Computational Science and Engineering
(International Master´s Program)

Technische Universität München

Master´s Thesis

Towards Non-blocking Combination Schemes in the Sparse Grid Combination Technique

Shreyas Shenoy

Computational Science and Engineering
(International Master's Program)

Technische Universität München

Master's Thesis

Towards Non-blocking Combination Schemes in the Sparse Grid Combination Technique

Author:	Shreyas Shenoy
Advisor:	Michael Obersteiner
Supervisor:	Univ.-Prof. Dr. Hans-Joachim Bungartz
Submission Date:	Feb 20th, 2019

I hereby declare that this thesis is entirely the result of my own work except where otherwise indicated. I have only used the resources given in the list of references.

Feb 20th, 2019 Shreyas Shenoy

Acknowledgments

I would like to thank my parents for all the support and love that they have rendered over the years.
I wish to express my sincere gratitude to my advisor Mr. Michael Obersteiner, for his immense patience, knowledge and invaluable guidance that helped me at all times while researching and writing this thesis.

"To achieve great things, two things are needed; a plan, and not quite enough time."

-Leonard Bernstein

Abstract

Despite the utilization of High Performance Computing (HPC) to solve high-dimensional partial differential equations (PDEs), the extent to which these problems are solvable in a considerable amount of time, is still restricted by the so called curse of dimensionality. One promising method that strives to mitigate this issue is the sparse grid combination technique. A sparse grid is a suitable approximation of the regular grid, capable of being decomposed further into different coarse and anisotropic computational grids of lower resolutions, called component grids. This enables dual levels of parallelism, as the component grids can be computed in parallel, completely independent from each other. Subsequently the computation on each component grid can also be subject to parallelism. However, certain time-dependent problems, in order to fulfill their stability and convergence criteria, necessitate the combination of component grid solution into sparse grid at regular intervals. This in turn, reintroduces the need for global synchronization and communication. Limiting the extent of scalability of this technique.

In this work, we introduce non-blocking/asynchronous combination techniques, where the combination step is also carried out in parallel with other computational steps, thus striving to eliminate the time lost, during global communication and synchronization. The credibility of this technique has been tested out on the Advection-Diffusion problem and GENE, a gyrokinetic simulation of plasma microturbulence in fusion devise. The ensuing speedup was found to be 5 times of those achieved using normal combination technique.

Contents

1. Introduction

Solving high dimensional partial differential equations (PDEs) has been one of the grand challenges of high-performance computing (HPC) [17]. The challenges arise from two major attribute of the problem being solved, firstly its computational complexity, and the communication complexity. Both the attributes are strongly intertwined with the nature of the problem and the system that is used for problem solving

The computational complexity, rears it head majorly with the intervention by the curse of dimensionality i.e exponential growth of the number of unknowns with the number of dimensions, which instills restrictions on how fine the simulation domain of the required problem could get. Furthermore it is required that the numerical algorithms that are used to solve the problems are capable of efficiently exploiting the massively parallel nature of HPCs. The problem is then made two fold by the communication complexity, raising from the communication/synchronization requirement between individual processes. This communication/synchronization bottleneck are already known to limit the scalability of the parallel scientific applications on today's petascale systems [4] [5] [6].

One promising approach to solve the high-dimensional PDEs is the sparse grid combination technique. This is a numerical scheme where a problem are projected on different coarse and anisotropic grids called component grids, which combine to form a sparse grid of much higher target resolution, while closely following the solution on a regular grid. As a result, the curse of dimensionality can be mitigated as high accuracy can be reached with significantly less computing resources.

The combination technique, offers two levels of parallelism, as the computation on each component grid can be carried out asynchronously and independently of each other, while the computation on component grid can also be subject to parallelism. Although there are PDEs problems, which require the combination to only be carried out once at the end. Certain problems impose the restriction of carrying out the combination step at regular intervals in order to satisfy the stability and convergence criteria. This leads to the reintroduction of communication/synchronization bottleneck. Since, the computational steps on all component grids come to a standstill, waiting for all the processes to have finished communicating their respective component grid values, combined them and updated the component grids with the new combined value.

Implementing an Asynchronous combination technique, where the combination step occurs in tandem with the computational step forms the main crux of this master thesis. Furthermore, the accuracy and limitations of this method compared to a normal combination technique, based on the results obtained from carrying out tests on Advection-Diffusion and GENE, a gyrokinetic simulation of plasma microturbulence in fusion devise, are also discussed.

The outline of this work is as follows. Chapter 2, provides an introduction to the sparse grid combination technique, MPI: a communication protocol used by HPCs; and also into SG++ framework: An implementation of parallel combination technique, to solve time-dependent and high-dimensional PDEs on a full supercomputer. Chapter 3 forms the main core of the thesis, where the different algorithms for asynchronous combination technique is discussed, along with the mathematical basis that justifies the approaches used. It also covers the methodology used to test the aforementioned algorithm. In chapter 4 the results of the tests carried out on Advection-Diffusion problem and GENE are covered. It also covers the advantages and also the limitations of asynchronous techniques that were uncovered during the testing process.

2. Theoretical Background

This chapter provides an comprehensive overview of relevant topics necessary for understanding the algorithm. It begins with the explanation of combination technique for sparse grid - the mathematical base for the entire project. This is followed by an overview of MPI, a communication protocol for programming parallel computers. Finally, an overview of SG++ framework is provided, which incorporates the aforementioned topics with emphasis on its combination step.

2.1. Combination Technique

Consider an anisotropic grid $\Omega_{\vec{l}}$ on bounded regular domain Ω, with mesh size $h_{\vec{l}} := (h_{l_1}, \dots, h_{l_d}) = 2^{-\vec{l}}$ in each coordinate direction $t, t = 1, \dots, d$. where $\vec{l} = (l_1, \dots, l_d) \in \mathbb{N}^d$ denotes a multi-index level, i.e the discretization resolution. The gridpoints are then represented by,

$$x_{\vec{l},\vec{j}} := (x_{l_1,j_1}, \dots, x_{l_d,j_d}) \tag{2.1}$$

where, $x_{l_t,j_t} := j_t \cdot h_{l_t}$ and $j_t = 0, \dots, 2^{l_t}$, representing the spatial position of the gridpoint. The associated space $V_{\vec{l}}$ of d-linear functions for such a grid is given by

$$V_{\vec{l}} := span\{\phi_{\vec{l},\vec{j}} | \leq j \geq 2^{\vec{l}}\} \tag{2.2}$$

spanned by the usual basis of d-dimensional piecewise d-linear hat functions.

$$\phi_{\vec{l},\vec{j}}(\vec{x}) := \prod_{t=1}^{d} \phi_{l_t,j_t}(x_t) \tag{2.3}$$

where the nodal basis function is defined by

$$\phi_{l,j}(x) = \begin{cases} 1 - |\frac{x}{h_l} - j| & , x \in [(j-1)h_l, (j+1)h_l] \cap [0,1] \\ 0 & , otherwise \end{cases} \tag{2.4}$$

2.1.1. Hierarchical subspace-splitting

Using an index set $B_{\vec{l}}$ defined by,

$$B_{\vec{l}} := \left\{ \vec{j} \in \mathbb{N}^d \middle| \begin{array}{ll} J_t = 1, \dots, 2^{l_t} - 1, j_t \text{odd}, & t = 1, \dots, d, \text{if } l_t > 0 \\ J_t = 0, 1 & t = 1, \dots, d, \text{if } l_t = 0 \end{array} \right\} \tag{2.5}$$

It is possible to construct hierarchical difference subspaces, $W_{\vec{l}}$ such that,

$$W_{\vec{l}} = \text{span}\{\phi_{\vec{l},\vec{j}} | \vec{j} \in B_{\vec{l}}\} \tag{2.6}$$

which allows for definition of multilevel sub-space decomposition. Further we can write V_n as direct sum of subspaces.

$$V_n := \bigoplus_{|\vec{l}|_\infty \leq n} W_{\vec{l}} \tag{2.7}$$

The basis function $\phi_{\vec{l},\vec{j}}$ which span $W_{\vec{l}}$ are disjunct for $\vec{l} > 0$ while the family of such functions are just hierarchical basis of V_n [7] [24] [10]. An example of construction of grid space V_n using $W_{\vec{l}}$ can be found in the following figure 2.1.

We can finally, define each function $f \in V_n$ as follows,

$$f(x) = \sum_{|\vec{l}|_\infty \leq n} \sum_{\vec{j} \in B_{\vec{l}}} \alpha_{\vec{l},\vec{j}} \cdot \phi_{\vec{l},\vec{j}}(x) = \sum_{|\vec{l}|_\infty \leq n} f_{\vec{l}}(\vec{x}) \text{ with } f_{\vec{l}} \in W_{\vec{l}} \tag{2.8}$$

where $\alpha_{\vec{l},\vec{j}} \in \mathbb{R}$ are the coefficients of the representation of hierarchical tensor product basis and $f_{\vec{l}}$ denotes the hierarchical component functions. It is to be noticed that direct use of such a complete grid is periled with the curse of dimensionality, as the number of basis functions which describe a $f \in V_n$ in nodal hierarchical basis is $(2^n + 1)^d$ [9].

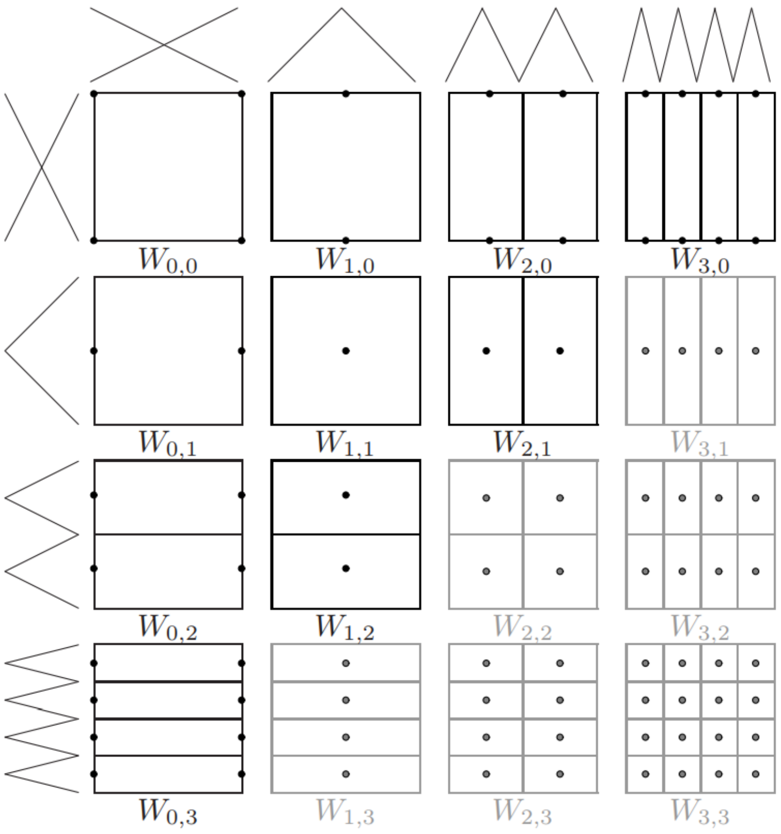


Figure 2.1.: The basis functions of the hierarchical subspaces W_l of the space V_3. The sparse grid space V_3^s contains the upper triangle of spaces shown in black

2.1.2. Sparse Grids

To overcome the aforementioned shortcomings of a complete grid, the concept of sparse grid is introduced [25] [13]. Here, hierarchical basis functions with a small support, and correspondingly small contributions are truncated off.

