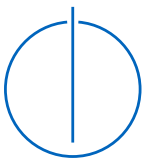
DEPARTMENT OF INFORMATICS

TECHNISCHE UNIVERSITÄT MÜNCHEN

Bachelor's Thesis in Informatics

Shared-Memory Parallelization of a Parallel Combination Technique Framework

Sven Hingst



DEPARTMENT OF INFORMATICS

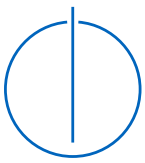
TECHNISCHE UNIVERSITÄT MÜNCHEN

Bachelor's Thesis in Informatics

Shared-Memory Parallelization of a Parallel Combination Technique Framework

Shared-Memory Parallelisierung eines parallelen Kombinationstechnikframeworks

Author:	Sven Hingst
Supervisor:	Prof. Dr. rer. nat. habil. Hans-Joachim Bungartz
Advisor:	M. Sc. Michael Obersteiner
Submission Date:	November 15, 2019



I confirm that this bachelor's thesis in informatics is my own work and I have documented all sources and material used.

Munich, November 15, 2019 Sven Hingst

Acknowledgments

I would like to thank my advisor Michael Obersteiner, for his advice and regular meetings
I would like to thank my parents for their support

Abstract

Numerically solving high dimensional partial differential equations(PDEs) is computationally difficult due to the curse of dimensionality. The sparse grid combination technique partially mitigates this problem. A highly parallel framework implementing this technique is the distributed combigrid module of SG++ which is currently only parallelized with message passing. Combining message passing with shared memory parallelization often yields better performance. In this thesis shared-memory parallelization is added to the distributed combigrid module. This allows users of the framework to use hybrid parallelization in their pde-solvers. Speedups up to 3 were measured, in many cases moderate speedups can be seen but for some cases minor slowdowns occurred in comparison to the original version. These slowdowns should still enable using hybrid parallelization for users of the framework and make it worthwhile.

Contents

1 Introduction

Solving high-dimensional partial differential equations is an important field in high performance computing. High-dimensional partial differential equations are affected by the curse of dimensionality, which means that the computational complexity grows exponentially in the number of dimensions.
Sparse grids [BG04] reduce the exponential impact of the dimensions to only act on a logarithmic part, without significantly impacting the approximation accuracy. Direct usage of sparse grids would require a complete redesign of the applied numerical algorithms. The combination technique [GSZ90] achieves similar results, by combining multiple standard instances of the used numerical algorithm.
The "distributed combigrid module" of SG++ implements this approach as a library for arbitrary time stepping algorithms. It can be used by implementing an interface for your own application. This allows usage of the combination technique for a wide variety of applications. The library is designed for large scale clusters, even for upcoming exascale computers. It is parallelized by message passing between processes with MPI. In high performance computing many programs combine message passing (MPI) with shared memory programming (with usually OpenMP) for better performance. In this thesis I parallelized this library to allow users of the distributed combigrid module to improve their applications by using shared memory parallelization.

In chapter 2 an introduction into sparse grids and the combination technique is given. Additionally hybrid OpenMP and MPI programming is explained and an overview of the modified code base is given. Chapter 3 outlines the modifications and additions of the code base that were done for shared-memory parallelization. In chapter 4 those changes are analyzed regarding the parallel efficiency. Finally the hybrid approach is compared against the previous MPI-only version.

2 Background

This chapter gives a short introduction into sparse grids[BG04] and the combination technique [GSZ90]. This introduction is based on [Hee18],[Pfl10], [GG10] and [OB19]. Additionally used features from MPI and OpenMp are explained and an overview of the used code base from [Hee18] is given.

2.1 Notation

To become better distinguishable from scalars, vectors are printed in bold. Vector comparisons are defined in the following way:

$$\mathbf{a} \leq \mathbf{b} \Leftrightarrow \forall i.a_i \leq b_i \tag{2.1}$$

$$\mathbf{a} < \mathbf{b} \Leftrightarrow \forall i.a_i < b_i \tag{2.2}$$

$$\mathbf{a} \geq \mathbf{b} \Leftrightarrow \forall i.a_i \geq b_i \tag{2.3}$$

$$\mathbf{a} > \mathbf{b} \Leftrightarrow \forall i.a_i > b_i \tag{2.4}$$

Note that $\mathbf{a} \not\leq \mathbf{b} \not\Rightarrow \mathbf{a} \geq \mathbf{b}$. The i^{th} unit vector is denoted as $\mathbf{u_i}$. $\mathbb{1}$ (or $\mathbb{1}^d$) is a vector of the appropriate (d) dimension containing only 1s. Regular grids are grids with dimensionally equidistant points, i.e. a vector $\mathbf{h} \in \mathbb{R}^n$ is used to describe the distances between points .

2.2 Nodal Basis

For many numerical methods functions are interpolated over a finite number of support points. Let I be the set of coordinates of support points, $\phi_i(x)$ the support function and α_i the corresponding contributions of the basis functions. Then the interpolant $u(x)$ is defined as:

$$f(x) \approx u(x) := \sum_{i \in I} \alpha_i \phi_i(x) \tag{2.5}$$

with ϕ_i being a so called basis function, denoting the impact of each support point on the current position. Basis functions that are used, have usually only local support

centered around its associated point to avoid problems with numerical stability and computation time. For finite elements methods linear basis functions are often used. In one dimension this basis (also called hat function), is 1 at its associated point and 0 at its neighboring support points and linear in between. For regular grids with distance h between support points, in one dimension the function is (see Fig. 2.1)

$$\phi_i(x) = max(1 - \frac{|x_i - x|}{h}, 0) \tag{2.6}$$

For multiple dimensions the basis function is generated, by a tensor product of hat function for each dimension.

$$\phi_i = \prod_{k=1}^{d} max(1 - \frac{|(\mathbf{x_i})_k - \mathbf{x}_k|}{h_k}, 0) \tag{2.7}$$

The set of basis functions of all used coordinates form a function space, which simplifies notation and can be helpful for proofs.

$$V_I = span\{\phi_i : i \in I\} \tag{2.8}$$

2.3 Hierarchical Basis

Another representation is the hierarchical basis, which facilitates adapting the discretization level and simplifies the interpolation of differently discretized grids.
This basis is usually constructed with regular grids in which the number of grid points in each dimension depends on a power of two. For better readable formulas the

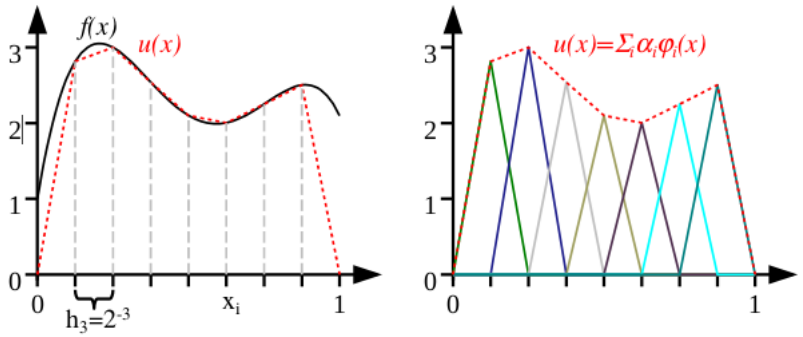


Figure 2.1: Interpolation of an example function f. Each basis function has a different color. Taken from [Pfl10].

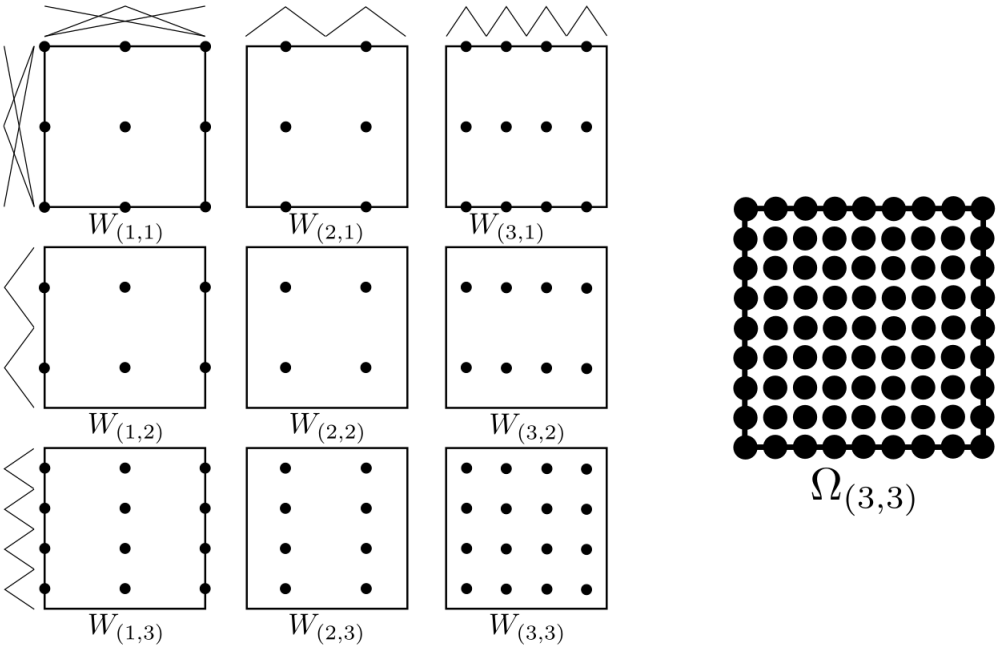


Figure 2.2: Subspaces of the full grid $\Omega_{(3,3)^T}$ with boundary points. Taken from [Hee18].

