Tables 16 and 17 both show that a significant speedup of the calculation using OMP is possible. The achieved speedup using the same amount of cores is even greater than with the MPI version. This is most likely the result of no longer necessary communications between different threads.

9.4 GPU compute/acceleration using CUDA

Compute Unified Device Architecture (CUDA) is a C++ extension by Nvidia that allows for easy use of GPUs in calculation-intensive programs. Modern GPUs are capable of doing thousands of calculations in parallel. Therefore they are often used to drastically speed up computations than can benefit from such high parallelization. In contrast to CPU parallelization, the problem with GPU parallelization is that the GPU cores are much simpler than CPU cores. In GPUs, not every core can perform a different independent calculation. When computing on the GPU with huge amounts of threads, the used threads are separated into thread blocks. On the hardware side, these blocks consist of warps. Every warp contains 32 threads, and all its threads can only run the same instruction. For programs that feature a massive amount of computations of the same calculation with different input values, this is not a problem. However, just simple changes in these computations, like adding an if-clause, can reduce the performance considerably. Now one part of the allocated threads will enter the if-clause while the other will not, resulting in different instructions on threads of the same warp. This problem is called thread divergence. The CUDA solution to this problem is to simply deactivate all threads in the warp that do not go into the if-block and continue execution. Then after the if-block is finished, all these threads will be deactivated, and only threads that enter the else-block will run. Afterward, normal operation continues. This workaround comes at a hefty performance cost. [6]

The program execution time would improve significantly by effectively a massively parallel execution. However, the program’s current state makes use of many branches and complex data structures that make a GPU implementation inefficient. This following CUDA implementation of a small part of the codebase has its purpose of exploring if CUDA can be feasible for this kind of task. It may serve as a proof of concept for following optimizations outside of this paper’s focus. A simple Poisson Neumann solver in 6 dimensions was used as a reference calculation for the optimizations. Profiling the runtime of different parts of the code showed that the stencil calculation uses over 90% of the program runtime. The stencil calculation happens every iteration for a huge amount of points. This makes it an interesting target for GPU acceleration. Unfortunately, the actual stencil calculation uses many if-clauses. This prevents the GPU from getting anywhere close to peak performance.
double calcStencil(Point p, Value u);

void applyStencilCPU(Grid &grid, Depth T, Vector &Ax, Vector &u) {
  for(point : grid){
    if(point.depth < T){
      Ax[point] = calcStencil(point,u[point]);
    }
  }
}

Vector calcStencilCUDA(vector<Point> ps, vector<double> us);

void applyStencilGPU(Grid &grid, Depth T, Vector &Ax, Vector &u) {
  vector<Point> points;
  vector<double> u_vals;
  for(point : grid){
    if(point.depth < T){
      points.push_back(point);
      for(dir : directions){
        Point np(point,dir);
        u_vals.push_back(u[np]);
      }
    }
  }
  Ax = calcStencilCUDA(points,u_vals);
}

Figure 48: Simplified pseudo-code versions of the applyStencil() function. On the left is the original for the CPU, and right is the new version for the GPU. The original version iterates through every point and calculates its stencil. The new version finds all the points of correct depth and their neighbor’s values and passes them to the calcStencilCUDA() function. This has to be done because the act of finding the value for a given point is implemented using a specialized hash map. Getting this hash map to run efficiently on the GPU is outside this paper’s scope.

In order to achieve better performance, some if-clauses and for-loops have been restructured or removed. Instead of checking for "Mass" or "Stiffness" stencil types, there are now dedicated functions for both. The IndexDimension, MultiDimCompass, and Depth classes have partially been reconstructed for usage with CUDA. The optimizations in the depth(), nextLeft() and nextRight() functions by the usage of dedicated CPU instructions have been replicated by using the CUDA Integer intrinsic device functions from the CUDA math api. [7]

__device__ void toNextLeft(int d) {
  unsigned firstRight = __ffs(index[d]);
  index[d] = index[d] >> (firstRight + 1 );
}

__device__ inline int depth(int d) {
  if(index[d]==0) return 0;
  return __clz(index[d]) ^ 31;
}

__device__ void toNextRight(int d) {
  unsigned inverted = ~(index[d]);
  unsigned firstRight = __ffs(inverted);
  index[d] = index[d] >> (firstRight + 1 );
}

Figure 49: Example CUDA implementation of some of the basic tree structure methods for the GPU. These implementations make use of some special CUDA device intrinsics from the CUDA math API [7] to calculate the results more efficiently. The resulting implementations are nearly identical to the CPU versions featured in section 5.

Table 18 shows the improved runtime of a single core process using CUDA for the stencil calculation.
Table 18: [Ref. system: TinyGPU (Table 1)][6D. Solver: see Table 13]

Possible further CUDA optimizations

The current CUDA implementation is rudimentary and not optimized. Following some possible further techniques for GPU programming that could improve performance even further.

Array of Structures (AoS) or Structure of Arrays (SoA): Currently, all the data for the GPU is contained in a single array where, depending on the problem’s dimensionality, the first \(n\) entries are reserved for CUDA thread 0, the next \(n\) for thread 1, and so on. This access pattern is called AoS and clashes with the CUDA-recommended SoA pattern. For efficient GPU memory access, the term memory coalescing is used. Memory coalescing describes memory access where each subsequent GPU thread will read from subsequent memory locations. Restructuring the array containing data blocks of size \(n\) to be \(n\) arrays containing continuous data will most likely improve performance.