Formally, the sparse grid function space $V_n^s \subset V_n$ is defined as

$$V_n^s := \bigoplus_{|\vec{l}| \leq n+d-1} W_{\vec{l}} \tag{2.9}$$

Which is graphically visualized in the figure 2.1, where the conserved subspaces $W_{\vec{l}}$ are displayed in black in comparison to the gray ones which are omitted.

The function representation gets modified accordingly as follows

$$f(x) = \sum_{|\vec{l}|_1 \leq n+d-1} \sum_{\vec{j} \in B_{\vec{l}}} \alpha_{\vec{l},\vec{j}} \cdot \phi_{\vec{l},\vec{j}}(x) = \sum_{|\vec{l}|_1 \leq n+d-1} f_{\vec{l}}(\vec{x}) \text{ with } f_{\vec{l}} \in W_{\vec{l}} \tag{2.10}$$

The advantage of such a representation is evident from the comparison of degree of freedom, which for a sparse grid is of the order $\mathcal{O}(2^n n^{d-1})$ to the $\mathcal{O}(2^{nd})$ of the full grid space; and the approximation accuracy, which is of the order $\mathcal{O}(h_n^2 \cdot n^{d-1})$ to $\mathcal{O}(h_n^2)$ of its full grid counterpart [3]. Clearly, the decrease in degree of freedom has only a slight deterioration the accuracy, which serves as a crucial advantage for sparse grid, helping it overcome the curse of dimensionality.

It is to be noted, the sparse grids that were constricted were from priori selection of grid-points which is only optimal if a certain smoothness conditions are met, i.e, if the function has a bounded second mixed derivatives [11]. In case, such conditions are not met, adaptive refinement strategies should be employed [8] [22] [21]. Further, the sparse grid functions also does not possess some of the properties that a full grid functions have, for instance a sparse grid function need not be monotone[22] [18].

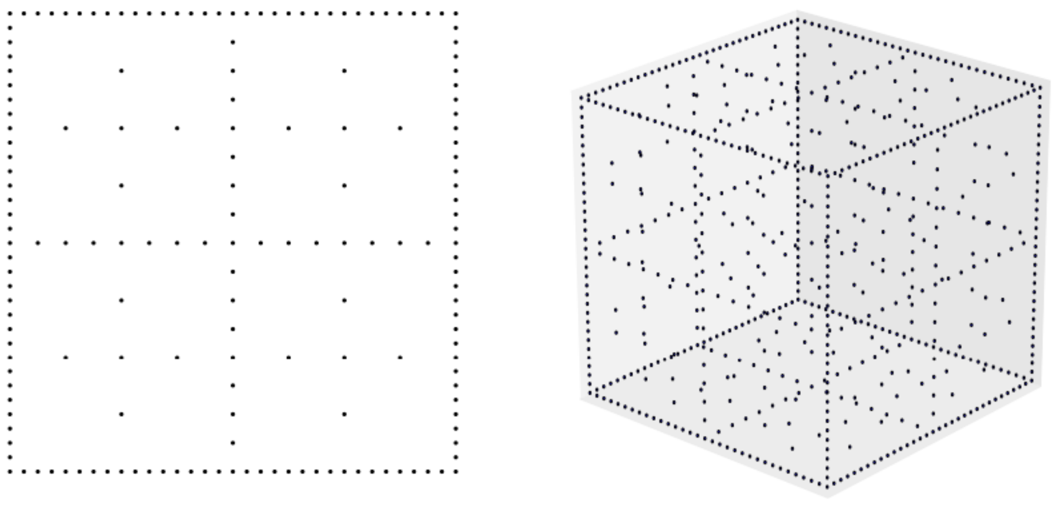


Figure 2.2.: Two-dimensional sparse grid (left) and three-dimensional sparse grid (right) of level $n = 5$

2.1.3. Sparse grid combination technique

Apart, from the drawbacks mentioned in the previous section, representation of sparse grid in directly in terms of hierarchical basis has some numerical disadvantages, for example in cases where the stiffness matrix is not sparse making the efficient computation of matrix-vector-product challenging in implementation. This is overcome by the so called combination technique [14], which is based on multi-variate extrapolation [1]. Here, the function is discretized on a certain sequence of grids using a nodal discretization. A linear combination of these partial functions then gives the sparse grid representation.

The function f is discretized on a certain sequence of anisotropic grids $\Omega_{\vec{l}} = \Omega_{l_1,\dots,1_d}$ with uniform mesh sizes $h_t = 2^{-l_t}$ in the t-th coordinate direction. The following grids $\Omega_{\vec{l}}$ are considered.

$$|\vec{l}|_1 := l_1 + \ldots + l_d = n + (d-1) - q, q = 0, \ldots d-1, l_t > 0 \tag{2.11}$$

Incorporating a finite element approach with piecewise d-linear functions $\phi_{\vec{l},\vec{j}}(\vec{x})$ on each grid $\Omega_{\vec{l}}$ gives us the representation in the nodal basis as follows.

$$f_{\vec{l}}(\vec{x}) = \sum_{j_1=0}^{2^{l_1}} \ldots \sum_{j_d=0}^{2^{l_d}} \alpha_{\vec{l},\vec{j}} \phi_{\vec{l},\vec{j}}(\vec{x}) \tag{2.12}$$

linearly combining the discrete partial functions $f_{\vec{l}}(\vec{x})$ from the different grids $\Omega_{\vec{l}}$ according to the previous combination formula.

$$f_n^c(\vec{x}) := \sum_{q=0}^{d-1} (-1)^q \binom{d-1}{q} \sum_{|\vec{l}|_1 = n+d-1-q} f_{\vec{l}}(\vec{x}) \tag{2.13}$$

A general combination scheme can be written as,

$$f_{\mathcal{I}}^c(\vec{x}) := \sum_{\vec{l} \in \mathcal{I}} c_{\vec{l}} f_{\vec{l}}(\vec{x}) \tag{2.14}$$

where $\mathcal{I}$ is the index set that specifies the set of level vectors corresponding to the component grids used in the scheme The solution obtained with the combination technique f_n^c for the numerical treatment of partial differentiation equation is in general not the sparse grid solution f_n^s [14]. However, the approximation property is of the same order, provided a certain series expansion of the error exists, as shown for model-problems in [2]. Also, since there are grid points which occur multiple times in different component grids, the total number of grid points being computed is higher for the combination technique than on the corresponding sparse grid. There are $\mathcal{O}(d \log(h^{-1})^{d-1})$ component grids with $\mathcal{O}(2^n)$grid points each [17]. However, asymptotically the number of grid points still is significantly lower than on the corresponding full grid.

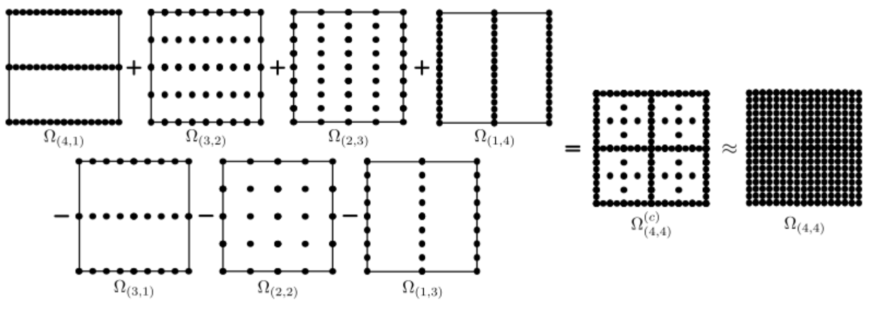


Figure 2.3.: The classical combination technique in two dimensions with n = 4. Seven component grids are combined to obtain a sparse grid approximation (on the grid $\Omega^s_{(4,4)}$) to the full grid solution on the grid $\Omega_{(4,4)}$.

2.1.4. Time integrated combination technique

The temporal discretization for time-dependent problems requires an application of problem specific operator $\mathcal{F}$ which evolves the sparse grid by a certain time step.

$$f^s_{\vec{n},t+\Delta t}(\vec{x}) = \mathcal{F}\Big\{ f^s_{\vec{n},t}(\vec{x}) \Big\} \tag{2.15}$$

For the combination technique, individual operator $\mathcal{F}_{\vec{l}}$ is applied on each individual component grid to evolve them by time step Δt and only then combination is performed

$$f^{c}_{\vec{n},t+\Delta t}(\vec{x}) = \sum_{\vec{l}\in\mathcal{I}} c_{\vec{l}}\mathcal{F}_{\vec{l}}\Big\{\mathcal{P}_{\vec{l}}\{f^{c}_{\vec{n},t}(\vec{x})\}\Big\} \tag{2.16}$$

where the operator $\mathcal{P}_{\vec{l}}$ is the projection of the combined solution at the time t into the approximation space $V_{\vec{l}}$ of the corresponding component grid.