problem domain is projected onto $[0,1]^d$.
For such grids (called (regular) full grid) the level vector $\mathbf{l} \in \mathbb{N}^d$ with l_i denotes $2^i \pm 1$ grid points in the i^{th} dimension (+1 if boundaries exists in that direction -1 if not). A level $\mathbf{l}$ is called higher than a level $\mathbf{j}$ if $\mathbf{l} \leq \mathbf{j}$ and $\mathbf{l} \neq \mathbf{j}$ hold and lower if $\mathbf{l} \geq \mathbf{j}$ and $\mathbf{l} \neq \mathbf{j}$ hold (not equal to $<$ or $>$ see the definition of vector comparisons in 2.1).
Grid points have indices $\mathbf{i}$ such that its coordinates are $\sum_{j=1}^{d} 2^{-\mathbf{i}_j}\mathbf{u}_\mathbf{j}$. If dimension j has boundary points then its indices range from $[0, l_j]$ else from $[1, l_j - 1]$
The hierarchical basis is composed of subspaces. A subspace with level $\mathbf{l} \in \mathbb{N}^d$ consists of all points of the full grid of the same level that are not contained in full grids (and thus subspaces) of a higher level. The subspace of level $\mathbb{1}$ consist of the central point $0.5 \cdot \mathbb{1}$ and the central and corner boundary points for dimensions for which they exist. Therefore subspaces have $2^{\mathbf{l}_j}$ points in the j^{th} dimension or 3 points if a boundary exists for dimension j and $\mathbf{l}_j = 1$. Figure 2.2 shows the subspaces of a grid of level $3 \cdot \mathbb{1}$ with boundary points.

The basis functions $\phi^\mathbf{l}$ of a subspace with level is the same as ϕ of a full grid of the same level (e.g. hat functions with $h_i = 2^{\mathbf{l}_i}$).

The full grid with levelvector $\mathbf{n}$ is composed of all subspaces with equal or higher level than n. A subspace $W_\mathbf{l}$ with level $\mathbf{l}$ can also be interpreted as the span of basis functions of all grid points in the subspace. This allows using the tensor sum to compose the full grid.

$$V_\mathbf{n} = \bigoplus_{\mathbb{1} \leq \mathbf{l} \leq \mathbf{n}} W_\mathbf{l} \tag{2.9}$$

$$f(x) \approx \sum_{\mathbb{1} \leq \mathbf{l} \leq \mathbf{n}} \sum_{i \in W_\mathbf{l}} \phi_i^\mathbf{l}(x) w_i^\mathbf{l} \tag{2.10}$$

w_i^l are the hierarchical coefficients at coordinates i and the level l of the subspace is given. Those coefficients are not the actual value at the grid points.

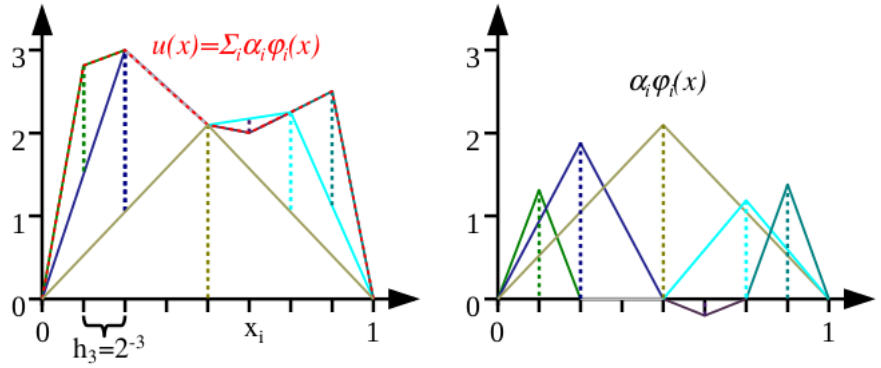


Figure 2.3: Interpolation with the hierarchical Basis in 1d of the same example function f as in Fig.2.1. Each hat function has a different color. Taken from [Pfl10].

The hierarchical predecessors of a point x in a subspace of level l are its neighboring points (i.e the indices differ by at most $\mathbb{1}$) in the fullgrid of level l.

2.3.1 Hierarchization and Dehierarchization

Hierarchization transforms a grid from a nodal basis to its hierarchical basis, dehierarchization is the inverse operation. Nodal points represent the actual function value on that point. In contrary, in the hierarchical basis all basis functions have to be evaluated at $\mathbf{x_i}$ and summed up to get the function value f_i of the grid point i. Therefore the values of basis functions of other points have to be subtracted by hierarchization. Only points in a subspace of higher level than the subspace i is in, will have a non zero basis function at i. As this also applies for their predecessors nodal point, the hierarchical point w_i can also be generated by subtracting the nodal values of its hierarchical predecessors (with respect to its basis function on this position) For linear basis functions we

get:

$$w_i = \sum_{j \in \{-1,0,1\}^d} (-2)^{-\|j\|_1} w_{i+leveloffset_j} \tag{2.11}$$

where $w_{i+leveloffset_j}$ means the nodal point reached by moving dimension wise to the hierarchical predecessor with -1 meaning in negative direction, 0 and 1 in no or positive direction respectively. The positive coefficients originate from those points being contained in the hierarchical representation of an even number of points closer to w_i. The function for one-dimensional linear bases is

$$w_i = f_i - \frac{1}{2}(f_{i+leveloffset_{-u}} + f_{i+leveloffset_{+u}}) \tag{2.12}$$

with u denoting the unit vector (of the corresponding dimension). This formula can be applied for each dimension separately, so that, starting with the second dimension, partially hierarchized values are used. That this formula produces the same results as equation 2.11 is called the unidirectional principle. Figure 2.4 shows an hierarchized example function with a dependency graph of its support points.
Dehierarchization does this in reverse with addition instead of subtracting the nodal predecessors. As hierarchization and dehierarchization both require the nodal values of their predecessors lower subspaces have to be dehierarchized before higher supspaces. When hierarchizing in-place (i.e. replacing nodal values with hierarchical coefficients) , higher subspaces have to be hierarchized before lower subspaces.

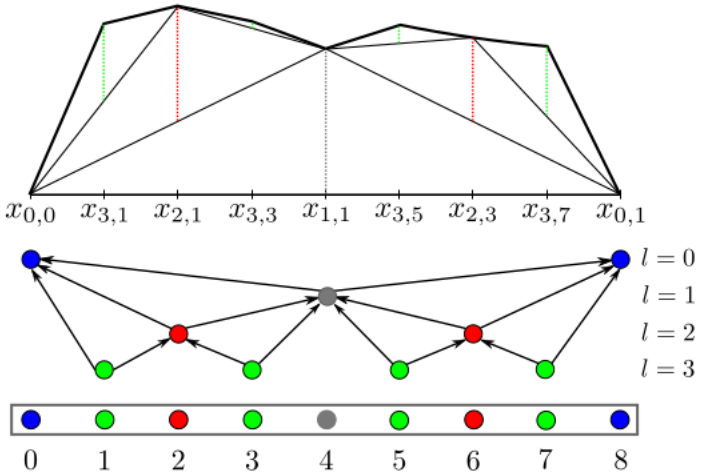


Figure 2.4: Hierarchical predecessors in one dimension. Taken from [Hee18]

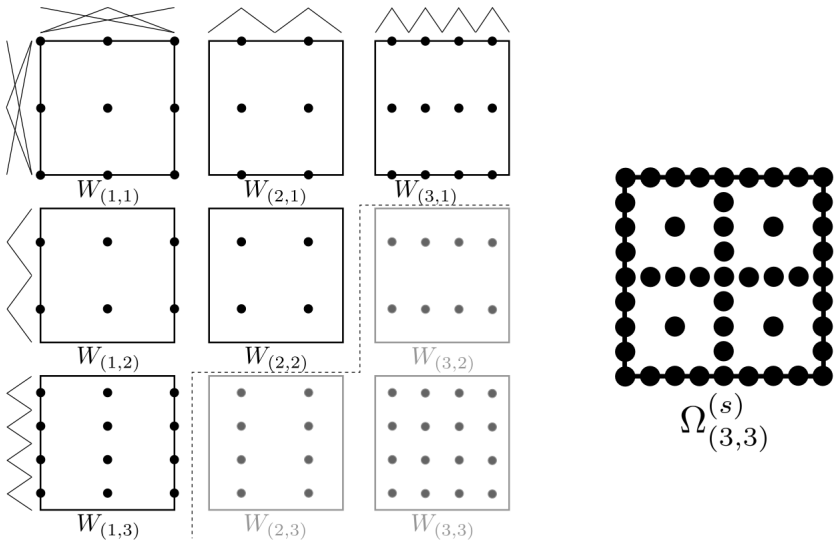


Figure 2.5: Included subspaces in the sparse grid $\Omega^s_{(3,3)^T}$. Taken from [Hee18].

2.4 Sparse Grids

To evade the curse of dimensionality only a subset of subspaces are included in a sparse grid. For functions with bounded mixed second order derivatives and discretization level $\mathbb{1}^d n$, choosing only subspaces up to level $\|\mathbf{l}\|_1 = n + d$ is optimal in terms of computation cost to approximation accuracy ratio. Figure 2.5 shows which subspaces are chosen for a sparse grid with $n = 3$ and Figure 2.6 shows the sparse grid with $n = 6$ and without boundaries. This leads to the number of grid points being reduced from $O(2^{nd})$ to

$$|V| = \sum_{i=0}^{n-1} 2^i \binom{d-1+i}{d-1} \in O(2^n n^{d-1}). \tag{2.13}$$

[GG10] While the approximation accuracy decreases from $O(h_n^2)$ only to $O(h_n^2 n^{d-1})$. Note that $h_n \in \theta(2^{-n})$ and therefore the second term is relatively small.