Shared memory: Currently, only global GPU memory is used. By using shared memory, memory access times could be improved. The memory latency is of shared memory roughly 100x lower than that of global memory. [8] While a performance increase of a hundredfold is not to be expected, a more realistic increase of around three times is plausible. [9]

Texture memory: is located on the chip and therefore is able to provide higher bandwidth and lower latency. Texture memory is best suited for data points that are updated rarely but read often. In the current small-scale CUDA kernel implementation, the features of this memory type are not used. However, a restructured implementation may make use of it.

Parallel CPU and CUDA operations: In the current implementation, the CPU and GPU are waiting for each other’s computations to finish. This is very resource inefficient. While the GPU is doing calculations, it is possible for the CPU to already prepare the data for the following calculation. This requires an extensive rewrite of the current execution timeline and is out of scope for this paper. By the nature of sparse grids, the number of gridpoints that have to be calculated for every iteration step is fairly small and insufficient to utilize the full parallel capabilities of modern GPUs. That means letting multiple calculation steps be done in parallel on the GPU is also possible.
Figure 50: Differences between the current and a possible async version of the CUDA communication.

Figure 50 shows the current implementation, where the CPU and GPU wait for each other’s task completion, on the left and a possible implementation that lets the CPU and GPU work in parallel on the right. Note that the timings shown in these images are set arbitrarily. In reality, each iteration will take a different amount of time, which would make these graphics needlessly complicated. The right graph shows that in an ideal case, the GPU is only idle in the first iteration while the CPU is preparing the data.

Asynchronous memory transfers: The memory transfers can be made asynchronous. Currently, all data is transferred to the GPU, then used for the calculation, and then transferred back to the host. By segmenting the data, it is possible to transfer a small first segment and start calculating this segment while transferring the second one in parallel. This has the potential to decrease program runtime significantly (see figure 51).

Figure 51: Comparison of the sequential and asynchronous version. [10]
9.5 Combining multiple parallelization techniques

In order to reach better scaling on multiple Nodes, it is required to combine the different parallelization techniques.

9.5.1 MPI/OMP hybrid

The currently implemented MPI parallelization has the problem of only having a limited number of tasks that can be calculated in parallel. Even worse, the number of these tasks depends not on the number of data points that have to be calculated but on the problem’s dimensionality. That means that problems of high fidelity are limited in their performance by their dimensionality. One usually tries to reduce the dimensionality of a given problem to make it more easily computable. This means that a lot of computations then only have a limited number of tasks that could be run in parallel. Because the number of tasks \( t \) depends on the dimensionality \( d \) where \( t = 2^d \), a 5-dimensional program only has \( 2^5 = 32 \) tasks that can be run in parallel. This limits parallelization severely.

A possible fix would be to distribute the currently implemented MPI parallelization on a per-CPU basis. Every task then gets allocated a whole CPU with multiple CPU cores. Each task can then use multithreading to parallelize the computation further. This could allow the program’s parallelization to scale not only with dimensionality but also with the number of points used.

Table 19 shows a speedup of nearly 100 times to the single-core version using 640 CPU cores. This table also shows that the MPI/OMP hybrid version features a higher speedup than the pure MPI parallelization for the same amount of cores. Although the MPI version stops improving at 64 cores, the hybrid version gets noticeable speedups until its maximum of 64 MPI tasks run on 640 CPU cores. (For the AR-Level 1 version see 31)

<table>
<thead>
<tr>
<th>CPU-cores used</th>
<th>Nodes per Node</th>
<th>MPI-Tasks per Node</th>
<th>OMP-Threads per MPI-Task</th>
<th>Runtime in seconds</th>
<th>AR-Level 2</th>
<th>Speedup</th>
<th>Efficiency</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>3366.38</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>20</td>
<td>1</td>
<td>20</td>
<td>0</td>
<td>441.88</td>
<td>7.61</td>
<td>0.38</td>
<td></td>
</tr>
<tr>
<td>40</td>
<td>2</td>
<td>20</td>
<td>0</td>
<td>311.08</td>
<td>10.82</td>
<td>0.27</td>
<td></td>
</tr>
<tr>
<td>60</td>
<td>3</td>
<td>20</td>
<td>0</td>
<td>256.60</td>
<td>13.11</td>
<td>0.22</td>
<td></td>
</tr>
<tr>
<td>80</td>
<td>4</td>
<td>20</td>
<td>0</td>
<td>218.60</td>
<td>15.40</td>
<td>0.19</td>
<td></td>
</tr>
<tr>
<td>100</td>
<td>5</td>
<td>20</td>
<td>0</td>
<td>218.28</td>
<td>15.42</td>
<td>0.15</td>
<td></td>
</tr>
<tr>
<td>20</td>
<td>1</td>
<td>1</td>
<td>10</td>
<td>375.08</td>
<td>8.98</td>
<td>0.45</td>
<td></td>
</tr>
<tr>
<td>40</td>
<td>2</td>
<td>1</td>
<td>10</td>
<td>233.28</td>
<td>14.42</td>
<td>0.72</td>
<td></td>
</tr>
<tr>
<td>80</td>
<td>4</td>
<td>1</td>
<td>10</td>
<td>144.13</td>
<td>23.36</td>
<td>0.30</td>
<td></td>
</tr>
<tr>
<td>160</td>
<td>8</td>
<td>2</td>
<td>10</td>
<td>90.09</td>
<td>37.37</td>
<td>0.23</td>
<td></td>
</tr>
<tr>
<td>320</td>
<td>16</td>
<td>2</td>
<td>10</td>
<td>56.71</td>
<td>59.36</td>
<td>0.19</td>
<td></td>
</tr>
<tr>
<td>640</td>
<td>32</td>
<td>2</td>
<td>10</td>
<td>36.68</td>
<td>91.78</td>
<td>0.14</td>
<td></td>
</tr>
</tbody>
</table>