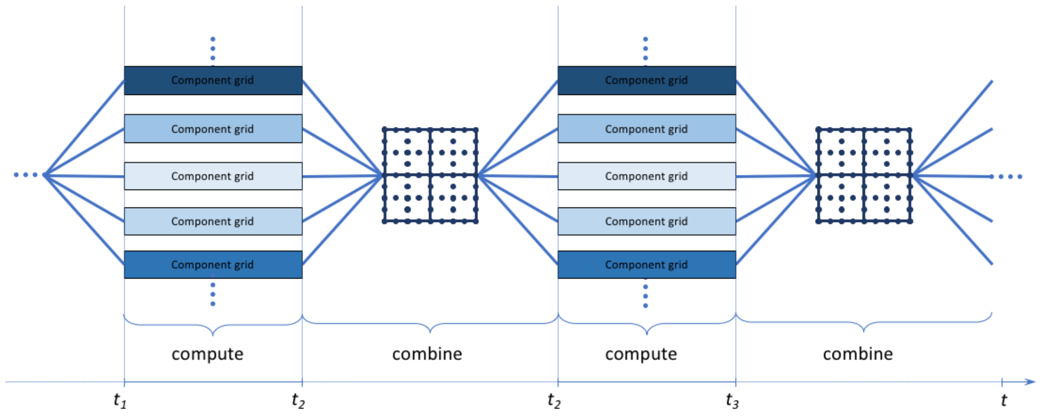


Figure 2.4.: Combination Technique

Then the combined solution is evaluated in the sparse grid approximation space. This combined solution is projected back onto the individual component grids and set as the initial values for the computation of the next time interval. Depending on the problem, only one combination maybe required at the end, for others, it may be required to carry out combinations at regular interval in order to ensure convergence and stability of the obtained solution [17].

Problems of later case unfortunately are riddled with issue of not being sufficiently parallel anymore, as the component grids cannot be computed independently, creating a need for global synchronization at various computational points. Thus, for the large-scale simulations where the simulation data of each component grid is distributed over thousands of processes of an HPC system, an efficient and scalable implementation of the combination step is most crucial for the overall performance [17].

2.2. MPI

Communications in such systems are carried out using Message Passing Interface or MPI. It is a communication protocol for programming parallel computers [16]. Though not sanctioned by any major body, it is considered to be a de-facto standard for communication among processes that model a parallel program that run on distributed memory system. MPI's high performance, scalable and portable nature, makes it a dominant model in high performance computing [23]. Simply put, MPI defines the syntax and semantics of the core library routines

Communication Mode	Needs Matching Receive		Communication Type
	To Start	To Complete	
Standard	No	Maybe	Non-Local
Buffered	No	No	Local
Synchronous	No	Yes	Non-Local
Ready	Yes	Yes	Non-Local

Table 2.1.: MPI Communication Modes

useful for writing message parsing program in C, C++ and Fortran.

MPI predominantly is available in two "flavours": MPICH and Open MPI. The primary difference between both being that MPICH is the high-quality reference implementation of the latest MPI standard [15], while, Open-MPI targets the common case, both in terms of usage and network conduits [12]. Since, MPICH was used in this particular implementation, any reference to MPI's working or syntax may have a bias towards that particular implementation. Further, since it is not possible to incorporate all the features of MPI, only the concepts necessary would be delved here.

2.2.1. Point-to-point communication

Sending and receiving of messages by processes is the basic MPI communication mechanism. All communication in an MPI occurs within a communicator (MPI_Comm) or a group of processes, where each process has a unique identifier. All the processes are contained in the predefined communicator MPI_COMM_WORLD. Each process can determine its rank and the size of the group it belongs to. Before any communication can occur it is required that MPI is initialized using MPI_Init(). Point-to-point communications are the simplest communications that involve just two processes. Any form of communication in MPI follows the primary rule that the messages are non-overtaking, i.e if a sender sends two messages in succession to the same destination and both match the receive then the operation receive the second message if the first one is still pending. This requirement ensures that the message passing is deterministic.

The communications methods act predominantly in two modes, blocking and non-blocking an overview of both can be seen in the following figure, along with a brief explanations of the same.

Blocking Operations

An operation is said to be blocking, if the return from the procedure indicates that the user is allowed to re-use resources specified in the call. In case of MPI_Send(), it simply means that operation is terminated only when the buffer passed to it can be reused, either because the buffer was saved somewhere by MPI or it has been received by the destination. Similarly, for MPI_Receive() the termination is when the receive buffer has been filled with valid data. MPI also offers four distinct communication modes that allows one more finer choice on communication protocol. The four modes being Standard (MPI_Send()), Buffered (MPI_BSend()) Synchronous (MPI_SSend()) and Ready (MPI_RSend()).

Non-blocking Operations

Non-blocking operations are used to improve performance by overlapping communication and computation. Unlike blocking, a nonblocking send start call initiates the send operation, but does not complete it. The send start call returns before the message was copied out of

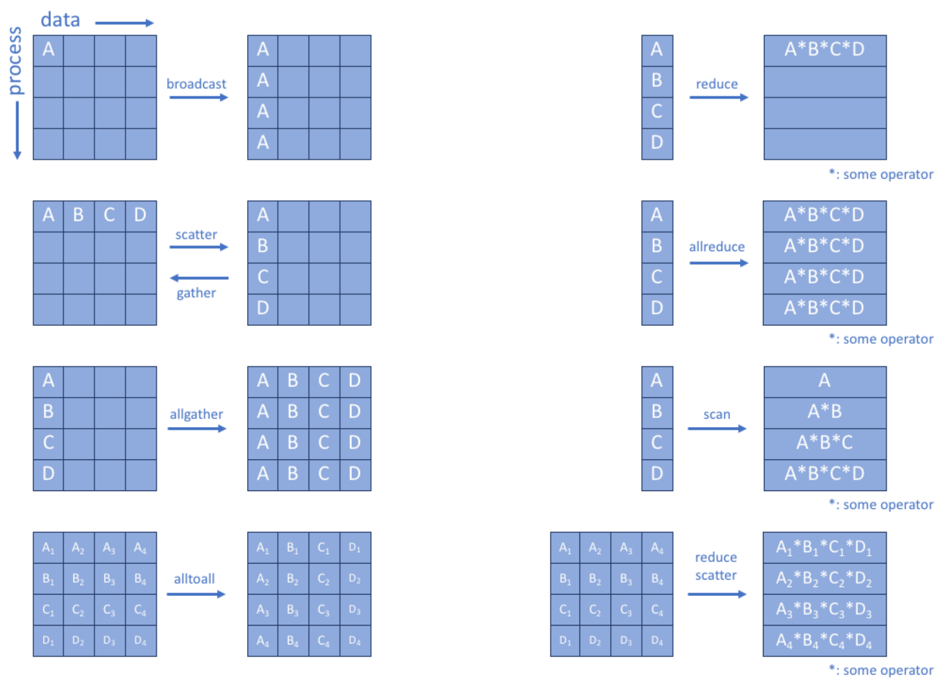


Figure 2.5.: MPI collective operations

the send buffer. The, a separate send complete call is needed to complete the communication, i.e., to verify that the data has been copied out of the send buffer. With suitable hardware, the transfer of data out of the sender memory may proceed concurrently with computations done at the sender after the send was initiated and before it completed. An similar process occurs for non blocking receive. A nonblocking receive may also avoid system buffering and memory-to-memory copying, as information is provided early on the location of the receive buffer. Further, the four mentioned in blocking: standard, buffered, synchronous and ready are also available in the non-blocking format. Improper use of non-blocking operations may lead to issues like deadlock and race conditions and the onus is completely on the programmer to take the necessary precautions to avoid such instances.

2.2.2. Collective Communication

Unlike Point-to-Point a collective communication involves a group of processes. It is executed by having all processes in the group call the communication routine, with matching arguments. One of the key arguments is a communicator that defines the group of participating processes and provides a context for the operation. Various MPI collective operations and their corresponding fucntions are displayed in the following figure

Like point-to-point, collective operations also have blocking and non-blocking counterparts. Also, it is important to know that collective routine calls can (but are not required to) return as soon as their participation in the collective communication is complete. The completion of a call indicates that the caller is now free to access locations in the communication buffer. It does not indicate that other processes in the group have completed or even started the operation (unless otherwise indicated in the description of the operation). Thus, a collective communication call may, or may not, have the effect of synchronizing all calling processes.

2.3. SG++ Framework

SG++ is a universal toolbox for spatially adaptive sparse grid methods and the combination technique [20]. It is a first of its kind implementation of parallel combination technique with scalability for time-dependent and high-dimensional PDEs on a full supercomputer. It works on a Manager-Worker pattern, with MPI messages being used for communication between both.

The manager process coordinates individual steps of the parallel combination algorithm. It is also assigned with the task of assigning component grids to worker process group in order to achieve a good load balance. The manger however, doesn't participate in any computation or simulation data storage steps. This is solely carried out by the process groups.

This abstraction of manager process group from component grids is achieved with the abstraction called Tasks. The tasks apart from consisting of the component grids also provide an interface to the applications. The manager is tasked with assigning these tasks to different process groups. So that the component grids are computed completely independent and asynchronously of each other in their respective process group. The manager uses different control signals to coordinate the different process groups.

Based on the signal received from the manager a process group changes it state accordingly. The finite state machine of the process group is provided in the figure 2.6, along with the explanation of the different states.

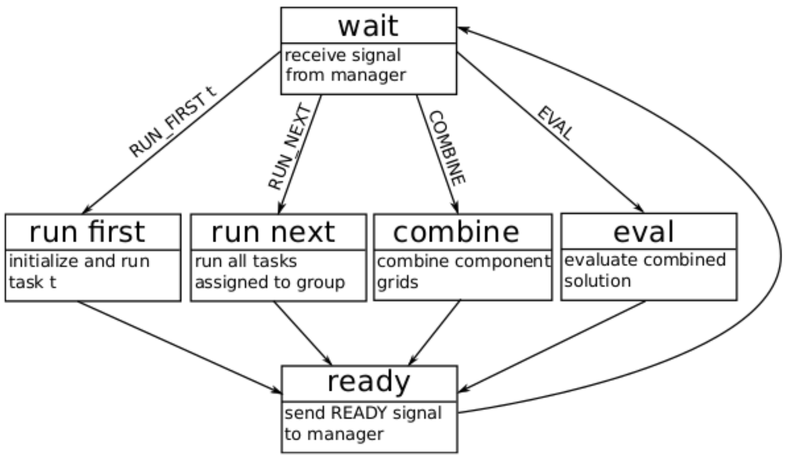

wait
receive signal from manager
RUN_FIRST t
RUN_NEXT
COMBINE
EVAL
run first
initialize and run task t
run next
run all tasks assigned to group
combine
combine component grids
eval
evaluate combined solution
ready
send READY signal to manager

Figure 2.6.: Finite state machine modeling the different modes of operation of a process group.