2.5 Combination Technique

Many PDE-solvers do not work directly on sparse grids/ subspaces and are difficult to parallelize. The (simple) combination technique is adding fullgrids of the lowest included level in the sparse grid together. This results in points being added multiple

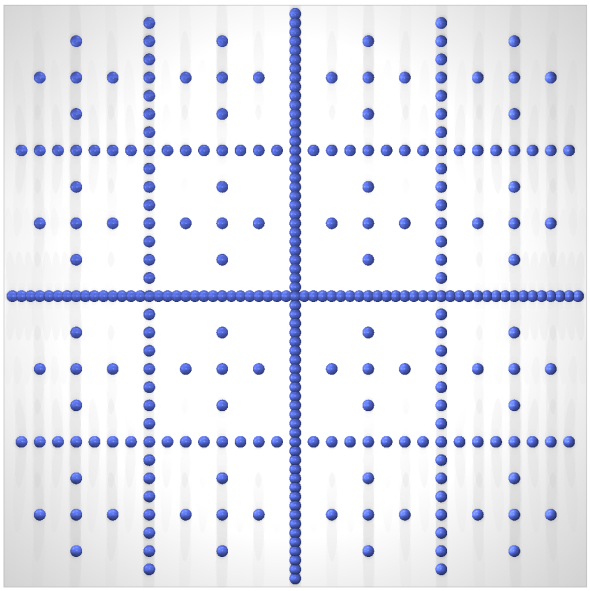


Figure 2.6: Sparse grid of level $n = 6$. Taken from [Pfl10]

times as many subspaces are included in several fullgrids. Those points have to be subtracted which can be done by subtracting fullgrids of lower levels.

$$f \approx u_n^{combi} := \sum_{k=0}^{d-1} (-1)^k \binom{d-1}{k} \sum_{\|\mathbf{l}\|_1 = n+d-1-k} \Omega_{\mathbf{l}} \tag{2.14}$$

As the solvers work on fullgrids, they can and usually will use the nodal basis for grid representation. As different fullgrids have different bases they cannot be summed up directly, hence interpolation would be required. Hierarchizing those grids has the effect that each point has the same basis in all grids it is in. This means that corresponding hierarchical coefficients can be summed up. This method does not give equal results as the sparse grids for PDEs. But for PDEs that fulfill the smoothness assumptions from [GSZ90], the approximation accuracy stays at $O(h_n^2 n^{d-1})$, while the number of grid points increases by a factor of d to $O(d2^n n^{d-1})$.

For anisotropic grids a set $\mathcal{I}$ of fullgrid level vectors can be chosen under the condition that it is downwards closed:

$$\forall i \in \mathcal{I} \forall (k \leq i) \in \mathbb{N}^d . k \in \mathcal{I} \tag{2.15}$$

The combination coefficients c_i (The factor each grid is added by) are calculated by

$$c_i = \sum_{\mathbf{z}=\mathbf{0}^d, \mathbf{z} \in \mathcal{I}}^{\mathbb{1}} (-1)^{\|\mathbf{z}\|_1}. \tag{2.16}$$

Equation 2.17 shows which grids have to be computed, by defining I^* as the set of grids with a combination coefficient other than zero.

$$\mathcal{I}^* = \{i \in \mathcal{I} : i + \mathbb{1} \notin \mathcal{I}\} \tag{2.17}$$

For some PDEs grids with a low number of points in a dimension may become numerical unstable and therefore only grids of a minimum level $\mathbf{l}'_{min}$ can be used. Removing the grids with a too small components would remove the grids with the largest discretization in a dimension and would reduce the target level. To achieve a target level of $\mathbf{n} \in \mathbb{N}^d$ a new minimum level $\mathbf{l}_{min} \geq \mathbf{l}'_{min}$ has to be used.

$$c = \min_{k \in [d]}(\mathbf{n}_k - \mathbf{l}'_{min,k}) \tag{2.18}$$

$$\mathbf{l}_{min} = n - c \cdot \mathbb{1} \tag{2.19}$$

$$\mathcal{I}_{n,\mathbf{l}_{min}} = \{l \in \mathbb{N}^d : \|l\|_1 \leq \|\mathbf{l}_{min}\|_1 + c, \mathbf{l}_{min} \leq \mathbf{n}\} \tag{2.20}$$

Grids with a combination coefficient of 0 can be removed the same way as in equation 2.17.

For time dependent PDEs one combination after all time steps is usually not accurate enough. Therefore after a certain amount of steps the fullgrids are combined into a sparse grid. After that the hierarchical coefficients are copied into their respective fullgrids and those are dehierachized.

2.6 Hybrid MPI/OpenMP Parallelization

MPI, short for Message Passing Interface, is a standard for an inter process and inter node communication library. It is the main communication protocol for programs running on HPC-Clusters that use more than one node. MPI allows processes to send data to other processes. All communication is performed within communicators, which are groups of processes in which each process gets an id called rank. On communicators collective operations can be invoked, like one process broadcasting its data to all other members of the communicator or the **(All)**Reduce function. **(All)**Reduce applies a commutative operation such as addition on all buffers of all members of a communicator and stores the result for a single process (or **all** processes). Those operations come in two different versions the standard one will block until completion, and the asynchronous variants return instantly and have to be manually waited on.
The OpenMP-standard specifies compiler directives and library functions to allow for easier multi-threading. With compiler directives (pragmas for C++) "parallel regions"

are created, which are executed in parallel by a number of threads set at runtime. In parallel regions for-loops can be distributed over all threads, with different scheduling strategies (e.g. evenly or dynamically over a set of indices). Using both OpenMP and MPI is called hybrid parallelization. This allows for tasks running on the same node or NUMA-domain to share memory. This results in faster synchronizations and eliminating the need of sending messages, and therefore a possible speedup. Accessing the same memory also makes other parallelizations strategy viable that may be faster or are more flexible.

2.7 Overview of the Code Base

The main work of this thesis was done on the "distributed combigrid module", a closed source module of SG++ initially developed by [Hee18] and currently worked on by the Technical University of Munich and the University of Stuttgart". This module is a library that implements the sparse grid combination technique with an interface for pde-solver. It is designed for large amounts of processors and cores and parallelized with MPI.

There exists a world manager process that coordinates all process groups, all other processes are called worker processes (even the group master). Each group handles several full grids, in the code usually referred to as tasks. All groups have $2^x = \prod_{i=1}^{d} p_i$ member processes ,with the vector $p \in 2^{\mathbb{N}^d}$ describing in how many parts a task is split per dimension. As not all tasks have the same size, for load balancing reasons, a group should and can have more than one task even if it has more than one member.
Users of the library have to inherit the Task class and implement its initialization and the pde-solver.
In the combination step each worker initializes an empty (all values are 0) *DistributedSparseGrid*, which stores only local points of that process such that each group has a complete sparse grid in total. Afterwards all tasks are hierarchized and added (with their corresponding combination coefficients) to the sparse grid. In the global reduction the subspaces are copied into a single buffer, such that the subspaces align for processes of the same rank in each group. Those buffers are then added up via the MPI_Allreduce function. The subspaces are copied back from the buffer into the sparse grid and from there into their respective Tasks. The tasks are then dehierarchized. In between combination steps a user defined number of time steps is applied. Most data is stored in *std::vector* but they will be in general called arrays unless specific functions of it are relevant.

3 Implementation

This chapter describes in detail what changes were done to add shared-memory parallelization to the existing code base.
The main computational effort of the library is the combination step. Therefore only that step was parallelized. Initialization was kept mostly sequential as it becomes irrelevant in the long run. NUMA-aware initialization is infeasible as hierarchization and dehierarchization stride through all dimensions. Such an implementation would have to essentially reimplement the splitting of grids of the MPI implementation, while at best providing negligible improvements.

3.1 Hierarchization and Dehierarchization

In [Hee18] it is stated that hierarchization and dehierarchization takes most of the time during the combination step.

In the single threaded version there exist functions for hierarchization in the first and for other dimensions (referred to as function for the N^{th} dimension) and of those variants for boundaries or for no boundaries in the N^{th} dimension. These functions only differ in small local optimizations and a few initialization steps. Therefore the same parallelization approach was used. The functions with or without boundaries for hierarchization in the N^{th}-dimension differed only in the used kernel, which in the boundary case is in a separate function. Hence, they have been combined by me via a template parameter indicating whether the boundary exists.

For dehierarchization the same original state existed, and for them the functions for the N^{th}-dimension can be combined. Those functions are also mostly similar to the hierarchization functions, apart from looping over levels in the other direction and addition of the grid points instead of subtraction. Those functions were not combined as implementing switches for those mentioned differences with template variables decreases the readability of the code.
To parallelize those functions suitable parts without data dependence have to be found. The necessary data points from other processes are exchanged in advance of hierarchizing/dehierarchizing a dimension and therefore do not hinder this approach.

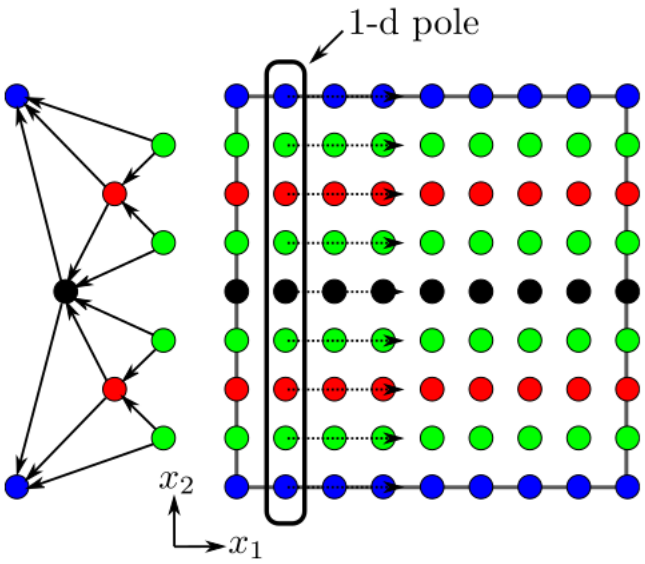


Figure 3.1: 1d-pole in direction of x_2. Taken from [Hee18]

The one-dimensional poles used in the original implementation in theory do not depend on each other.