Table 19: [Ref. system: Meggie (Table 1)] [6D. Solver: see Table 13] MPI vs. MPI/OMP hybrid scaling.

Using CPUs with more cores makes it possible to scale even further. Table 20 shows scaling using the Fritz cluster. With 36 cores per CPU and a maximal number of 64 MPI-tasks, it is possible to use \( 64 \cdot 36 = 2304 \) CPU cores to achieve a speedup of nearly 200 times. (For the AR-Level 1 version see 32) The runtime and efficiency in reference to the number of MPI-tasks used are plotted in figure 52
<table>
<thead>
<tr>
<th>CPU-cores used</th>
<th>Nodes</th>
<th>MPI-Tasks</th>
<th>OMP-Threads per MPI-Task</th>
<th>Runtime AR-Level 2 in seconds</th>
<th>Speedup</th>
<th>Efficiency</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>3532.52</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>36</td>
<td>1</td>
<td>1</td>
<td>36</td>
<td>195.06</td>
<td>18.11</td>
<td>0.50</td>
</tr>
<tr>
<td>72</td>
<td>1</td>
<td>2</td>
<td>36</td>
<td>155.67</td>
<td>22.69</td>
<td>0.32</td>
</tr>
<tr>
<td>144</td>
<td>2</td>
<td>4</td>
<td>36</td>
<td>88.35</td>
<td>39.98</td>
<td>0.28</td>
</tr>
<tr>
<td>288</td>
<td>4</td>
<td>8</td>
<td>36</td>
<td>56.37</td>
<td>62.67</td>
<td>0.22</td>
</tr>
<tr>
<td>576</td>
<td>8</td>
<td>16</td>
<td>36</td>
<td>35.63</td>
<td>99.14</td>
<td>0.17</td>
</tr>
<tr>
<td>1152</td>
<td>16</td>
<td>32</td>
<td>36</td>
<td>23.28</td>
<td>151.74</td>
<td>0.13</td>
</tr>
<tr>
<td>2304</td>
<td>32</td>
<td>64</td>
<td>36</td>
<td>18.41</td>
<td>191.88</td>
<td>0.08</td>
</tr>
</tbody>
</table>

Table 20: [Ref. system: Fritz (Table 1)][6D. Solver: see Table 13] MPI/OMP hybrid scaling using 36 core CPUs.

Figure 52: [Ref. system: Fritz (Table 1)][6D. Solver: see Table 13] Graphically represented runtimes of Table 20. The runtime of the not parallelized cases is so great that it was excluded from this graph.

9.5.2 MPI + CUDA

The current CUDA implementation supports only the usage of one GPU. Support for multi-GPU computing is possible but will be bottlenecked by the single CPU process calculating all non-CUDA computations. A simple solution to this problem is to use MPI and give each MPI task a dedicated GPU to work with. Now it is possible to calculate each case in parallel on different GPUs. Upon further performance analysis, however, it is clear that more than a single case is needed to fully utilize the performance of modern GPUs. Using one GPU for multiple MPI-tasks results in nearly identical performance. Therefore, the implementation now allows multiple MPI tasks to use the same GPU. This allows for high CPU side parallelization while still using only a selected number of GPUs. Table 21 shows the runtime of the MPI implementation combined with the CUDA stencil calculation. Notice how the extra GPU does not significantly increase performance. This shows that current GPUs are capable of calculating for multiple MPI-tasks in parallel without significant performance degradation. Still, even with a high amount of MPI tasks the usage of a dedicated GPU nearly halves the program runtime.

Table 22 and figure 53 show the runtime depending on the number of MPI tasks used with a single GPU.
### Table 21: [Ref. system: TinyGPU (Table 1)][6D. Solver: see Table 13]

<table>
<thead>
<tr>
<th>MPI-Tasks</th>
<th>GPUs</th>
<th>Runtime in sec</th>
<th>Speedup</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0</td>
<td>171.15</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>42.04</td>
<td>4.07</td>
</tr>
<tr>
<td>2</td>
<td>1</td>
<td>27.01</td>
<td>6.34</td>
</tr>
<tr>
<td>2</td>
<td>2</td>
<td>26.70</td>
<td>6.41</td>
</tr>
<tr>
<td>64</td>
<td>0</td>
<td>10.13</td>
<td>16.89</td>
</tr>
<tr>
<td>64</td>
<td>1</td>
<td>5.85</td>
<td>29.26</td>
</tr>
<tr>
<td>64</td>
<td>2</td>
<td>4.32</td>
<td>39.62</td>
</tr>
</tbody>
</table>

### Table 22: [Ref. system: TinyGPU (Table 1)][6D. Solver: see Table 13] Scaling of multiple MPI processes using a single GPU together.