- *wait*: The process group remains idle till it receives another signal from the manager.
- *run first*: Along with the RUN_FIRST signal process groups also receive their individual task t. Dedicated initialization happens here, after which data structures to store distributed grids are created and simulation data is transferred from the application to component grid.
- *run next*: The process group here, computes all its tasks, one after the other. Simulation data is transferred from component grid to application's data structure before starting a computation of a task; and on completion data is transferred back into the component grid

- *combine*: Combination of the component grid in the sparse grid happens in this state. This happens in three stages. First, all the locally available component grids are combined inside the process group. Then, the combined solution is reduced globally across all process groups. In a last step, the combined solution is transferred back onto the component grid of each task. A more in-depth explanation of this state is given in the next section.
- *eval*: Evaluation of the combined solution by the process group takes places here.
- *ready*: After completion of any of the above state, process groups change to this state, notifying the manger that it is ready to receive new signals.

2.3.1. Parallel combination technique

The implementation of combination technique in SG++ happens as follows. distributed sparse grid is created for each group in parallel, this is followed by hierarchization of indvidual functions. This is followed by sparse grid reduction , which involves creating a temporary buffer, to which all the herarchical subspaces are appended. An MPI all reduce operation is performed, and the indivdual heirarchical subsapces are then refilled with the new combined solution. After which, the control is returned back to the combination procedure which then dehierarchizes all the grids in parallel before returning [20]. The Algorithms of both, Parallel combination and sparse grid reduction procedure are given below:

Algorithm 1 Parallel Combination Technique

```
Input:
    $\mathcal{I}$ : set of component grids
    $c_u$ : combination coefficient corresponding to component grid $u$
    groups : set of process groups
    grids($g$) : set of component grids assigned to group $g$

 1: for all $g \in groups$ do in parallel
 2:     $dsg_g \leftarrow dsg(\vec{n})$                          ▷ create distributed sparse grid
 3:     for $u \in grids(g)$ do
 4:         $u \leftarrow$ hierarchize($u$)                       ▷ hierarchize component grid
 5:         $dsg_g \leftarrow dsg_g + c_u \cdot u$      ▷ add hierarchical coefficients to sparse grid
 6:     end for
 7: end for
 8: $dsg_g \leftarrow \sum_{g \in groups} dsg_g$                  ▷ global reduction of sparse grid
 9: for all $g \in groups$ do in parallel
10:     for $u \in grids(g)$ do
11:         $u \leftarrow dsg_g$                 ▷ extract hierarchical coefficients from sparse grid
12:         $u \leftarrow$ dehierarchize($u$)                   ▷ dehierarchize component grid
13:     end for
14: end for
```

Reduced Combination

SG++ also allows for a special combination technique called *Reduced Combination*. Unlike the conventional combination technique, where the complete sparse grid is constructed, here the user can set limits on the maximum and minimum levels of subspaces to used for creating the distributed sparse grid. Reducing the sparse grid size, leads to faster execution, due to

Algorithm 2 Sparse Grid Reduce

```
1: create buffer $B$
2: for each $W_{\vec{l}} \in V^s_{n-1}$ do
3:     if $W_{\vec{l}} \in V^s_{n-1}$ then
4:         append $W_{\vec{l}}$ to $B$
5:     else
6:         append zeros to $B$
7:     end if
8: end for
9: AllReduce($B.\mathcal{I}$)
10: for each $W_{\vec{l}} \in V_{\vec{l}}$ do
11:     extract $W_{\vec{l}}$ from $B$
12: end for
```

obvious reduction in the amount of data required to be communicated during the reduction operation. More importantly, this also forms the basis for optimization, as the values in the highest subspaces are not shared with any other levels and hence it is redundant to include them in the communication process. Though this method provides for decrease in communication volume, it also causes considerable drop in accuracy.

3. Implementation

In a normal combination technique the component grid evaluation is paused during the communication phase. The evaluation resumes only after the combination is carried out in its entirety and the corresponding combined values are reprojected on the respective grids. The main objective of this thesis was to incorporate the non-blocking statements of MPI, while carrying out combination; which would render a near continuous computation phase, with the communication being carried out in parallel to it.

The following section, dwells into the two non-blocking or asynchronous combination techniques that were implemented over the course of the thesis; their respective modifications to integrate them into the reduced combination scheme; and their eventual assimilation into the problems: Advection-diffusion and GENE.

3.1. Asynchronous Combination Technique

The initial asynchronous algorithm was constructed by splitting the normal parallel combination technique and its embedded sparse grid reduction into two halves. The first half encompassed of initialization steps that prepare the component grid values for global reduction. The other block covered the re-projection of the values onto the component grid after global reduce. The two blocks were distributed over two separate but subsequent time steps, in contrast to a normal combination technique where both the parts transpired over the same time step. The comprehensive set of actions that are performed in the either blocks are as follows:

3.1.1. Initialization Phase

When the process groups are signaled by the manager to enter into asynchronous combination mode, they begin by creating their respective distributed sparse grids in parallel. The component grid values are then stored in-order to be utilized during the value correction step of the extraction phase. This is followed by hierarchization of the component grid values. The hierarchized values are then added to the previously created distributed sparse grids and then the initialization phase of sparse grid reduction is initiated. Here, every process group creates its own local buffer mimicking the complete sparse grid and transfers its contained subspace values into the corresponding location in the local buffer. An *MPI_IAllReduce* is then performed to combine the values spread over the local buffers of the process groups. Since *MPI_IAllReduce* is a non-blocking statement, the control returns to the respective process groups before the global reduction is completed. Each process group then de-hierarchizes the previously hierarchized values and proceeds to the next computation phase.

3.1.2. Extraction Phase

The extraction phase is invoked when process groups enter into combination mode, and the previously initiated *MPI_IAllReduce* has commenced. The extraction part of sparse grid reduce is carried out first. Where the buffers which now contain globally reduced values are transferred to the respective subspaces of the process groups. This completes the sparse grid extraction process and starts the extraction phase of overall asynchronous technique. The subspace values are then de-heirarchized. Unlike normal combination technique, the de-heirarchized

Algorithm 3 Parallel Combination Technique Asynchronous-Initialization phase

```
Input:
    $\mathcal{I}$ : set of component grids
    $c_u$ : combination coefficient corresponding to component grid $u$
    groups : set of process groups
    grids($g$) : set of component grids assigned to group $g$

1:  procedure AsyncCombineInit
2:      for all $g \in groups$ do in parallel
3:          $dsg_g \leftarrow dsg(\vec{n})$                      ▷ create distributed sparse grid
4:          for $u \in grids(g)$ do
5:              $u_{old} \leftarrow u$                       ▷ store the current component grid values
6:              $u \leftarrow$ hierarchize($u$)                    ▷ hierarchize component grid
7:              $dsg_g \leftarrow dsg_g + c_u \cdot u$      ▷ add hierarchical coefficients to sparse grid
8:          end for
9:      end for
10:     for $u \in grids(g)$ do
11:         S parseGridReduceInit()          ▷ global reduction of sparse grid initialization
12:     end for
13:     for all $g \in groups$ do in parallel
14:         for $u \in grids(g)$ do
15:             $u \leftarrow$ dehierarchize($u$)                    ▷ dehierarchize component grid
16:         end for
17:     end for
18: end procedure
```

Algorithm 4 Sparse Grid Reduce Asynchronous - Initialization Phase

```
Input:
    $B$ : local buffer for storing combined values
1:  procedure SparseGridReduceInit                ▷ Initialization of Sparse Grid Reduce
2:      for each $W_{\vec{l}} \in V^s_{n-1}$ do
3:          if $W_{\vec{l}} \in V^s_{n-1}$ then
4:              append $W_{\vec{l}}$ to $B$
5:          else
6:              append zeros to $B$
7:          end if
8:      end for
9:      IAllReduce()
10: end procedure
```

subspace values are not directly transferred to their respective locations in the component grid. As this would render void the updated values in the component grid resulting from the computation steps carried out before the extraction phase initiation. Hence, to compensate for this, globally reduced values transferred into component grids are appended with a *delta* correction: which is given by the difference between values currently held by the component grids and the values that were stored during the previous initialization phase. The proof that this is an adequate compensation can be established as follows:

from the equation 2.1.4, the value of $f_{\vec{l}}$ on each component grid $\Omega_{\vec{l}}$ after a combination at $t + \Delta t$ is given by

$$f_{\vec{l},t+\Delta t} = P_{\vec{l}} \cdot f^c_{\vec{n},t+\Delta t} \tag{3.1}$$

Where $f^c_{\vec{n}}$ is the combined value at the sparse-grid and $P_{\vec{l}} \cdot f^c_{\vec{n},t+\Delta t}$ the projected value on the corresponding grid.
We would like to write the following equation in terms of previous values contained in the sparse grid, as the asynchronous combination is initiated Δt time step before the current time with using the values at that point. Applying Taylor expansion and truncating at the first term we could re-write the previous equation 3.1

$$f_{\vec{l},t+\Delta t} \approx P_{\vec{l}} f^c_{\vec{n},t} + J_{P_{\vec{l}} f^c_{\vec{n},t}} \Delta t \tag{3.2}$$

Where, $J_{P_{\vec{l}} f^c_{\vec{n},t}}$ is the Jacobian matrix defined over the function $P_{\vec{l}} f^c_{\vec{n},t}$.

Since $J_{P_{\vec{l}} f^c_{\vec{n},t}} = P_{\vec{l}} J_{f^c_{\vec{n},t}}$ and assuming, that $J_{f^c_{\vec{n},t}}$ is only influenced by the time aspect, and independent of other parameters like neighboring values, the term $J_{f^c_{\vec{n},t}}$ can be replaced with its one dimensional counter part $f'^c_{\vec{n},t}$
rewriting the equation

$$\begin{aligned} f_{\vec{l},t+\Delta t} &\approx P_{\vec{l}} f^c_{\vec{n},t} + P_{\vec{l}} f'^c_{\vec{n},t} \Delta t \\ &\approx P_{\vec{l}} f^c_{\vec{n},t} + P_{\vec{l}} (f^c_{\vec{n},t+\Delta t} - f^c_{\vec{n},t}) \end{aligned} \tag{3.3}$$

As $f' \Delta t = f_{t+\Delta t} - f_t$
Further it can be seen that the last two terms are just the values of $f_{\vec{l}}$ at t and $t + \Delta t$ respectively, hence

$$\begin{aligned} f_{\vec{l},t+\Delta t} &= P_{\vec{l}} f^c_{\vec{n},t} + f_{\vec{l},t+\Delta t} - f_{\vec{l},t} \\ &= P_{\vec{l}} f^c_{\vec{n},t} + \mathcal{F}_{\vec{l}}(f_{\vec{l},t}) - f_{\vec{l},t} \\ &= P_{\vec{l}} f^c_{\vec{n},t} + delta \end{aligned} \tag{3.4}$$

Hence, the value on the component grid after combination can be seen as the current value appended with a *delta*, which is equal to the difference of combination of previous terms and value on the component grid before executing that combination.