In practice the only accesses to shared data structures are the elements of the local grid, the data from other processes and a few local variables. As each element is in only one one-dimensional pole in a direction, only caching behavior has to be accounted for. The remote data is read-only and the data structure that contains it, is also thread-safe. The local variables are either constants, or *std::vector*s which only depend on the current 1-d pole but for whose reallocations are avoided. By allocating those separately for each thread, the loop over one-dimensional poles looses its dependencies and can be parallelized with OpenMP pragmas.

3.1.1 Addition to and Extraction from Sparse Grids

On each process each task stores its grid points continuous in memory in the so called elements vector. The *DistributedSparseGrid* has an array for each subspace in which the points are stored in the same order as in the tasks. Each task has an array which acts as a map from the index in the elements vector to a subspace Id.
During addition or extraction the single threaded version creates iterators for all subspaces. Afterwards it loops over the elements vector and looks up the correct subspace iterator via the map. During addition the current element (with combination coefficient) is added to the sparse grid. During extraction the element from the iterator is copied

```
Function hierarchizeN<bool boundary>(remoteData,dim)
    Initialization;
    #pragma omp parallel
        Allocation of std::vectors ;
        #pragma omp for
            forall 1-d pole in dimension dim do
                collect data of pole;
                if boundary then
                    boundary hierarchization kernel;
                else
                    hierarchization kernel;
                end
            end
        end
    end
```

Algorithm 1: Schema of of a hierarchization function specifically for other dimensions than the first

back to the elements vector. Then the iterator is advanced by 1. See algorithm 2, while ignoring the orange parts.
The usage of iterators makes directly parallelizing impossible. The extraction could be done in parallel for each task, but would cause load imbalance when groups do not have enough Tasks (that already wary largely in size).
I therefore parallelized the iterator version. Here, during the initialization of the full grid, after the subspace map is created, each thread gets assigned an interval of the elements vector. The size of the interval is $\left\lfloor \frac{elements.size()+threadId}{num\_threads} \right\rfloor$. Each thread then sums up how many grid points of the interval of the previous thread are in each subspace. The results are added up such that for each thread for each subspace the position for inserting and extracting is known (see algorithm 3).
These changes during the initialization, allows creation of individual iterators for each threads and advancing them to the correct points. Algorithm 2 shows the modifications needed for parallelizing both addition and extraction from sparse grids. It uses *std::advance*, which forwards the iterator it gets as the first argument by the number of elements specified in the second argument. As subspaces are stored as *std::vector* this is a fast operation in constant time, as simple pointer arithmetic is used.

```
#pragma omp parallel
    threadId=omp_get_thread_num();
    forall subspaceId that exists in this grid do
        it_sub[subspaceId]=getSubspaceIterator(subspaceId);
        std::advance(it_sub[subspaceId],
          threadAssignmentListStart[threadId][subspaceId]);
    end
    for i ← threadStart[threadIId] to threadStart[threadId+1]-1 do
        *it_sub[subspaceAssigmentList[i]] += CombinationCoefficent * elements[i];
        it_sub[subspaceAssigmentList[i]]++;
    end
end
```

Algorithm 2: Addition to sparse grid. The changes made to parallelize it are in orange. In the sequential version the second for loop iterates from 0 to $elements.size() - 1$.

```
#pragma omp parallel
    threadId=omp_get_thread_num();
    if threadId != 0 then
        for i ← threadStart[threadId-1] to threadStart[threadId]-1 do
            threadAssignmentListStart[threadId][assigmentList[i]]++;
        end
    end
end
for i ← 2 to num_threads do
    forall subspaceId that exists in this grid do
        threadAssignmentListStart[i][subpaceId
          ]+=threadAssignmentListStart[i-1][subspaceId];
    end
end
```

Algorithm 3: Pre-calculation of iterator shifts for usage in addition and extraction from sparse grid

3.2 Global Reduction

Another important part of the combination step is the global reduction step, which handles the combination of the sparse grids of different groups. In the global reduction each process uses a single *MPI_Allreduce* call to add all subspaces together. No

multiplications with combination coefficients are needed as those happen during the addToSparseGrid function. The subspaces of the sparse grid are not continuous in memory to allow changes to the sparse grid like spatial adaptivity.As not all subspaces are on each process' part of the distributed sparse grid, the sizes of those are exchanged. With that information a buffer of the total size is constructed and the subspaces are copied into it.
When using the hybrid variant, groups usually have fewer members, which increases the size of those buffers on individual processes. Therefore parallelizing this part is important although it runs for comparatively little time (obviously shorter than addition or extraction from the sparse grid). While summing up the total size of the subspaces, the start indices of each subspace are stored. This allows copying in parallel, by iterating over subspaces. As subspaces differ largely in sizes, a dynamic schedule is used. For copying the data back from the buffer the same strategy is applied.
The buffer was originally a *std::vector* that was initialized with zeros. *std::vector* can neither create uninitialized storage nor can it be initialized in parallel by OpenMP. Those buffers can get up to several GB in size and therefore sequential initialization is not acceptable. Instead memory is allocated via *malloc* and initialization of unused subspaces is done during copying.

During some tests it was found that a single *MPI_Allreduce* takes significantly longer than allreduction over the same amount of data split up over more processes and therefore more global reduce operations. Two strategies to reduce the time of the *MPI_Allreduce* are presented here.
Additionally MPI uses signed 32-bit integer for the buffer size in an *MPI_Allreduce* call, which limits the number of possible values exchanged to $2^{31}-$ (16 GB with doubles), which might become an issue with more than available 64GB of memory per node. Both presented strategies for speedup also solve this problem as smaller buffers are used.

3.2.1 Multi threaded *MPI_Allreduce*

Although in general not being supported very well, MPI can be initialized so that it allows being called from multiple threads in parallel. One possible issue is that the MPI-standard does not require parallel execution of requests from multiple threads, even allowing globally locking the whole library [MPI15]. The goal of this approach is to split the buffer and let several (n) threads execute the allreduction in parallel. For this the used MPI communicator gets duplicated n-times. The buffer gets split in n same sized pieces, which are allreduced with n threads in parallel on their corresponding communicator.

This approach may suffer from mentioned internal locks, which would cause large performance impacts or serialize the allreduction. In [MK19] Intel describes a new feature of the 2019 version their MPI-library, which aims to support such use-cases. The model one has to follow when using it, is called "MPI_THREAD_SPLIT". It requires that every communicator is only used by a single thread and only threads with the same thread id (set by OpenMP) can communicate. To enable those features the multi-threaded versions of the Intel MPI library (*release_mt* or *debug_mt*) have to be used and the PSM2 library and the "MPI_THREAD_SPLIT" model have to be activated via environment variables.
Those requirements of this approach are fulfilled by this implementation so only testing is needed.

3.2.2 Asynchronous

The previous strategy strongly depends on the used MPI-implementation and experimental features of Intel version, which the mpp2 cluster of the lrz does only support since September 27^{th} 2019. Therefore another approach is needed. Like in the first one the buffer is split up and the communicators are duplicated. Instead of using multiple threads multiple asynchronous allreductions are started from a single thread which immediately after that waits for the completion of all of them.

3.3 Smaller changes

At the start of the combination step the parameters sparse grid may change due to special features (adaptivity,reduction of sparse grid coefficients). Therefore the sparse grid object was deleted and a new one, with possibly changed parameters, was created. As the sparse grids are usually larger when using the hybrid variant instead of the pure mpi version, parallelization is needed here. As only specific features change the general structure of the sparse grid, it suffices to zero out the subspaces. This is done in parallel by distributing whole subspaces over all threads. Each sparse grid gets an unique id such that the subspaces of the fullgrids do not have to be remapped.

4 Results

This chapter evaluates the parallel efficiency of the shared-memory parallelization in this thesis and compares it against the MPI-only version.

4.1 Test environment

At first general information on the setup and analysis of the test cases is given.

4.1.1 Time Measurement

The project has an integrated tool for measuring execution times. Each process can start and stop events locally. At the end of the execution each process groups the start- and stop-times by event. This data is then stored in a single JSON-file. Time measurements used in this chapter are acquired in the following ways. The time measured by the world manager was used whenever possible (time of the combination step and the total time). For events that happen once per combination, like the global reduction, the time is taken from the slowest process per combination step and those values are then averaged. For events that occur once per task, like addition to the sparse grid, the times are added up per combination step per process and then proceeded like above.

4.1.2 Used Hardware

The experiments were done on the mpp2-Cluster of the Leibniz Supercomputing Centre. The cluster has 318 nodes of which up to 60 are usable in a single job. Each Node consists of two Intel Xeon E5-2697 v3 (Haswell) 14-core processors and has 64 GB of memory, out of which 56 GB are usable, distributed over 4 memory channels per processor. The nodes are connected via a FDR14 Infiniband interconnect.
The used SLURM scheduler does not assign nodes in continuous blocks. This means the node distribution is essentially random. This can severely impact the performance of MPI calls especially during collectives. Large distances between nodes amplify the impact of other traffic on the cluster.