<table>
<thead>
<tr>
<th>MPI-Tasks</th>
<th>GPUs</th>
<th>Runtime in sec</th>
<th>Speedup</th>
<th>Efficiency</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>42.04</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>1</td>
<td>27.01</td>
<td>1.56</td>
<td>0.78</td>
</tr>
<tr>
<td>4</td>
<td>1</td>
<td>16.96</td>
<td>2.48</td>
<td>0.62</td>
</tr>
<tr>
<td>8</td>
<td>1</td>
<td>11.04</td>
<td>3.81</td>
<td>0.48</td>
</tr>
<tr>
<td>16</td>
<td>1</td>
<td>7.97</td>
<td>5.27</td>
<td>0.33</td>
</tr>
<tr>
<td>32</td>
<td>1</td>
<td>6.66</td>
<td>6.31</td>
<td>0.20</td>
</tr>
<tr>
<td>64</td>
<td>1</td>
<td>5.85</td>
<td>7.19</td>
<td>0.11</td>
</tr>
</tbody>
</table>

### Figure 53: [Ref. system: TinyGPU (Table 1)][6D. Solver: see Table 13] Graphically represented runtimes of Table 22. Here one GPU is used in conjunction with a number of MPI-Tasks.

### 9.5.3 MPI/OMP hybrid + CUDA

The usage of all three parallelization types combined has the potential to lead to even greater performance. Unfortunately, the current OMP and CUDA parallelizations make use of the same function to optimize and are therefore incompatible. As a proof of concept, the data fetching for the CUDA calculation is now done with OMP support. The data has to be loaded out of a hashmap into an array for usage with CUDA. Currently, efficient hashmap access is not implemented for the CUDA version; therefore, the CPU has to complete this task. Each load takes a lot of time, and there are hundreds of thousands of them happening. Parallelising this...
gives a small performance boost. The other perk of this approach is that it is now possible to use multiple GPUs more effectively. Most often, there will be multiple CPU cores per GPU on a system, and the number of MPI-Tasks is severely limited by the dimensionality d of the problem. With the maximum number of MPI tasks being $2^d$. And the actually efficient number of MPI tasks being way below this number (see Table 22). On an example system consisting of multiple nodes with one 32-core CPU and one GPU, each this would prevent a 5-dimensional problem to be solved on multiple GPUs without having some CPU cores do nothing because no more MPI tasks are available. A 5-dimensional problem would have $2^5 = 32$ possible cases resulting in a maximum of 32 MPI tasks. These 32 tasks could all be handled by a single 32-core CPU. Nevertheless, this CPU only has access to one GPU. If one wanted to use a second GPU on a different node, then this would mean that there are now two 32-core CPUs available but only 32 MPI tasks. Resulting in half of the available CPU core count doing no work at all. This problem gets even more exaggerated by the fact that a problem with 32 cases may support 32 MPI-Tasks but will no longer efficiently scale after 16 MPI tasks. With the new MPI/OMP hybrid + CUDA implementation, it is possible for OMP to use the cores that MPI could not. This can be used to have a single MPI task per CPU and GPU without wasting precious computing power. Extending the previous example, this would allow a configuration of one MPI task per node resulting in the usage of 32 GPUs and $32 \cdot 32 = 1024$ CPU cores.

```c
Vector calcStencilCUDA(vector<Point> ps, vector<double> us);

void applyStencilGPU(Grid &grid, Depth T, Vector &Ax, Vector &u) {
    vector<Point> points;
    for(point : grid){
        if(point.depth < T){
            points.push_back(point);
        }
    }
    vector<double> u_vals(points.size() * directions.size());

    #pragma omp parallel for
    for(int i = 0; i < points.size(); i++){
        point = points[i]
        for(int k = 0; k < directions.size(); k++){
            dir = directions[k];
            Point np(point,dir);
            u_vals[i * directions.size() + k] = u[np];
        }
    }

    Ax = calcStencilCUDA(points,u_vals);
}
```

Figure 54: Simplified pseudo-code versions of the applyStencil function.

Figure 54 shows the pseudo-code version of the stencil calculation using OMP and CUDA. Here Compute Unified Device Architecture (CUDA) is used for the stencil calculation and Open Multi-Processing (OMP) for creating the u_vals vector. Currently, OMP is only used to fetch the u values from memory. This task involves searching through the hashmap that stores these values, resulting in many cache misses. Better scaling could be archived by also using the OMP threads on other parts of the code. As otherwise, all OMP threads stay idle for the rest of the program. Still, as seen in Table 23, the performance boost resulting from this straightforward OMP parallelization is significant.
<table>
<thead>
<tr>
<th>MPI-Tasks</th>
<th>OMP-Threads</th>
<th>GPUs</th>
<th>Runtime in sec</th>
<th>Speedup</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>171.15</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
<td>42.04</td>
<td>4.07</td>
</tr>
<tr>
<td>32</td>
<td>0</td>
<td>0</td>
<td>16.27</td>
<td>10.52</td>
</tr>
<tr>
<td>32</td>
<td>0</td>
<td>1</td>
<td>6.66</td>
<td>25.70</td>
</tr>
<tr>
<td>8</td>
<td>32</td>
<td>1</td>
<td>6.36</td>
<td>26.91</td>
</tr>
<tr>
<td>64</td>
<td>0</td>
<td>2</td>
<td>4.32</td>
<td>39.62</td>
</tr>
<tr>
<td>16</td>
<td>64</td>
<td>2</td>
<td>4.09</td>
<td>42.85</td>
</tr>
<tr>
<td>32</td>
<td>128</td>
<td>4</td>
<td>2.70</td>
<td>63.40</td>
</tr>
</tbody>
</table>