Alternative proof

It is also possible to arrive at the previous solution by considering that combination is a update of the component grid values $f_{\vec{l},t}$ by an correction vector $c_{\vec{l},t}$. i.e

$$f^*_{\vec{l},t} = f_{\vec{l},t} + c_{\vec{l},t} \tag{3.5}$$

The solver $\mathcal{F}$ would then act on this value as $\mathcal{F}_{\vec{l}}(f^*_{\vec{l},t})$. But due to asynchronous combination, the value $f^*_{\vec{l},t}$ wouldn't be available. Hence we rewrite $\mathcal{F}_{\vec{l}}(f^*_{\vec{l},t})$ in terms of $f_{\vec{l},t}$ using Taylor expansion as follows:

$$\begin{aligned} \mathcal{F}_{\vec{l}}(f^*_{\vec{l},t}) &= \mathcal{F}_{\vec{l}}(f_{\vec{l},t} + c_{\vec{l},t}) \\ &= \mathcal{F}_{\vec{l}}(f_{\vec{l},t}) + J_{\mathcal{F}_{\vec{l}}}(f_{\vec{l},t}) c_{\vec{l},t} + O(\|\delta f_{\vec{l},t}\|^2) \end{aligned} \tag{3.6}$$

Since the Jacobi matrix of a black box PDE solver cannot be known in advance, we assume that the values in the next time step are independent of the surrounding points and depend only on itself. This assumption leads to $J_{\mathcal{F}_{\vec{l}}}(f_{\vec{l},t}) = I$, where I is the identity matrix. Hence we finally get:

$$\begin{aligned} \mathcal{F}_{\vec{l}}(f^*_{\vec{l},t}) &= \mathcal{F}_{\vec{l}}(f_{\vec{l},t}) + c_{\vec{l},t} \\ &= \mathcal{F}_{\vec{l}}(f_{\vec{l},t}) + f^*_{\vec{l},t} - f_{\vec{l},t} \\ &= f^*_{\vec{l},t} + \mathcal{F}_{\vec{l}}(f_{\vec{l},t}) - f_{\vec{l},t} \end{aligned} \tag{3.7}$$

Where again, the first term on the right hand side is the combined terms using previous component grid values, and the last two terms is the *delta* value that was previously arrived at.

Algorithm 5 Parallel Combination Technique Asynchronous -Extraction Phase

```
1: procedure AsyncCombineExtract
2:     for $u \in grids(g)$ do
3:         $dsg_g \leftarrow$SparseGridReduceExtract()  ▷ extract values from global reduction of sparse grid
4:     end for
5:     for all $g \in groups$ do in parallel
6:         for $u \in grids(g)$ do
7:             $u_{new} \leftarrow u$
8:             $\Delta \leftarrow u_{new} - u_{old}$
9:             $u \leftarrow dsg_g$                    ▷ extract hierarchical coefficients from sparse grid
10:            $u \leftarrow$ dehierarchize($u$)         ▷ dehierarchize component grid
11:            $u \leftarrow u + \Delta$
12:        end for
13:    end for
14: end procedure
```

Algorithm 6 Sparse Grid Reduce Asynchronous - Extraction Phase

Input:
B : local buffer for storing combined values

```
1: procedure SparseGridReduceExtract    ▷ Extract new Sparse Grid Values from buffer
2:     for each $W_{\vec{l}} \in V_{\vec{l}}$ do
3:         extract $W_{\vec{l}}$ from $B$
4:     end for
5: end procedure
```

3.1.3. Execution Cycle

It was preferred that for every new asynchronous combination cycle, that the extraction phase of the previously initiated global reduce be carried out first before starting the next initialization phase. Hence, a snapshot of single asynchronous combination would consist of an extraction phase, followed by the next initialization block. (A reversal of sequence carried out in a normal combination). This ensures that every new asynchronous combination is initialized with values updated from the previous sparse grid reduction. The first and the last asynchronous combination step however, only consists of the initialization and extraction phase respectively. Since a asynchronous combination step is spread out over two time steps, the number of global reduce carried out would always be one less than required. To compensate for this, the last combination step carries out a normal combination cycle, after the last extraction block. For experiments that require just one combination, it was decided to just carry out a normal combination technique, as asynchronous method require minimum of two cycles to complete.

Since non-blocking reduce was used, the extraction phase could only be initiated if the previous sparse grid reduction was completed. This gave rise to two variants of asynchronous method, depending on whether it was decided to skip the combination cycle if the global reduce had not commenced; or if it was decided to carry the next step by explicitly waiting for the commencement of communication. The two approaches are discussed in detail in the section 3.3.

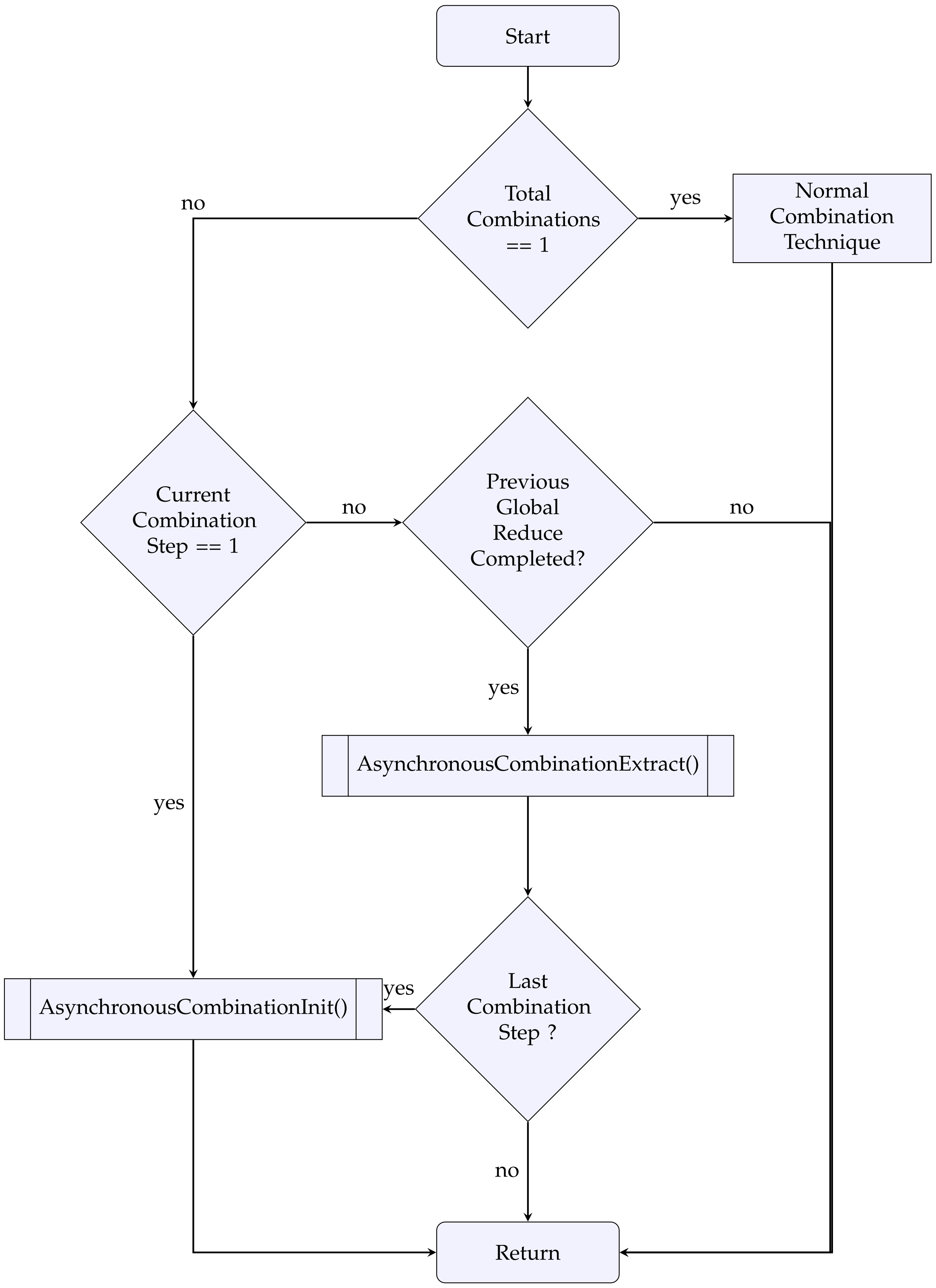

Start
Total Combinations == 1
yes
no
Normal Combination Technique
Current Combination Step == 1
no
Previous Global Reduce Completed?
no
yes
AsynchronousCombinationExtract()
yes
Last Combination Step ?
yes
AsynchronousCombinationInit()
no
Return

Figure 3.1.: Asynchronous Combination Flowchart

3.2. Asynchronous Odd-Even Combination Technique

Preliminary tests comparing the normal and asynchronous combination technique, revealed that the normal combination technique had faster execution time in comparison to the later in certain scenarios. This was due to the presence of two de-hierarchization stages per one combination step of the asynchronous technique. As, de-hierarchization is a time intensive method, any time advantage gained by asynchronous combination was rendered ineffective. To overcome this, a modified combination technique was developed that ensured that the number of de-hierarchization stage per combination step remained at one. The new algorithm was conceived by rewriting the existing equation 3.4 into two distinct parts as follows:

Since, right after the combination step, the asynchronous combination technique prepares its data for combination. The values obtained at the end of next combination at $t + 2\Delta t$ would be

$$f_{\vec{l},t+2\Delta t} = P_{\vec{l}} f^c_{\vec{n},t+2\Delta t} + f_{\vec{l},t+2\Delta t} - f_{\vec{l},t} \tag{3.8}$$