4.1.3 Job Configurations

The code is not designed to have a process span multiple NUMA-nodes. Therefore only up to 14 threads per process are used. The structure of processes, described in section 2.7, has further limitations for the hybrid version. As the world manager is treated equally as other processes by the scheduler, it occupies a whole processor even though it needs only one core. The number of members ($nprocs$) groups can have, has to be a power of 2 , which limits the number of nodes which are reasonable to use (e.g. with $nprocs = 4$, four NUMA-nodes per group are needed for the OpenMP variant. Therefore on even number of nodes only the same number of groups as on one node fewer can be used, as the world manager would occupies a NUMA node). The 14 cores per processor also impacts the ability to compare the hybrid version with the pure MPI version, as multiplying $nprocs$ by 14 does not result in a power of two. Additionally the best performing configurations of both versions, neither have the same number of groups, nor always have a similar number of cores per group. The configurations closest to 14 threads have 8 or 16 times more processes per group (the closest possible values to 14).

4.2 Scaling of implemented parallelizations

4.2.1 Hierarchization and Dehierarchization

The test case with the high minimum level used has 4 dimensions with a minimum level of 5 and a maximum of 8 which results in 34 component grids. The test case without a minimum level uses 5 dimensions with a maximum level of 11 and has 2750 component grids. The tests were performed on one node and on three nodes with one and five groups respectively, such that each group gets assigned to one processor (i.e. one NUMA-node).

Figure 4.1 shows the measured times. A speedup of 10.1 was reached for one group and 8.6 was reached for five groups on the test case with minimum level. Without minimum level a speedup of 9.36 for one group and for 9.16 five groups was reached. The lower speed up is due to the short execution time of the parallel regions. The time per task is only between $200\mu s$ and $1.5ms$, which means that the parallel regions are only $60\mu s$ to $300\mu s$ long.

4.2.2 Copying

Addition and Extraction from the sparse grid and copying into and from the allreduction buffer and zeroing the subspaces requires almost no computational effort. Those

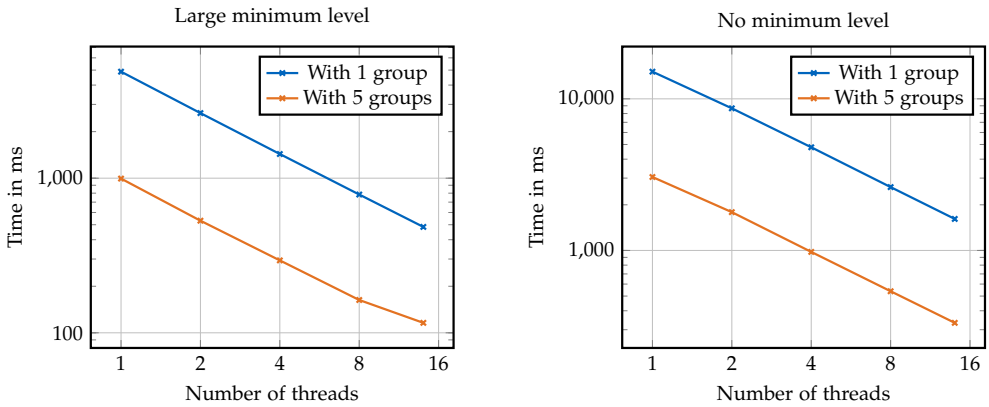


Figure 4.1: Execution times of the hierachization with different number of threads (1,2,4,8,14)

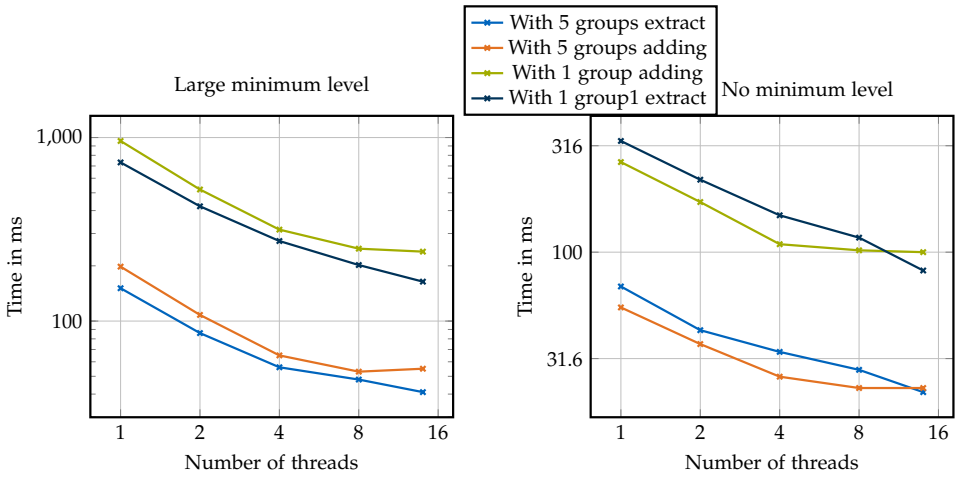


Figure 4.2: Threading speedup for addition and extraction from sparse grid.

functions are therefore strongly memory bound. The parallel speedup for those is relatively small. Figure 4.2 shows the measured time for addition to and extraction from the sparse grid.

Those functions are memory bound and therefore the parallel efficiency is low. Figure 4.3 shows the memory bandwidth usage during one combination step (Although the OpenMP regions at the top are probably misaligned by about 50*ms* to the left by vtune). The maximum bandwidth of the used hardware is 55GB/s per CPU-package (i.e. per NUMA-node and for this configuration per process. In this example copying into the allreduction buffer averages at 48GB/s, while copying out of it averages at 50GB/s. Zeroing the subspaces also averages at 48GB/s. Each fullgrid is extracted in a separate OpenMP region and the measurement was done over all instances (including the OpenMP overhead) which leads to a lower bandwidth usage of 44GB/s. Here it can also be seen that the percentage of read data is higher, as the assignment list has also to be read. Additions to the sparse grid are done directly after the hierarchization of the corresponding fullgrid which means that no accurate data can be provided as those regions are to short to have a meaningful amount of measurements. Extraction from sparse grid is done directly before dehierarchizing the corresponding grid again, as this yields better performance.
For copying into and from the reduction buffer similar results were found. As these where not manually parallelized different scheduling strategies were tried. For copying into the buffer a dynamic scheduling with a block size of 16 was found to be optimal, although not being significantly faster than smaller block sizes. Zeroing out the subspaces performs best with a static scheduling with a block size of 4. Iterating from the largest to the smallest subspaces, increases the execution time slightly and is therefore not used.

4.2.3 AllReduction

Here the tests were performed on three and on ten nodes. For three nodes a 4 dimensional grid with minimum level of 5 and maximum level of 10 was used . The sparse grid contains 615 157 761 doubles (4.58GB) which need to be summed up. On ten nodes the used example has 5-dimensions and minimum and maximum level of 4 and 8. This results in a sparse grid of 403 891 457 doubles (3.01GB), when using two processes per group, one half has to reduce 209 126 529 doubles (1.56GB) the other 194 764 928 doubles (1.45GB). In the measured time very little synchronization overhead is expected as the buffer sizes were exchanged via a collective directly beforehand.

Multi-threaded Allreduction: As support for the thread split model is not available everywhere, runs with the default and the special MPI configurations were made.

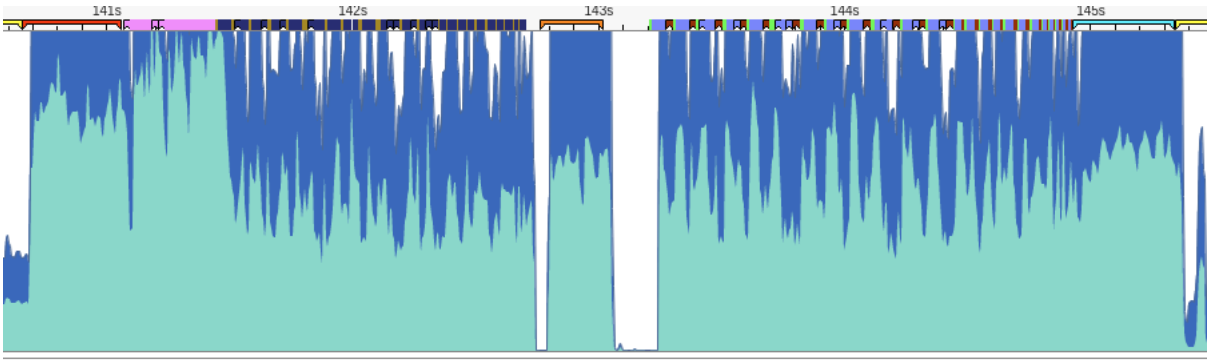


Figure 4.3: Memory bandwidth usage of a test with a high minimum level from the end of of a global reduction to the start of the next one. On the bottom the bandwidth usage on a scale to the maximum of 55GB/s is shown. Light blue is read Data and dark blue stands for written data.
On the top the OpenMP regions are marked. From left to right; Yellow: multi-threaded Allreduction, red: copy buffer into the sparse grid,pink (multiple regions): extract from sparse grid, darkblue and brown: dehierarchization, orange: zeroing subspaces, lightblue and green: hierarchization, turquoise:copy sparse grid into allreduction buffer.

Figure 4.4 shows the measured times. On three nodes the a speedup of 24% is reached when using 14 threads instead of only one. For the example on ten nodes the time was reduced by 21% with 19 groups and by 33% with 9 groups. The current version of the thread split model only works when either one are all threads are used. Therefore it was only tested with 14 and with one thread. Here can be seen that simply using the release_mt variant reduces the execution time slightly. With 14 threads a time of 3.4*s* was reached which is 40% faster than with one thread, 25% faster with out support for multi-threaded *MPI_Allreduce* and 43% faster than without both.