Table 23: [Ref. system: TinyGPU (Table 1)] [6D. Solver: see Table 13] Runtimes of the MPI/OMP hybrid + CUDA parallelization on a single node.
10 Summary

10.1 Conclusion

Improving the runtime of performance-intensive calculations is a complex and necessary topic. Many ways exist to improve the runtime of performance-intensive calculations, the ExPDESG library being no exception. This thesis shows that the ExPDESG library still has room for performance optimizations resulting in greatly reduced execution times. The old library version suffered from a great number of unnecessary data transfers, which crippled its performance. Data vectors were copied in their entirety when only a copy of small chunks would have sufficed. Also, some operations that could be performed on the input grid itself required a second output grid bloating memory consumption and resulting in unnecessary memory transfers. Basic access methods have been optimized to make use of modern x86 hardware instructions reducing the runtime of these methods by fractions of a second. This may not sound like much in itself, but after considering that these methods get called extremely often, this decreases the program runtime significantly. Through the introduction of new data structures and rewritten grid access patterns, it was possible to prevent having to loop through all existing data points in order to find just a selected few that are actually needed saving valuable time wasted on equality checks and pipeline stalls resulting from incorrect branch prediction. Some methods have been rewritten to achieve the same result with fewer memory accesses or have been simplified in order for the compiler to better recognize possible optimizations. In conjunction, these optimizations not only decreased the program runtime considerably but also improved its performance scaling behavior when the number of points or dimensions increased. Some new implementations now even have runtimes independent of the number of data points the whole grid contains. This paper also managed to show that this library can be parallelized rather efficiently on different amount of cores depending on the type and dimensionality of the calculations performed. A further parallelization also has been implemented but suffers from less-than-ideal scaling using hundreds of cores. With the added CUDA implementation, it was possible to greatly increase the performance for not only multi-CPU/Node systems but also for ordinary home computers. And an even further hybrid implementation using MPI, OMP, and CUDA allowed for the usage of nearly arbitrary many GPUs and CPUs. This paper, therefore, showed that great improvements in the performance of the current ExPDESG implementation are possible. Further, it showed that it is also possible to reduce the runtime significantly by the usage of parallelization.

10.2 Outlook

This paper featured an overview of performance optimization techniques used for performance-critical tasks. Advanced performance analysis like instruction throughput, latency, and memory bandwidth, as well as usage of special hardware, instructions are able to increase performance significantly. This, unfortunately, often comes at the cost of extreme source code obfuscation of even the simplest of calculations combined with new special conditions required of the input data and underlying data structures, which makes these types of optimizations impractical for libraries that are still actively worked on and whose feature set is not finalized. This paper’s goal was to noticeably increase the performance without changing or obfuscating the existing implementation. The adjustment of data structures, mathematical algorithms, and their implementations may lead to even more significant performance improvements. Such tasks, however, are outside the scope of this paper. Moving from single-core to multi-core calculations, this paper was able to show that parallelization of the used calculations is not only possible but also highly efficient. In its current state though the hybrid parallelization using multiple compute nodes with or without CUDA suffers from high waiting times for unnecessary synchronizations and high idle times of large amounts of CPU and GPU resources. This leads to less-than-ideal speedup scaling for high amounts of processors. Fixing this problem requires a restructure of data structure and calculations to allow them to be run in parallel more efficiently. Being able to use a GPU to accelerate highly parallel computations allows for dramatically reduced runtimes. Currently, the GPU implementation, however, is just a proof of concept. As such, it encompasses only a tiny fraction of the code base and should be subject to further investigation into its feasibility and efficiency.
## A Tables