Rewriting $f^c_{\vec{n},t+2\Delta t}$ as the output of combination

$$f_{\vec{l},t+2\Delta t} = P_{\vec{l}} \sum_{\vec{l}\in\mathcal{I}} c_{\vec{l}} f_{\vec{l},t+\Delta t} + f_{\vec{l},t+\Delta t} - f_{\vec{l},t} \tag{3.9}$$

since the value of $f_{\vec{l},t+\Delta t}$ is already known from 3.4, we get

$$f_{\vec{l},t+2\Delta t} = P_{\vec{l}} \sum_{\vec{l}\in\mathcal{I}} c_{\vec{l}} \Big\{ P_{\vec{l}} f^c_{\vec{n},t} + f_{\vec{l},t} - f_{\vec{l},t-\Delta t} \Big\} + f_{\vec{l},t+\Delta t} - f_{\vec{l},t} \tag{3.10}$$

from $P_{\vec{l}} \sum_{\vec{l}\in\mathcal{I}} c_{\vec{l}} \Big\{ P_{\vec{l}} f^c_{\vec{n},t} + f_{\vec{l},t} - f_{\vec{l},t-\Delta t} \Big\}$ the last two terms can be pulled out of the projection operator as they are dependent only on the subspace, and $\sum_{\vec{l}\in\mathcal{I}} c_{\vec{l}} P_{\vec{l}} f^c_{\vec{n},t}$ is just $f^c_{\vec{n},t}$, this gives us the final resulting equation

$$f_{\vec{l},t+2\Delta t} = P_{\vec{l}} f^c_{\vec{n},t} + f_{\vec{l},t+\Delta t} - f_{\vec{l},t-\Delta t} \tag{3.11}$$

This shows that the asynchronous combination can be constructed even from combination of values two time steps before the current one. So for odd and even time step ($t + (2n+1)\Delta t$ and $t + (2n+1)\Delta t$)the equation can be rewritten as

$$\begin{aligned} f_{\vec{l},t+(2n)\Delta t} &= P_{\vec{l}} f^c_{\vec{n},t+(2n-2)\Delta t} + f_{\vec{l},t+(2n-1)\Delta t} - f_{\vec{l},t-(2n-3)\Delta t} \\ f_{\vec{l},t+(2n+1)\Delta t} &= P_{\vec{l}} f^c_{\vec{n},t+(2n-1)\Delta t} + f_{\vec{l},t+(2n)\Delta t} - f_{\vec{l},t-(2n-2)\Delta t} \end{aligned} \tag{3.12}$$

Hence, for every hierarchization and combination initiated at an odd time step, its appending of *delta* followed by de-hierarchization can be performed at the next even time step. Performing them in tandem, renders only one hierarchization and de-hierarchization per each time step. As a consequence of its behavior, this new technique is termed as *Odd-Even Asynchronous Combination Technique*.

A slight drawback with this scheme however is that, the values should be stored for two time steps now, compared to only one as in the previous asynchronous combination scheme. Figure 3.2 shows the complete work flow for this scheme. As seen, the check for completion of sparse grid reduce, would be need to be done before the next odd-even asynchronous combination step is initiated.

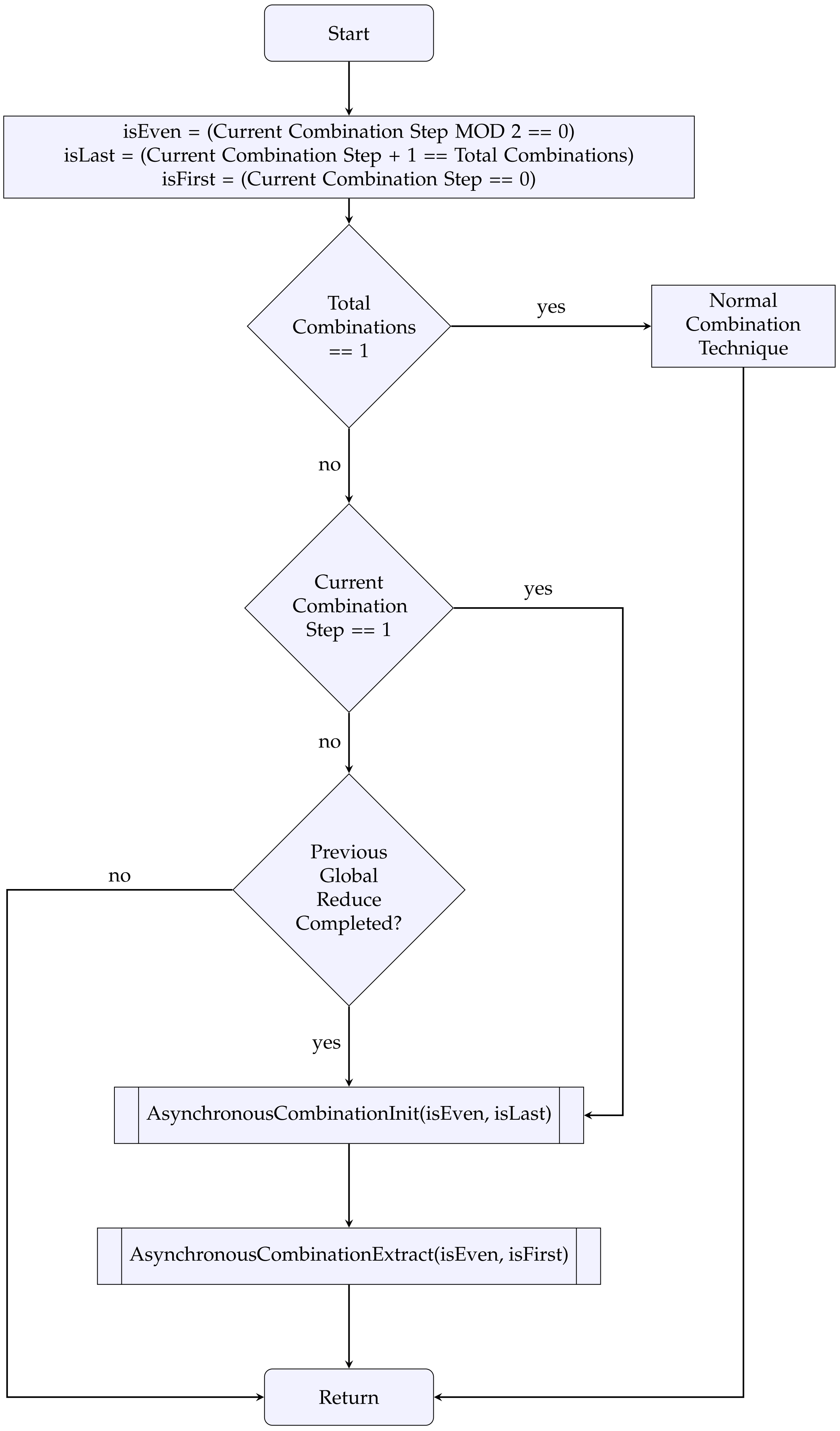

Start
isEven = (Current Combination Step MOD 2 == 0)
isLast = (Current Combination Step + 1 == Total Combinations)
isFirst = (Current Combination Step == 0)
Total Combinations == 1
yes
Normal Combination Technique
no
Current Combination Step == 1
yes
no
Previous Global Reduce Completed?
no
yes
AsynchronousCombinationInit(isEven, isLast)
AsynchronousCombinationExtract(isEven, isFirst)
Return

Figure 3.2.: Asynchronous Odd Even Combination Flowchart

3.3. Reduced Asynchronous Combination

Though the implementation of reduced combination was already present in the SG++ framework, initial tests showed that the values were not in agreement with the expected ones. This was because the previous implementation rendered every non communicated data to zero. Though this was fixed in later iterations, the results obtained form the old implementation are also included in the performance analysis tests.

For a normal combination step employing a reduced combination, the component grids which were updated after sparse grid reduction and those which were isolated from the global reduction both remain in hierarchized state. In asynchronous combination however, as the component grids are not hierarchized during the beginning of the extraction phase, reduced combination would cause the component grids involved in sparse grid reduction to be hierarchized while the grids that were not involved in the reduction would remain de-hierarchized. Additionally, after a full asynchronous sparse grid reduction all the component grids would have values reduced from previous time cycle. Whereas in a reduced sparse grid reduction, non-participating component grids would still contain current iteration of values.

One solution to these problems, would be to transfer the stored component grid values of the previous time step into the current component grids, and then hierarchize all the component grids at the beginning of each extraction phase. but as hierarchization is a time intensive step this solution was rejected.

Instead, it was preferred to store the hierarchized values during the hierarchization step of the initialization block, and transfer those hierarchized values into the component grids at the start of the extraction step. This would mean that the process groups would now have to store two sets of previous component grid values, one hierarchized and one not. Swapping the execution order of de-heirarchization and append step in the extraction phase mitigates this problem. As de-hierarchization and appension are both linear operators, the oder in which they are performed has no impact on the overall execution. But this ensures that the process group be required to store only one set of component grid values from previous time step (i.e the hierarchized values).

3.4. Dynamic Asynchronous Combination

Since the execution of the extraction phase of asynchronous combination technique, requires the global reduction process to be completed, at the arrival of the control flow. Based on how this could be handled, two variants of the asynchronous combination technique were implemented. The static method utilized MPI_WaitAll and explicitly waited for the reduction process to complete and carried out the next asynchronous block in the initiated time step. The Dynamic method on the other hand used MPI_TestAll to poll is the sparse grid reduce had commenced. If it hadn't it would skip the current combination cycle, and would continue skipping the combination step, till the sparse grid reduction finally terminated. Only then would the next sequence of asynchronous combination step be carried out. Figures 3.3 and 3.4 demonstrate both the aforementioned variants.

The dynamic method, would in principle have faster execution time than the static method (provided that the simulations have communication cycle that takes at-least twice the time required for the calculation phase). As the dynamic variant would just continue on with the next calculation phase in case of unfinished sparse grid reduction. In contrast, the static implementation would lose time by explicitly waiting for the commencement of communication, before carrying out the next calculation phase. This also makes the dynamic method volatile, as it cannot be accurately determined when the combination steps will be skipped and when they will be carried out. The accuracy would also be slightly reduced, as lesser combination steps will be executed. Nonetheless, as no tests could be carried with communication steps overtaking the calculation steps over the course of this thesis. The characteristics of dynamic combination technique is mere conjecture.