Figure 4.5 shows the activity of the cores during the allreduction. It looks like during the standard version each thread gets a small time slices during which it can use the MPI-library. When explicitly supported it looks like threads are running more in parallel, with the interruptions being waits on remote data. But it looks like each thread is still doing the same operations and therefore not using its maximum performance. Using different communication schemes could utilize the available bandwidth better and choose more paths which would also increase the number of connections used in the cluster.

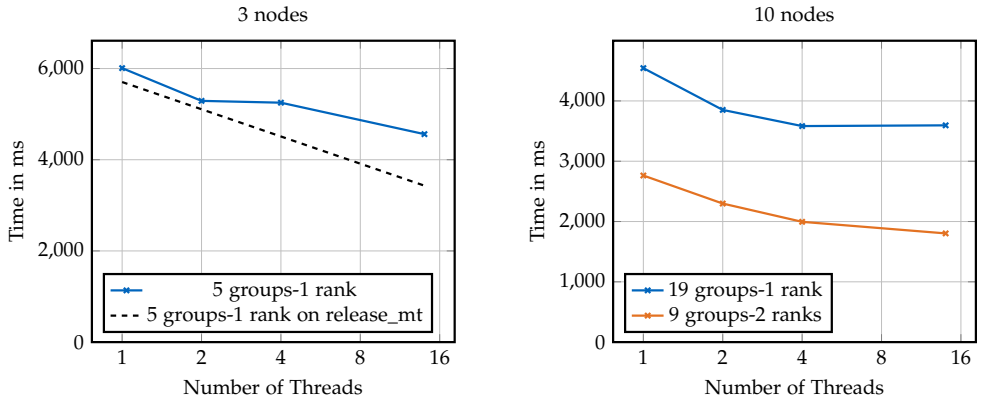


Figure 4.4: Execution time of multi-threaded *MPI_Allreduce* on three nodes on the left and on ten nodes on the right.

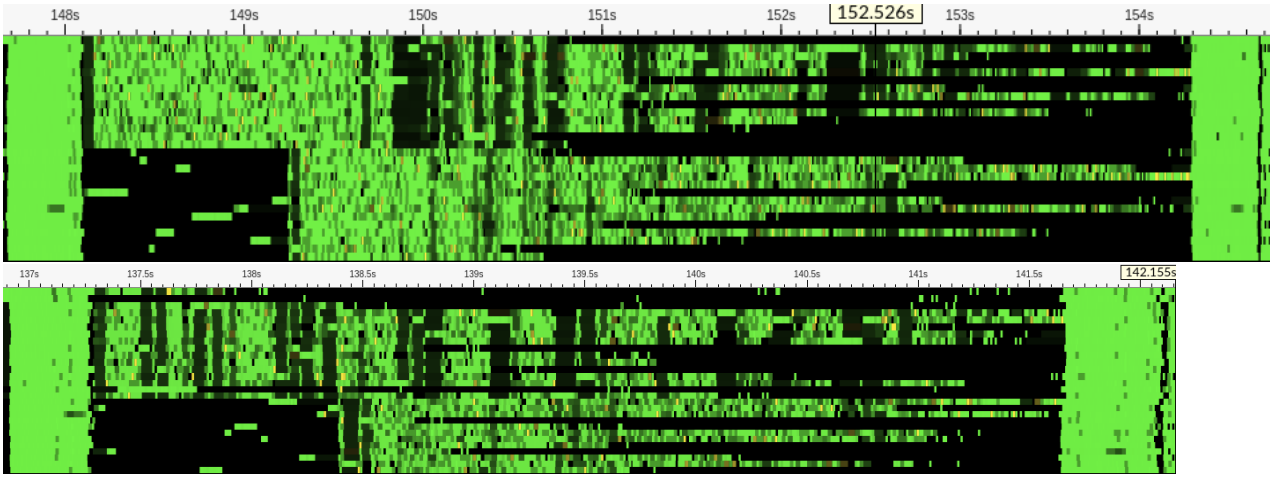


Figure 4.5: Schema of used core during a multi-threaded allreduction on five Nodes. The "MPI_THREAD_SPLIT" model is disabled in the upper image and enabled in the lower one. Green stands for running, yellow fur MPI-busy waiting and Black for inactive. Two processes of a node are depicted. The large green bars at the start and end are from copying into and out of the used buffer.

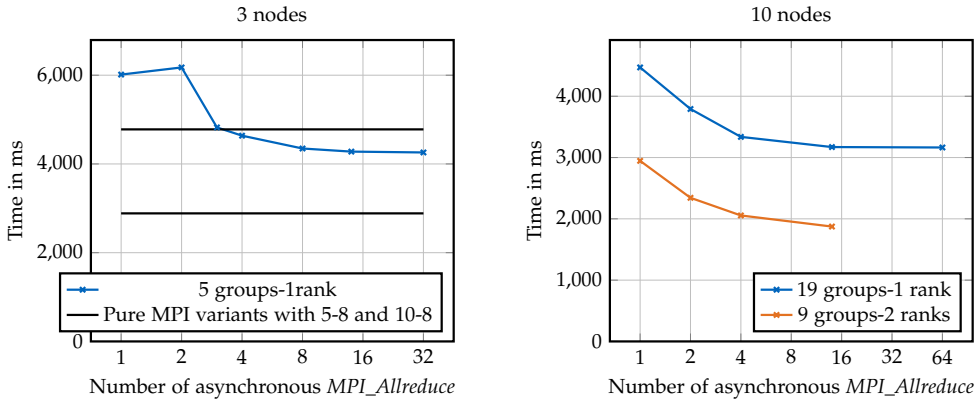


Figure 4.6: Asyncronous allreduction on 3 nodes on the left and 10 on the right

Asynchronous Allreduction: The same input files as for the multi-threaded variant were used. Figure 4.6 shows the measured times. Using 14 asynchronous calls instead of one gives a 29% speedup for the runs with 5 and 19 groups a 36% speedup for the runs with 9 groups. The execution time does not decrease further when using more calls. The MPI implementation of the cluster only uses one thread to handle asynchronous calls. This only allows for limited speedup, by better distributing the computations and bandwidth. In the version of the Intel MPI library optimized for multi-threading (*release_mt*), also used above, the number of threads for asynchronous calls can be increased. This feature has to be enabled by setting the following environment variables *I_MPI_ASYNC_PROGRESS=1* and *I_MPI_ASYNC_PROGRESS_THREADS=num_threads* which would also increase the performance. Enabling this feature on the Linux-cluster causes the program to hang and therefore it could not be tested.

In conclusion using the asynchronous allreduction is faster than the multithreaded variant without enabling non-standard features. With the "MPI_THREAD_SPLIT" model enabled, the multithreaded variant is faster in almost all cases.

4.3 Comparison to the MPI-only version

For strong scaling one test case without minimum level and one with a high minimum level were chosen. The test case without a minimum level has a maximum level of 14 which results in 7280 component grids. The test case with the high minimum level was already used for the global reduction tests (minimum level= 5, maximum level= 10 and 4 dimensions) . Both test cases use about 50GB RAM when using one node. The

test were run on 1, 3, 7 and 13 nodes, which results in 1, 5, 13 and 25 number of for shared-memory usable processors.

4.3.1 Hierarchization

Figure 4.7 shows that the shared-memory variant is significantly (about three times) faster than the MPI-only variant when used on grids without a minimum level. Even though it looks like using less processes per group, would be faster on node, these configurations exceed the available memory and can therefore not be run. As this test does not have a minimum level the distributed grids have very few points per dimension (only one for some dimensions with $nprocs > 1$) which limits the speedup for instances with a high number of processes per group. Therefore the graph is chaotic and the variants with OpenMP are significantly faster. Even though increasing the number of processes per group reduces the execution time, *nprocs* can not be larger than 32 as some processes would have no data points for some grids, which the framework does not allow.

On tests with large minimum levels more points exist per dimension, which causes a better and more predictable speedup as figure 4.8 shows. This reduces the advantages of the shared-memory variant. The OpenMP version with only one thread per group is the fastest one. The other multi-threaded versions are faster than the MPI-only version with 16 times more *nprocs* that uses a few more processes per group and a little bit slower than the MPI-only version with 8 more *nprocs*, which has almost double the number of groups.

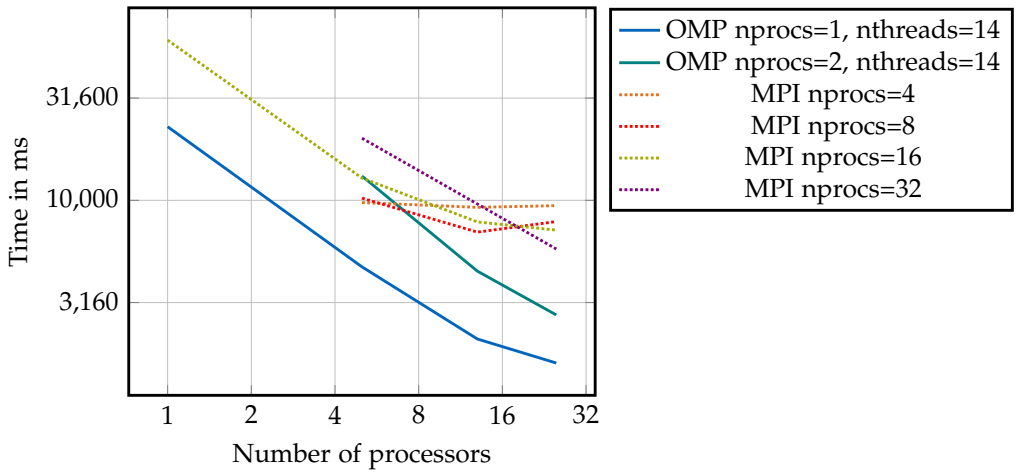


Figure 4.7: Comparision of the performance of the hierarchization without a minimum level, with or without shared memory parallelization