<table>
<thead>
<tr>
<th>Index: in $\log_2(x)$</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
</tr>
</thead>
<tbody>
<tr>
<td>Original:</td>
<td>33489</td>
<td>34046</td>
<td>77324</td>
<td>116412</td>
<td>133954</td>
<td>150129</td>
<td>166436</td>
<td>187663</td>
</tr>
<tr>
<td>C++20:</td>
<td>69121</td>
<td>69091</td>
<td>69086</td>
<td>69132</td>
<td>69130</td>
<td>69225</td>
<td>69311</td>
<td>69410</td>
</tr>
<tr>
<td>Index:</td>
<td>8</td>
<td>9</td>
<td>10</td>
<td>11</td>
<td>12</td>
<td>13</td>
<td>14</td>
<td>15</td>
</tr>
<tr>
<td>Original:</td>
<td>218080</td>
<td>238684</td>
<td>236343</td>
<td>277537</td>
<td>293895</td>
<td>284476</td>
<td>305296</td>
<td>334812</td>
</tr>
<tr>
<td>C++20:</td>
<td>69378</td>
<td>69104</td>
<td>69412</td>
<td>70220</td>
<td>69322</td>
<td>70314</td>
<td>69127</td>
<td>69534</td>
</tr>
<tr>
<td>Index:</td>
<td>16</td>
<td>17</td>
<td>18</td>
<td>19</td>
<td>20</td>
<td>21</td>
<td>22</td>
<td>23</td>
</tr>
<tr>
<td>Original:</td>
<td>354892</td>
<td>367798</td>
<td>388353</td>
<td>420942</td>
<td>423396</td>
<td>440469</td>
<td>458068</td>
<td>470936</td>
</tr>
<tr>
<td>C++20:</td>
<td>69519</td>
<td>69044</td>
<td>69213</td>
<td>69795</td>
<td>69373</td>
<td>69516</td>
<td>69035</td>
<td>69365</td>
</tr>
<tr>
<td>Index:</td>
<td>24</td>
<td>25</td>
<td>26</td>
<td>27</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Original:</td>
<td>489991</td>
<td>505338</td>
<td>519783</td>
<td>547143</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>C++20:</td>
<td>69636</td>
<td>69426</td>
<td>69474</td>
<td>69182</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Table 24: [Ref. system: Work (Table 1)] Runtimes in ns of two depth(index) implementations for different input indices. The runtime of these tests is measured for $2^{16}$ iterations. The original implementation suffered from rising runtime cost for higher input indices in ns. This has been resolved in all new optimizations.

<table>
<thead>
<tr>
<th>Depth:</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
<th>10</th>
<th>11</th>
</tr>
</thead>
<tbody>
<tr>
<td>Dim 1:</td>
<td>20481</td>
<td>18434</td>
<td>16388</td>
<td>14344</td>
<td>12304</td>
<td>10272</td>
<td>8256</td>
<td>6272</td>
<td>4352</td>
<td>2560</td>
<td>1024</td>
</tr>
<tr>
<td>Dim 2:</td>
<td>20481</td>
<td>18434</td>
<td>16388</td>
<td>14344</td>
<td>12304</td>
<td>10272</td>
<td>8256</td>
<td>6272</td>
<td>4352</td>
<td>2560</td>
<td>1024</td>
</tr>
<tr>
<td>Dim 3:</td>
<td>20481</td>
<td>18434</td>
<td>16388</td>
<td>14344</td>
<td>12304</td>
<td>10272</td>
<td>8256</td>
<td>6272</td>
<td>4352</td>
<td>2560</td>
<td>1024</td>
</tr>
</tbody>
</table>

Table 25: Gridpoint distribution of regular sparse grid containing 114,687 points with 13 levels in 3 dimensions.

<table>
<thead>
<tr>
<th>Depth:</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
</tr>
</thead>
<tbody>
<tr>
<td>Dim 1:</td>
<td>0</td>
<td>98305</td>
<td>90114</td>
<td>81924</td>
<td>73736</td>
<td>65552</td>
<td>57376</td>
<td>49216</td>
</tr>
<tr>
<td>Dim 2:</td>
<td>0</td>
<td>98305</td>
<td>90114</td>
<td>81924</td>
<td>73736</td>
<td>65552</td>
<td>57376</td>
<td>49216</td>
</tr>
<tr>
<td>Dim 3:</td>
<td>0</td>
<td>98305</td>
<td>90114</td>
<td>81924</td>
<td>73736</td>
<td>65552</td>
<td>57376</td>
<td>49216</td>
</tr>
</tbody>
</table>

Table 26: Gridpoint distribution of regular sparse grid containing 647,167 points with 15 levels in 3 dimensions.
<table>
<thead>
<tr>
<th>t:</th>
<th>Original:</th>
<th>Hash:</th>
<th>Inverted:</th>
<th>Inplace:</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>2528</td>
<td>2562</td>
<td>21457230</td>
<td>20245011</td>
</tr>
<tr>
<td>1</td>
<td>45801551</td>
<td>9137605</td>
<td>24813986</td>
<td>24756999</td>
</tr>
<tr>
<td>2</td>
<td>65637620</td>
<td>14302203</td>
<td>20790221</td>
<td>20258641</td>
</tr>
<tr>
<td>3</td>
<td>55463012</td>
<td>18044625</td>
<td>21354467</td>
<td>15533028</td>
</tr>
<tr>
<td>4</td>
<td>69448953</td>
<td>22271273</td>
<td>18081134</td>
<td>14069211</td>
</tr>
<tr>
<td>5</td>
<td>97981626</td>
<td>25162151</td>
<td>13876678</td>
<td>11101122</td>
</tr>
<tr>
<td>6</td>
<td>121401248</td>
<td>31005196</td>
<td>8918432</td>
<td>8623510</td>
</tr>
<tr>
<td>7</td>
<td>172219086</td>
<td>28764820</td>
<td>7552090</td>
<td>10129138</td>
</tr>
<tr>
<td>8</td>
<td>168521551</td>
<td>33926537</td>
<td>6105244</td>
<td>8097624</td>
</tr>
<tr>
<td>9</td>
<td>188953571</td>
<td>37322510</td>
<td>5573042</td>
<td>3859939</td>
</tr>
<tr>
<td>10</td>
<td>178153483</td>
<td>32338466</td>
<td>4059372</td>
<td>1811532</td>
</tr>
<tr>
<td>11</td>
<td>167894690</td>
<td>33928466</td>
<td>2014361</td>
<td>839930</td>
</tr>
<tr>
<td>12</td>
<td>186007868</td>
<td>43847982</td>
<td>302176</td>
<td>266205</td>
</tr>
<tr>
<td>13</td>
<td>243700556</td>
<td>38614245</td>
<td>3295</td>
<td>2962</td>
</tr>
<tr>
<td>14</td>
<td>222838608</td>
<td>32077899</td>
<td>3195</td>
<td>2865</td>
</tr>
<tr>
<td>15</td>
<td>200943069</td>
<td>32045802</td>
<td>3083</td>
<td>2826</td>
</tr>
<tr>
<td>16</td>
<td>242263981</td>
<td>38452697</td>
<td>3016</td>
<td>3515</td>
</tr>
</tbody>
</table>