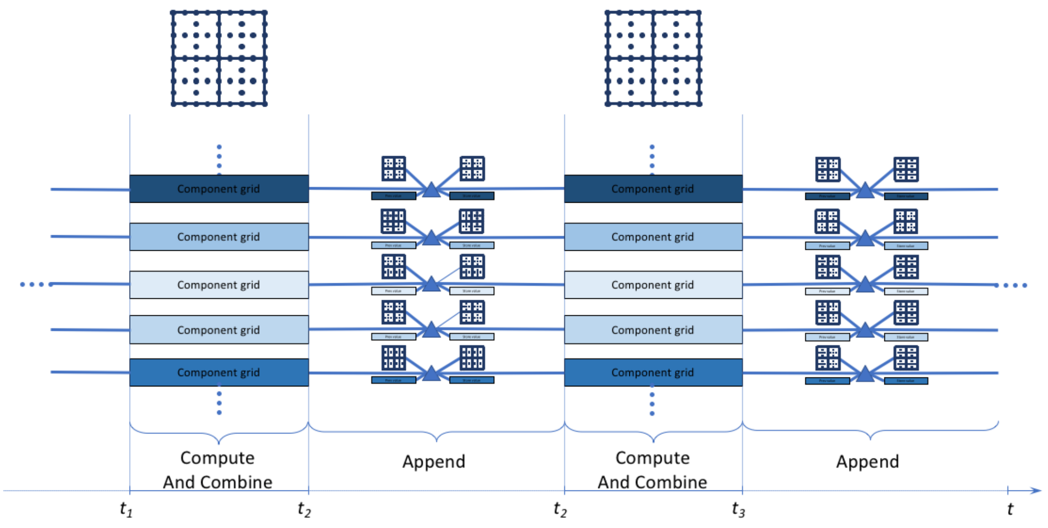


Figure 3.3.: Static asynchronous combination technique

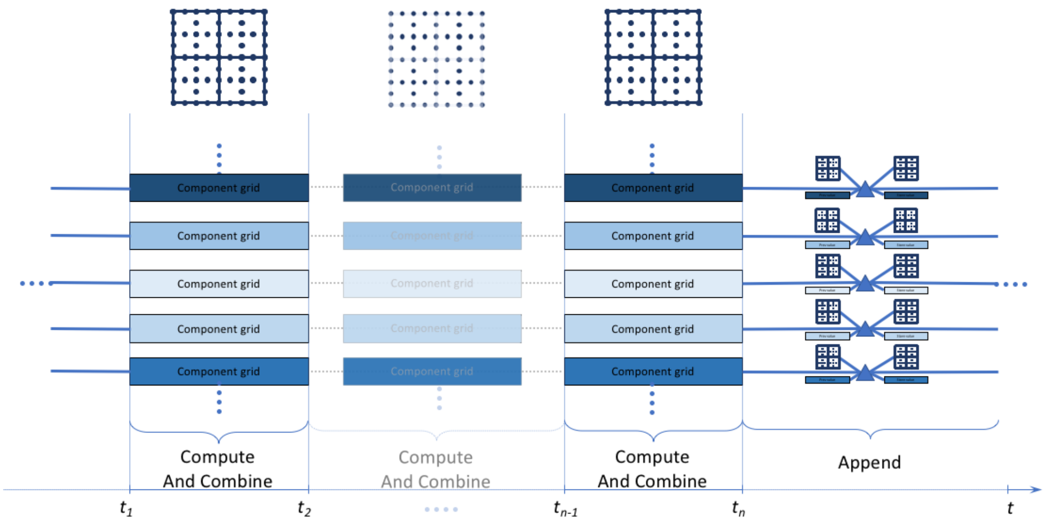


Figure 3.4.: Dynamic asynchronous combination technique

3.5. Integration Into SG++

SG++ framework's philosophy of having the solver independent of the problem was followed while integrating the algorithm. New signals *CombineAsync* and *CombiAsyncOddEven* were added alongside the other existing states that the worker processes could take. The TASK class was also suitably modified in order to accommodate storing of component grid values from two time steps. Furthermore, the TASK class instances were also equipped with *MPI_REQUESTS* variables to retrieve status of the sparse grid reduction. The local buffers were also made part of the TASK class so that are accessible during both initialization and extraction phase. The Appendix sectionA.3 contains all the implemented code.

3.6. Benchmarks

Performance of the asynchronous combination techniques were evaluated using two time dependent test problems: Advection-Diffusion and GENE. Both the problems had important characteristics that made them ideal for analysis. These problems require combination technique to be carried out at regular interval in order to guarantee stability. Another important feature of these problems was that they had readily available full grid solutions: either in the form of data from the previous work carried out on these problems, or in case of Advection-Diffusion a standard analytical solution which could be easily calculated in definite amount of time. The ability of these problems to scale easily in terms of grid-points and dimensions also serve as important criteria for their selection. The following section further elaborates on the two test problems, their integration into the combination scheme followed by the methodology used to evaluate the results obtained in either.

3.6.1. Advection-Diffusion

A standard Advection-Diffusion model deals with time evolution of particles in a flowing medium [19]. Its one-dimensional general representation for a concentration $u(x,t)$ is given by:

$$\underbrace{u_t + (au)_x}_{advection} = \underbrace{(du_x)_x}_{diffusion} + f(u) \tag{3.13}$$

Where f denotes the local change in $u(x,t)$ due to sources, sinks or reactions. And a and d are the advection and diffusion coefficients.

For analysis, the equation was developed for d spatial dimensions and the advection and diffusion coefficients were considered to independent of u. Local changes f was set to a constant and explicit Euler scheme was used for time integration.

Following denotes the final equation with its appropriate initial conditions:

$$\begin{aligned} \partial_t u - \Delta u + \vec{a} \cdot \nabla u &= f \quad in\ \Omega \times [0,T) \\ u(\cdot,t) &= 0 \qquad\quad in\ \partial\Omega \end{aligned} \tag{3.14}$$

with $\Omega = [0,1]^d, \vec{a} = (1,1,\ldots,1)^T$ and $u(\cdot,0) = \exp^{-100\sum_{i=1}^{d}(x_i-0.5)^2}$. Where Δ and ∇ denote the Laplace ($\sum\limits_{i=1}^{d} \frac{\partial^2}{\partial x_i^2}$) and Gradient ($\sum\limits_{i=1}^{d} \frac{\partial}{\partial x_i^2}$) operator respectively.

3.6.2. GENE

GENE(Gyrokinetic Electromagnetic Numerical Experiment) is a software package dedicated to solving of the Gyrokinetic Vlasov equation in a flux tube domain. The equation represented by

$$\frac{\partial F_s}{\partial t} + \frac{d\vec{X}}{dt} \cdot \nabla F_s + \frac{dv_\parallel}{dt}\frac{\partial F_s}{\partial v_\parallel} + \frac{d\mu}{dt}\frac{\partial F_s}{\partial \mu} = 0 \tag{3.15}$$

describes the propagation of a species s in time through a 5 dimensional distribution function $F_s(x,y,z,v_\parallel,\mu)$. Where x,y,z are the co-ordinates of the species, and $v_\parallel$ and μ are the velocity and magnetic moment respectively. (It is to be noted that GENE represents x and y in its spectral space. Hence, k_x and k_y are the exact parameters that were subject to modification during simulations). Each species represents a different type of simulated particles (usually ions and electrons). GENE Employs δf-splitting the distribution function cab be split into a Maxwellian background distribution $F_{0,s}$ and a fluctuating part g_s. Thus, the governing equation solved by GENE can be symbolically represented as

$$\frac{\partial \vec{g}}{\partial t} = \mathcal{L}\vec{g} + \mathcal{N}\vec{g} \tag{3.16}$$

where $\vec{g}$ is a vector of different g_s and the mathematical operators $\mathcal{L}$ and $\mathcal{N}$ represent the linear and non-liner part of the equation.

For analysis, GENE was operated in linear mode and only one species was considered. Rendering the final equation that was solved as

$$\frac{\partial g}{\partial t} = \mathcal{L}g \tag{3.17}$$

3.6.3. Test Execution

The following sequence of steps were executed for every test run. First, the *ctparam* file containing the configuration settings for the current test is read. Then the MPI processes are divided according to the provided process groups and their corresponding communicators initialized. Following which, the *load model* is created. An assertion is then made to check if the number of processes are in agreement with the provided parallelization vector. A *combination scheme* is then created in adherence to the provided dimension and, the maximum and minimum level vectors. New *Tasks* are created in accordance to the *combination scheme*. Next, the *process manager* is the initialized along with the combination parameters, which are then sent to all the process groups under the manager. Finally the *tasks* are distributed in accordance with the *load model* and the computation is initiated, with the specified combination scheme occurring at intervals specified in the initial configuration. In case of GENE, fault detection was performed after every computation and combination step. And a suitable fault correction steps are executed, which includes but is not limited to: recovery of communicators, re-computation of all tasks and even reinitialization of process groups. The flow commences with the error evaluation of the initiated test, along with the timing file, which stores the time taken by various activities in a json format.

3.6.4. Error Evaluation

The sparse grid to full grid error for Advection-Diffusion was defined by the parameters E_2 and E_∞, with

$$\begin{aligned} E_2 &= \frac{\|u^c_{\vec{n}} - u_{ref}\|_2}{\|u_{ref}\|_2} \\ E_\infty &= \frac{\|u^c_{\vec{n}} - u_{ref}\|_\infty}{\|u_{ref}\|_\infty} \end{aligned} \tag{3.18}$$

where u_{ref} and $u^c_{\vec{n}}$ refer to full grid solution and the solution using combination technique at the target level $\vec{n}$ respectively.

While, for GENE The error computation was considered to be $e^c_{\mathcal{I}} := |\lambda^c_{\mathcal{I}} - \lambda_{\text{ref}}|$ with λ being the eigenvalue of $\mathcal{L}$ with the largest (positive) real part.

4. Results

This section deals with the results that were obtained from various tests runs using the Advection-Diffusion and GENE for different values of $\vec{l}_{\text{min}}$ and $\vec{l}_{\text{max}}$, dimensions (d) and number of combination per steps. Every asynchronous combination scenario, was compared to its normal combination counterpart, and a reference part, which consisted of equal number of calculation steps, but only only one combination step. Only, basic tests were carried out on Advection-Diffusion problem to test the stability of the implemented asynchronous solution, while more extensive tests were performed with GENE.

4.1. Advection-Diffusion

figures 4.1 and 4.2 show the errors E_2 and E_∞ obtained for different $\vec{l}_{\text{max}}$ and dimensions. All tests were carried out with 1000 total time step, one combination for every 100 calculation steps, or a hundred total combination steps. $\vec{l}_{min}$ was kept to be 1 for all dimensions. The Euler time step ΔT was also fixed at 10^{-4}. Further the task was divided between 6 workers with 1 process each. The dimension d was varied from 2 to 4. The reference full grid target level $\vec{n}$ was set equal to the $\vec{l}_{min}$ and they both were varied.

It can be seen that errors for asynchronous combination technique were found to between a normal combination technique, and the one involving just one combination step at the end. For all dimensions, error values decreases with increase in $\vec{l}_{min}$.