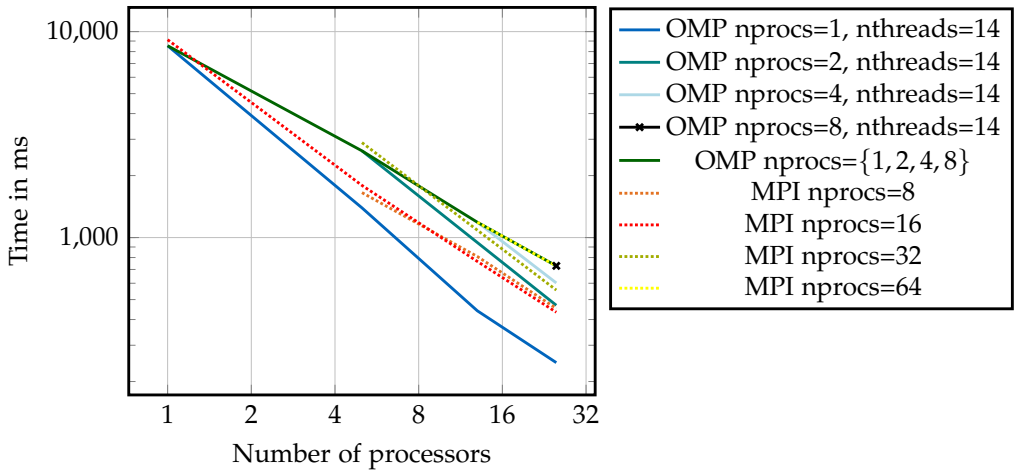


Figure 4.8: Comparision of the performance of the hierarchization with a high minimum level, with or without shared memory parallelization

Figure 4.9 shows the time for extraction from sparse grid without minimum level. Here the pure-MPI cases are up to two times faster. This could be caused by the fact that not all processes on the same core are doing extraction in parallel and dehierarchize instead. The test case does not fit into the memory of one NUMA-node when using a single node, so additional NUMA overhead occurs. With minimum level the addition to the sparse grid takes longer than the extraction. For no minimum level this is the other way round, as the smaller grids are still in the cache after the hierarchization. Figure 4.10 shows the measured times for addition to the sparse grid. Here the graph is also much cleaner. And similar results are achieved.

4.3.2 Global Reduction

Figure 4.11 shows the execution time for the global reduction. For shared-memory cases the "MPI_THREAD_SPLIT" model is enabled. The time is very short on one processor as with only one group no allreduction runs. The execution time does not decrease with more processes as each processes has to do the same auxiliary work and the same amount of data per process has to be summed up over more groups. Here doubling *nprocs* halves the execution time. As for the OpenMP version only smaller number of processes per group make sense (due to both the hierarchization-time and using all nodes), and multi-threaded MPI collectives are not as effective as using more processes, it is slower than the pure-MPI variants with similar number of groups.

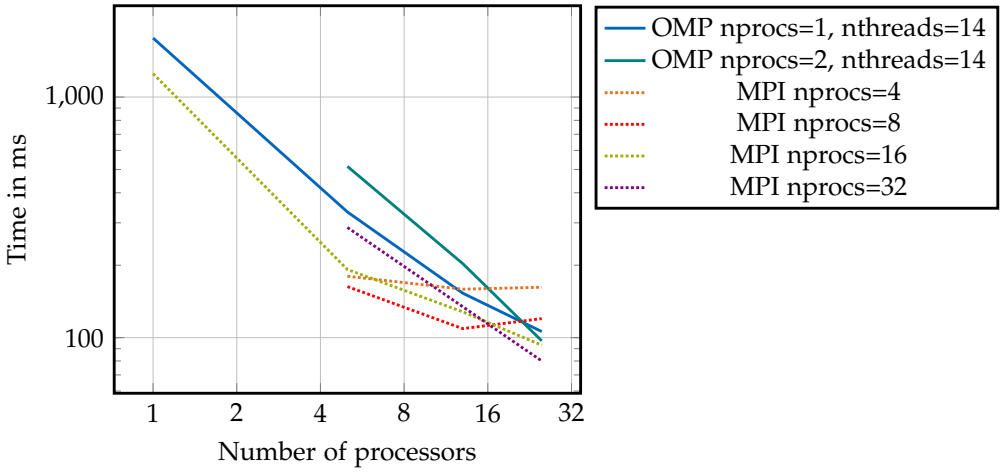


Figure 4.9: Extraction from sparse grid without a minimum level.

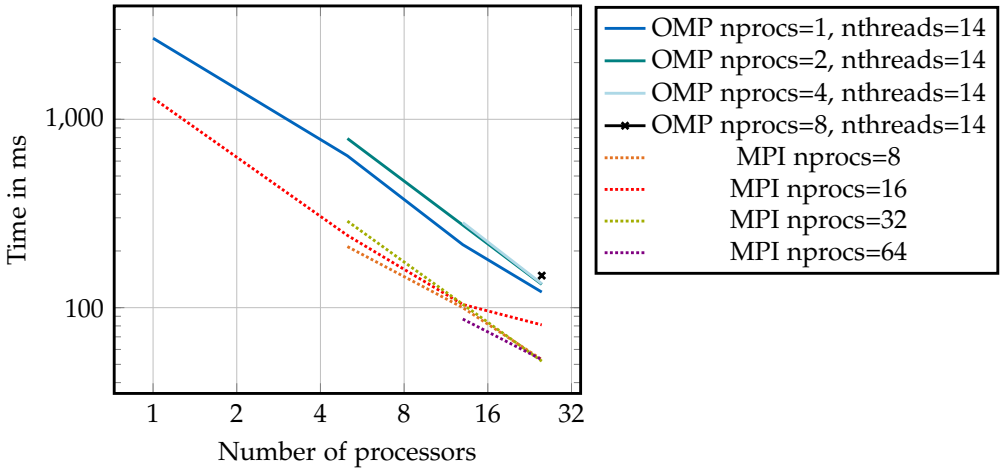


Figure 4.10: Addition to sparse grid with ahigh minimum level .

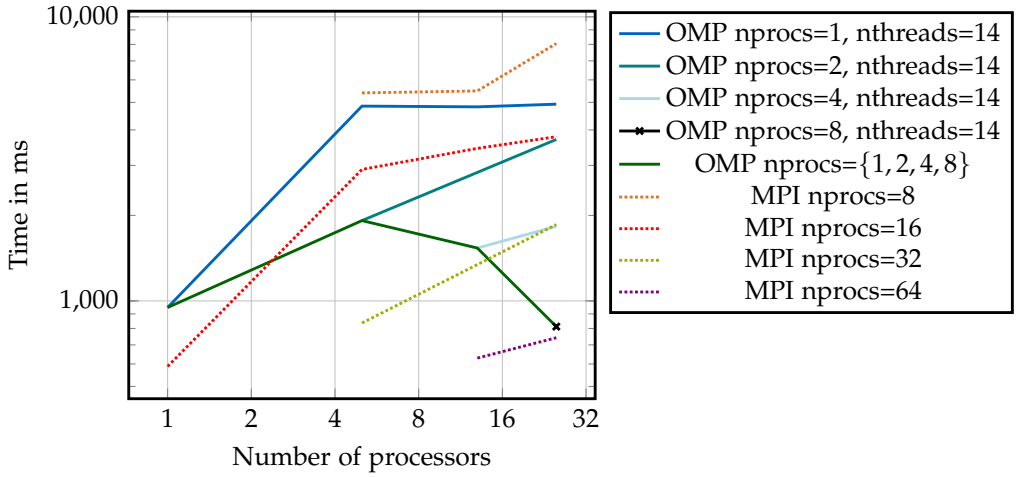


Figure 4.11: Comparision of the performance of the global reduction with high minimum level, with multithread *MPI_Allreduce* with *MPI_THREAD_SPLIT* enabled for shared-memory cases.

4.3.3 Complete Combination Step

Figure 4.12 shows that shared-memory parallelization is significantly faster due to the large speedup in the hierarchization.

Figure 4.13 shows the times for the combination step for the test case with the large minimum level. Here the MPI-only versions are faster. This is the result of more cores being used as Figure 4.14 shows. With 64 processes per group the grid parts become so small that most of grid-parts fully fit in the L3-caches. This makes the hierarchization almost as fast as the fastest shared-memory variant (See Figure 4.8) and therefore in total significantly faster.

The size of the global reduction buffer and the size and number of the fullgrids scale differently. Therefore, direct weak scaling comparisons do not make sense. Instead tests that use most of the available memory were individually constructed for certain number of nodes. For 13 nodes the test case is 5-dimensional, has a minimum level of 3 and a maximum level of 11. Figure 4.15 shows the total execution time of the combination step and the contribution of individual parts to it. Here the best version is the hybrid version with 25 groups, even though it has the largest global reduction time (the buffer has a size of 8GB). It can be seen that hierarchization and dehierarchization are faster for the OpenMP version compared to the pure MPI versions with similar number of groups. The global reduction is problematic for the shared-memory version. Those effects can best be seen on the runs with 5 groups (MPI) and 6 groups (OpenMP).