Table 27: [Ref. system: Work (Table 1)] \( \text{restriction1D}(t, d) \) runtimes in ns of an 3-dimensional grid of level 15 in direction \( d = 0 \).

<table>
<thead>
<tr>
<th>t:</th>
<th>Original:</th>
<th>Hash:</th>
<th>Inverted:</th>
<th>Inplace:</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1167</td>
<td>1157</td>
<td>1019742</td>
<td>676515</td>
</tr>
<tr>
<td>1</td>
<td>898941</td>
<td>1160135</td>
<td>1121266</td>
<td>886197</td>
</tr>
<tr>
<td>2</td>
<td>1687889</td>
<td>2033231</td>
<td>1099353</td>
<td>860536</td>
</tr>
<tr>
<td>3</td>
<td>2414665</td>
<td>2581495</td>
<td>872641</td>
<td>716676</td>
</tr>
<tr>
<td>4</td>
<td>2835790</td>
<td>3178422</td>
<td>720053</td>
<td>589582</td>
</tr>
<tr>
<td>5</td>
<td>4160714</td>
<td>3631707</td>
<td>680761</td>
<td>486984</td>
</tr>
<tr>
<td>6</td>
<td>5127783</td>
<td>3985863</td>
<td>459149</td>
<td>369991</td>
</tr>
<tr>
<td>7</td>
<td>6172709</td>
<td>4271216</td>
<td>424356</td>
<td>266440</td>
</tr>
<tr>
<td>8</td>
<td>8102657</td>
<td>4726021</td>
<td>261310</td>
<td>170725</td>
</tr>
<tr>
<td>9</td>
<td>6005063</td>
<td>4597175</td>
<td>160800</td>
<td>96207</td>
</tr>
<tr>
<td>10</td>
<td>5875282</td>
<td>4869870</td>
<td>53367</td>
<td>37771</td>
</tr>
<tr>
<td>11</td>
<td>5347475</td>
<td>4698593</td>
<td>1829</td>
<td>1292</td>
</tr>
<tr>
<td>12</td>
<td>7341071</td>
<td>4646120</td>
<td>1731</td>
<td>1334</td>
</tr>
<tr>
<td>13</td>
<td>7033878</td>
<td>4861949</td>
<td>2680</td>
<td>1264</td>
</tr>
<tr>
<td>14</td>
<td>6913174</td>
<td>4874307</td>
<td>2323</td>
<td>1290</td>
</tr>
</tbody>
</table>

Table 28: [Ref. system: Work (Table 1)] \( \text{restriction1D}(t, d) \) runtimes in ns of an 3-dimensional grid of level 15 in direction \( d = 0 \).
### Table 29: \textit{prolongation1D(t, d)} runtimes in ns of an 3-dimensional grid of level 13 in direction $d = 0$.

<table>
<thead>
<tr>
<th>t</th>
<th>Original:</th>
<th>Inplace:</th>
<th>Hash:</th>
<th>Inverted:</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1380</td>
<td>1047</td>
<td>1140</td>
<td>1173</td>
</tr>
<tr>
<td>1</td>
<td>915211</td>
<td>649519</td>
<td>549613</td>
<td>1158</td>
</tr>
<tr>
<td>2</td>
<td>879353</td>
<td>701883</td>
<td>749573</td>
<td>645731</td>
</tr>
<tr>
<td>3</td>
<td>633783</td>
<td>595749</td>
<td>765163</td>
<td>876447</td>
</tr>
<tr>
<td>4</td>
<td>479745</td>
<td>455747</td>
<td>711831</td>
<td>904403</td>
</tr>
<tr>
<td>5</td>
<td>384041</td>
<td>384417</td>
<td>634828</td>
<td>831352</td>
</tr>
<tr>
<td>6</td>
<td>325277</td>
<td>313081</td>
<td>537988</td>
<td>729412</td>
</tr>
<tr>
<td>7</td>
<td>249701</td>
<td>241504</td>
<td>441615</td>
<td>596481</td>
</tr>
<tr>
<td>8</td>
<td>172913</td>
<td>175084</td>
<td>343193</td>
<td>454347</td>
</tr>
<tr>
<td>9</td>
<td>113088</td>
<td>114369</td>
<td>231286</td>
<td>5553</td>
</tr>
<tr>
<td>10</td>
<td>66305</td>
<td>66549</td>
<td>131623</td>
<td>188789</td>
</tr>
<tr>
<td>11</td>
<td>23968</td>
<td>22929</td>
<td>50830</td>
<td>78317</td>
</tr>
<tr>
<td>12</td>
<td>1659</td>
<td>1157</td>
<td>1066</td>
<td>5720</td>
</tr>
<tr>
<td>13</td>
<td>1651</td>
<td>1036</td>
<td>1112</td>
<td>5553</td>
</tr>
<tr>
<td>14</td>
<td>1376</td>
<td>1092</td>
<td>1096</td>
<td>5570</td>
</tr>
</tbody>
</table>