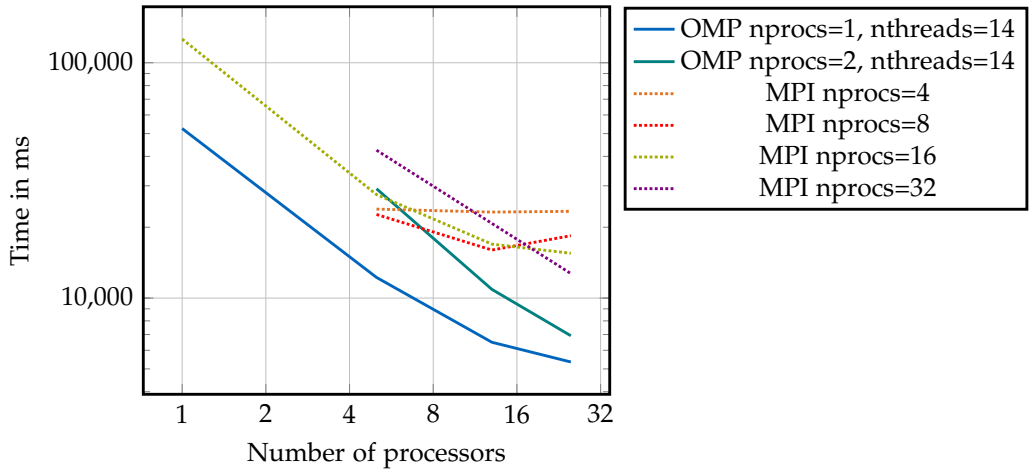


Figure 4.12: Comparision of performance the combination without minimum level, with or without shared memory parallelization

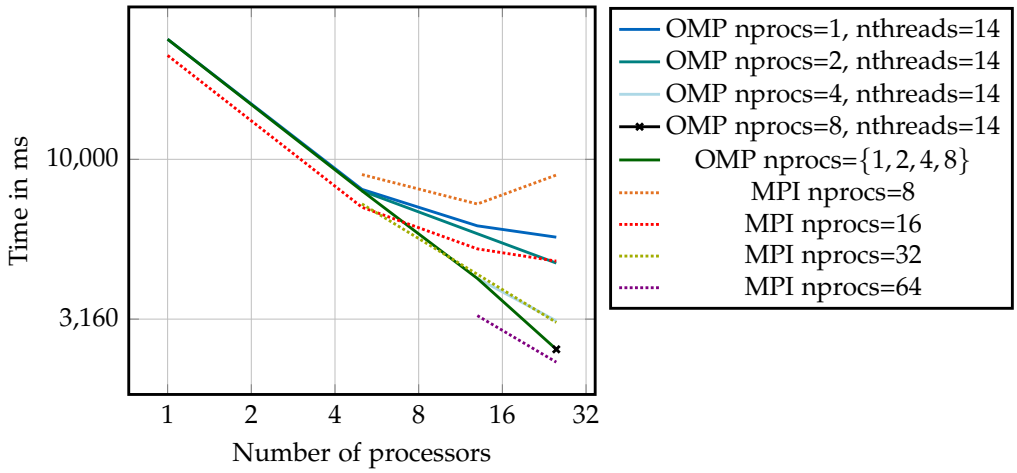


Figure 4.13: Comparision of performance the combination with high minimum level, with or without shared memory parallelization

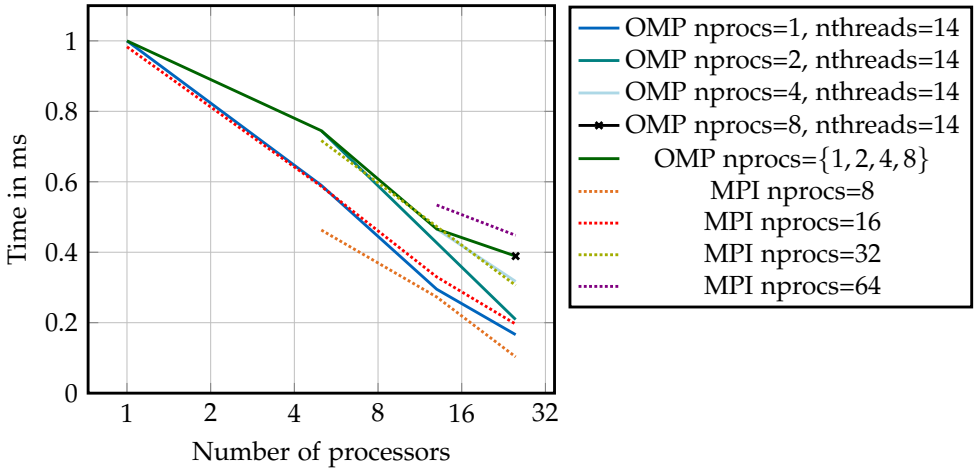


Figure 4.14: Comparision of the parallel efficency of the combination with high minimum level. The test with one group and 14 threads was chosen as a baseline

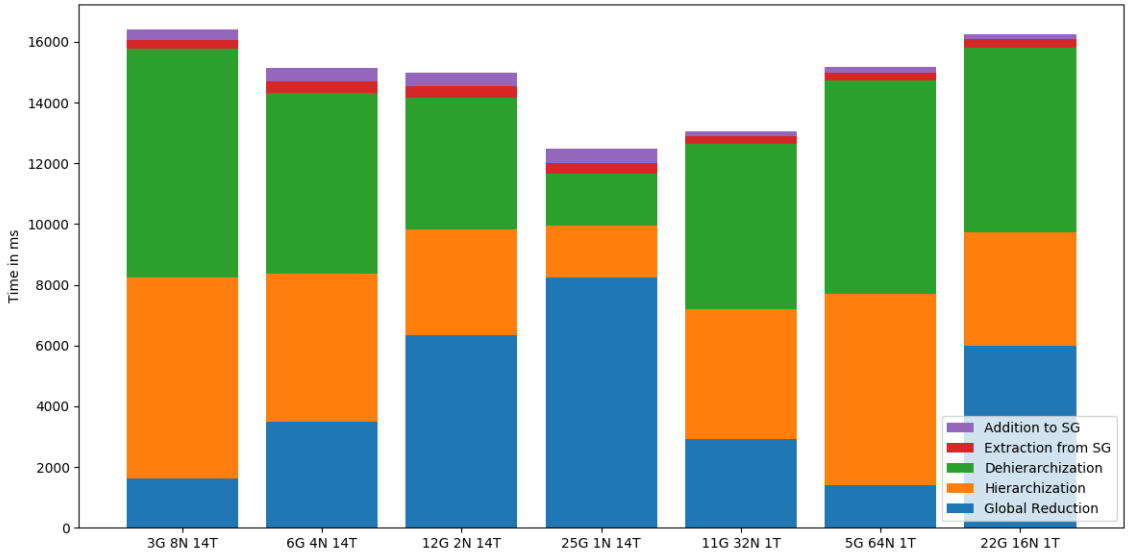


Figure 4.15: Time slice of different operations on 13 nodes with different number of groups (G), nprocs(N) and threads(T)

The hierarchization and dehierarchization are faster, while the global reduction takes more than twice as long. This results in the shared-memory version being about 700ms (5%) faster.

OpenMP overhead/ general improvements vs before

It is also important that the changes for the shared-memory parallelization did not reduce the initial performance. The time for the combination step increased by 50ms from 8.71s to 8.76s, which is less than 1% worse. The total time changed by the same amount even though ten combination steps were performed. The only significant difference is during the dehierarchization which gets about 160ms worse. This is caused by processes waiting on other processes in the group to finish the global reduction.

5 Conclusion and Future Work

The shared-memory parallelization provides significant performance improvements when choosing a low minimum level. When using larger minimum levels the performance of the global reduction diminishes those. This may perform differently on other clusters and may improve overtime with better support for hybrid OpenMP and MPI. More important is that the shared-memory parallelization of the combination step allows for already hybrid algorithms to be efficiently used and gaining speedup when adding shared-memory parallelization to currently used applications. A time step does usually take about one order of magnitude longer than a combination step, and usually several time steps per combination step are executed. Therefore, configurations where the hybrid version of the combination step takes longer than the pure-MPI-variant may be overall the fastest. Due to the large duration difference between the two steps, a $10\% - 20\%$ slower combination step will barely affect the speedup of the hybridization of the time stepping algorithm. Another advantage is that algorithms which are only shared-memory parallelized can be used effectively on multiple (relatively few) nodes without requiring parallelization with MPI.

The distributed combigrid module is designed for exascale applications. Therefore to fully analyze the performance of the shared-memory parallelization it has to be run on significantly more nodes (i. e. a few thousand nodes). This could not be done in this thesis as I only had access to the mpp2-Cluster.
Determining the optimal number of members per group is difficult to predict. An auto-tuning approach that would try different configurations could be considered. It would either require large parts of the code base to be rewritten or the program would have to be restarted for every different configuration.

List of Figures

Bibliography

[BG04] H.-J. Bungartz and M. Griebel. “Sparse grids.” In: *Acta numerica* 13 (2004), pp. 147–269.

[GG10] T. Gerstner and M. Griebel. “Sparse Grids.” In: *Encyclopedia of Quantitative Finance*. Ed. by R. Cont. John Wiley and Sons, 2010.

[GSZ90] M. Griebel, M. Schneider, and C. Zenger. “A combination technique for the solution of sparse grid problems.” In: (1990).

[Hee18] M. Heene. “A massively parallel combination technique for the solution of high-dimensional PDEs.” In: (2018).

[MK19] R. K. Malladi and D. A. S. Kapoor. “Boost Performance for Hybrid Applications with Multiple Endpoints in Intel® MPI Library.” In: *Intel Parallel Universe Magazine* 39 (2019).

[MPI15] MPI-Forum. *MPI: A Message-Passing Interface Standard Version 3.1*. 2015.

[OB19] M. Obersteiner and H.-J. Bungartz. “A Spatially Adaptive Sparse Grid Combination Technique for Numerical Quadrature.” en. In: *Sparse Grids and Applications - Munich 2018*. Springer Verlag, Jan. 2019.

[Pfl10] D. M. Pflüger. “Spatially Adaptive Sparse Grids for High-Dimensional Problems.” Dissertation. München: Technische Universität München, 2010.