### Table 30: \textit{prolongation1D(t, d)} runtimes in ns of an 9-dimensional grid of level 15 in direction $d = 0$.

<table>
<thead>
<tr>
<th>CPU-cores</th>
<th>Nodes</th>
<th>MPI-Tasks per Node</th>
<th>OMP-Threads per MPI-Task</th>
<th>Runtime AR-Level 1 in seconds</th>
<th>Speedup</th>
<th>Efficiency</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>160</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>10</td>
<td>1</td>
<td>20</td>
<td>0</td>
<td>22.99</td>
<td>6.96</td>
<td>0.35</td>
</tr>
<tr>
<td>40</td>
<td>2</td>
<td>20</td>
<td>0</td>
<td>16.93</td>
<td>9.45</td>
<td>0.24</td>
</tr>
<tr>
<td>60</td>
<td>3</td>
<td>20</td>
<td>0</td>
<td>14.03</td>
<td>11.40</td>
<td>0.19</td>
</tr>
<tr>
<td>80</td>
<td>4</td>
<td>20</td>
<td>0</td>
<td>12.44</td>
<td>12.86</td>
<td>0.16</td>
</tr>
<tr>
<td>100</td>
<td>5</td>
<td>20</td>
<td>0</td>
<td>12.42</td>
<td>12.88</td>
<td>0.13</td>
</tr>
<tr>
<td>200</td>
<td>10</td>
<td>2</td>
<td>10</td>
<td>20.06</td>
<td>7.98</td>
<td>0.40</td>
</tr>
<tr>
<td>400</td>
<td>20</td>
<td>2</td>
<td>10</td>
<td>12.66</td>
<td>12.64</td>
<td>0.32</td>
</tr>
<tr>
<td>800</td>
<td>40</td>
<td>2</td>
<td>10</td>
<td>7.96</td>
<td>20.10</td>
<td>0.25</td>
</tr>
<tr>
<td>1600</td>
<td>80</td>
<td>2</td>
<td>10</td>
<td>5.03</td>
<td>31.91</td>
<td>0.20</td>
</tr>
<tr>
<td>3200</td>
<td>160</td>
<td>2</td>
<td>10</td>
<td>3.17</td>
<td>50.47</td>
<td>0.16</td>
</tr>
<tr>
<td>6400</td>
<td>320</td>
<td>2</td>
<td>10</td>
<td>2.03</td>
<td>78.82</td>
<td>0.12</td>
</tr>
</tbody>
</table>

Table 31: \textit{[Ref. system: Meggie (Table 1)] [6D. Solver: see Table 13]} MPI/OMP hybrid runtimes.
<table>
<thead>
<tr>
<th>CPU-cores used</th>
<th>Nodes</th>
<th>MPI-Tasks</th>
<th>OMP-Threads per MPI-Task</th>
<th>Runtime in seconds</th>
<th>Speedup</th>
<th>Efficiency</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>168.90</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>36</td>
<td>1</td>
<td>1</td>
<td>36</td>
<td>10.89</td>
<td>15.51</td>
<td>0.43</td>
</tr>
<tr>
<td>72</td>
<td>1</td>
<td>2</td>
<td>36</td>
<td>7.70</td>
<td>21.94</td>
<td>0.15</td>
</tr>
<tr>
<td>144</td>
<td>2</td>
<td>4</td>
<td>36</td>
<td>4.90</td>
<td>34.47</td>
<td>0.12</td>
</tr>
<tr>
<td>288</td>
<td>4</td>
<td>8</td>
<td>36</td>
<td>3.14</td>
<td>53.79</td>
<td>0.19</td>
</tr>
<tr>
<td>576</td>
<td>8</td>
<td>16</td>
<td>36</td>
<td>2.10</td>
<td>80.43</td>
<td>0.14</td>
</tr>
<tr>
<td>1152</td>
<td>16</td>
<td>32</td>
<td>36</td>
<td>1.43</td>
<td>118.11</td>
<td>0.10</td>
</tr>
<tr>
<td>2304</td>
<td>32</td>
<td>64</td>
<td>36</td>
<td>0.95</td>
<td>177.79</td>
<td>0.08</td>
</tr>
</tbody>
</table>

Table 32: [Ref. system: Fritz (Table 1)] [6D. Solver: see Table 13] MPI/OMP hybrid runtimes.
References


Acronyms

**AR-Level**  Adaptive Refinement-Level

**ASMx86**  Assembly x86

**bsr**  Bit-Scan-Reverse

**clz**  Count-Leading-Zeros

**CPU**  Central Processing Unit

**ctz**  Count-Trailing-Zeros

**CUDA**  Compute Unified Device Architecture

**DoF**  Degrees of Freedom

**ExPDESG**  Expression Templates for Partial Differential Equations on Sparse Grids

**ffs**  Find-First-Set

**GPU**  Graphics Processing Unit

**MPI**  Message Passing Interface

**OMP**  Open Multi-Processing

**SIMD**  Single Instruction Multiple Data