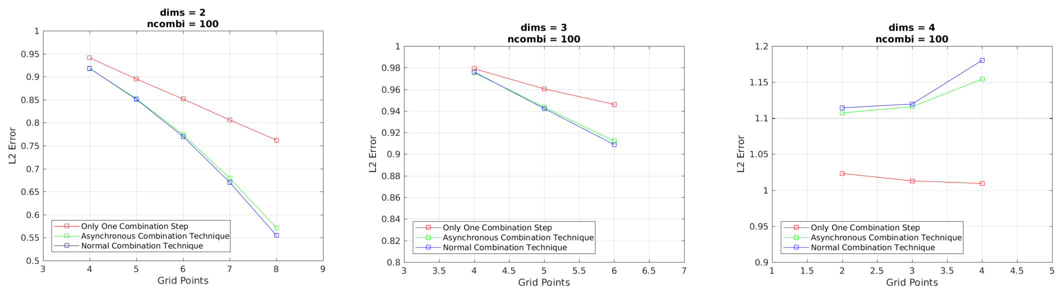


Figure 4.1.: E_2 for Advection

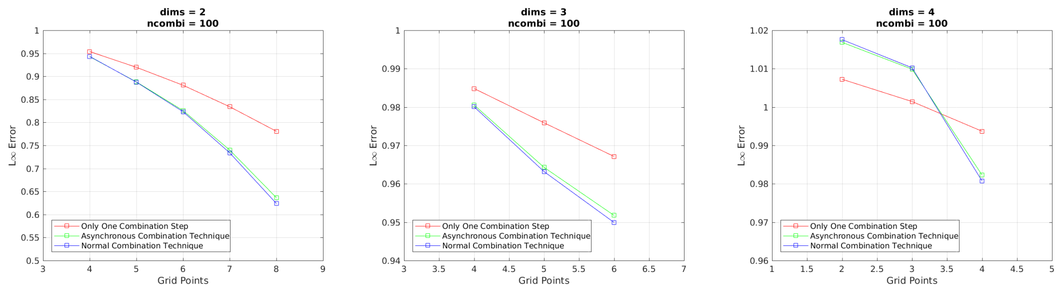


Figure 4.2.: E_∞ for Advection

It was also observed that if the time steps between two subsequent combinations were significantly large, then the result obtained from asynchronous combination techniques were of no use, this was probably due to the fact that values that values had significant changes between those time steps, and adding back the previous combination values just led to further degradation instead of improvement of the current results. This was clearly seen by the exponential rise in errors for the asynchronous combination technique. Further, the relative errors got worse with increase in total combination steps. This degradation was seen to be significant at lower grid-points as the dimensions of the problem increased. The table 4.1 lists the following degradation of a 10000 time step asynchronous combination technique, and the first grid point where this occurs for a particular dimension and the total number of combinations used in that trial.

Dimension	Number of combination	Grid Point of first degradation	E_∞	E_2
2	20	9	3.035933	6.180694
	50	9	465.603428	900.002642
	100	9	720.815	1448.52
3	203	8	0.934907	4.203893
	50	8	0.897940	2.011485
	100	8	719.450955	107.687332

Table 4.1.: Asynchronous combination technique degradation

4.2. GENE

To establish that the proposed asynchronous techniques work, it was necessary that it satisfied two conditions. One: The accuracy obtained by asynchronous method is considerably similar to the normal combination technique. Second: the execution time ought to be noticeably faster than the latter. Hence more comprehensive tests were carried out with GENE, in-order to argue the same. Multiple test were carried out using various combination of dimensions, combination steps and reduced dimension combination. The error rate and the timing obtained as a result of this are discussed in the following section.

4.2.1. Error Analysis

Like stated before the error to the full grid solution was $e^c_{\mathcal{I}} := |\lambda^c_{\mathcal{I}} - \lambda_{\text{ref}}|$. Full grid evaluation was already present for $\vec{n}$ = [3 1 7 7 7 1] and [3 1 8 8 8 1] as consequence of other experiments that are carried out using GENE. Hence different tests concerning errors were carried out with $\vec{l}_{max}$ = [3 1 7 7 7 1] and [3 1 8 8 8 1]. Also, as both the asynchronous combination schemes are supposed to have same behavior, most graphs in this section will only contain one plot representing both the techniques.

Dimension Variation

Figure 4.3 shows the error variation with $\vec{l}_{max}$ set to [3 1 7 7 7 1] and [3 1 8 8 8 1] for a total of 6000 calculation steps, with one combination step after every step. The $\vec{l}_{max}$ was set to [3 1 5 5 5 1] for both. 4 process groups were used, with 128 processes in each. The simulations were executed against normal combination, the asynchronous combination and also a case of just one combination at the end(which serves as the baseline for worst performance).

As seen, the asynchronous combination closely follows the error of a normal combination. The errors are significantly better than those obtained by carrying a single combination at the end. The errors also decreased with increase in $\vec{l}_{max}$.

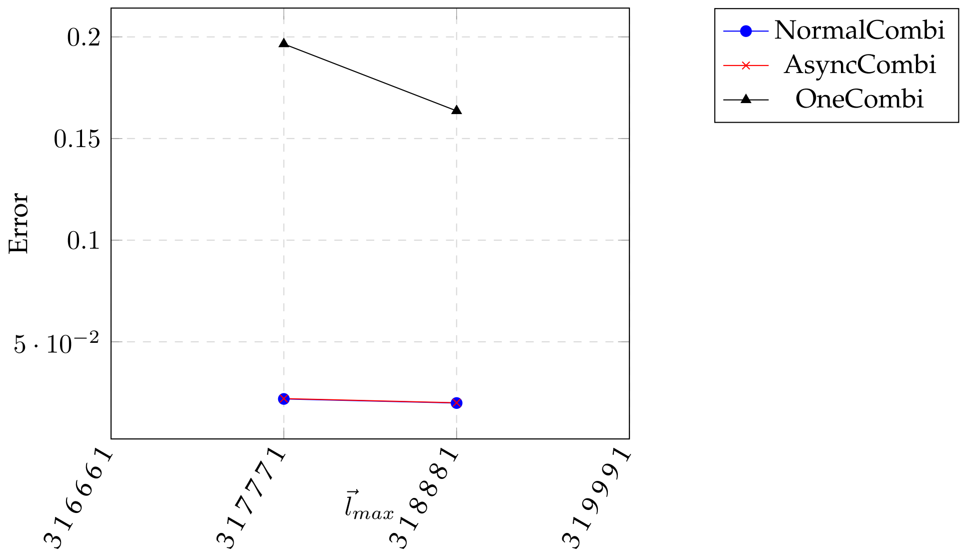


Figure 4.3.: Error Variation with $\vec{l}_{max}$

Varying Combination Steps

Figure 4.4 shows the error variation with number of combination steps. All computations were performed with total of 6000 calculation steps and $\vec{l}_{min}$ = [3 1 5 5 5 1] as before, for both $\vec{l}_{max}$ [3 1 7 7 7 1] and [3 1 8 8 8 1], but with different combination steps i.e., 2, 10, 60 ,100 ,600, 3000 and 6000 for $\vec{l}_{max}$ = [3 1 7 7 7 1] and 2, 600, 3000, 6000 for $\vec{l}_{max}$ = [3 1 8 8 8 1]

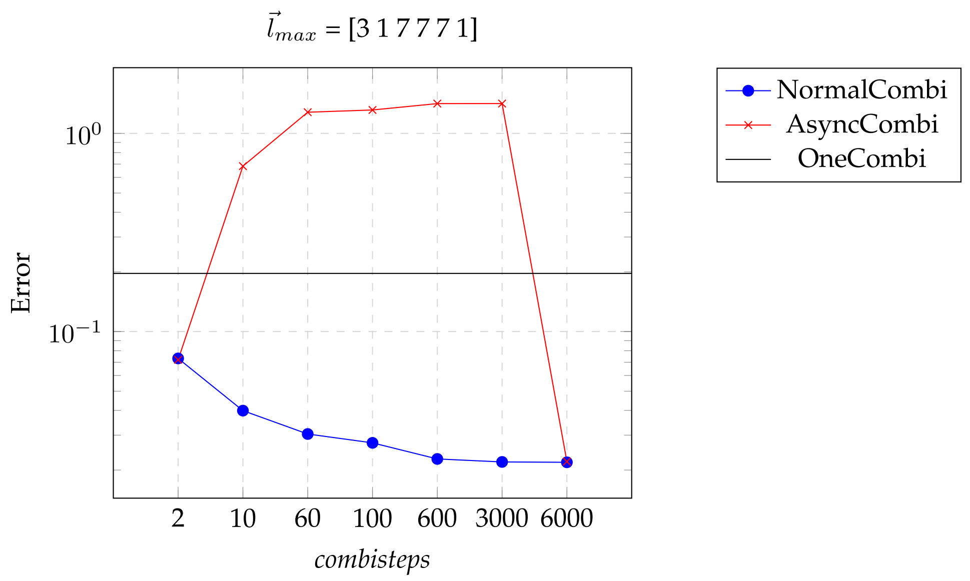
It can be seen that for arbitrary combination step values, the error of asynchronous technique, degrades below the level of a single normal combination step. The error values in asynchronous schemes didn't seem to follow any obvious pattern, though the normal combination displayed a decrease in error rate with increasing combination steps. This trend was not of much concern, as even normal combination techniques have demonstrated the same behaviour during previous instances, though for different set of $\vec{l}_{max}$ and $\vec{l}_{min}$ [17]. No concrete explanation can be provided at the moment for this behavior. But it is believed that effect of number of combination steps on the convergence of the simulation is a property of the problem being solved. Therefore, while carrying out time intensive simulations, it is important to verify at regular intervals if the provided configuration leads to convergence or not.

However the issue of blow up previously observed with Advection-Diffusion wasn't observed with GENE. There were no random degradation of error values for different combination steps. The error values always fell under the hard limit of 1.4142, never going beyond this point.

Reduced Combination

To check if the asynchronous methods can be used in tandem with other communication accelerating methods, tests with Reduce Combination methods were necessary. Hence, simulations were carried again for 6000 combination steps with $\vec{l}_{max}$ set to [3 1 7 7 7 1] and $\vec{l}_{min}$ set to [3 1 5 5 5 1]. But the sparse grid constructed during sparse grid reduce, was not a complete one rather a reduced one which used reduced minimum and maximum level vectors $\vec{l}^{R}_{min}$ and $\vec{l}^{R}_{max}$ respectively.

The figure 4.6 shows the different levels of error that were obtained over three different iterations of reduced combination algorithm, with only $\vec{l}^{R}_{max}$ set to [0 0 1 1 1 0].(This configuration was used because in a correctly implemented situation the error values obtained would be equal to the non-reduced combination). Test were carried out for both asynchronous combi-



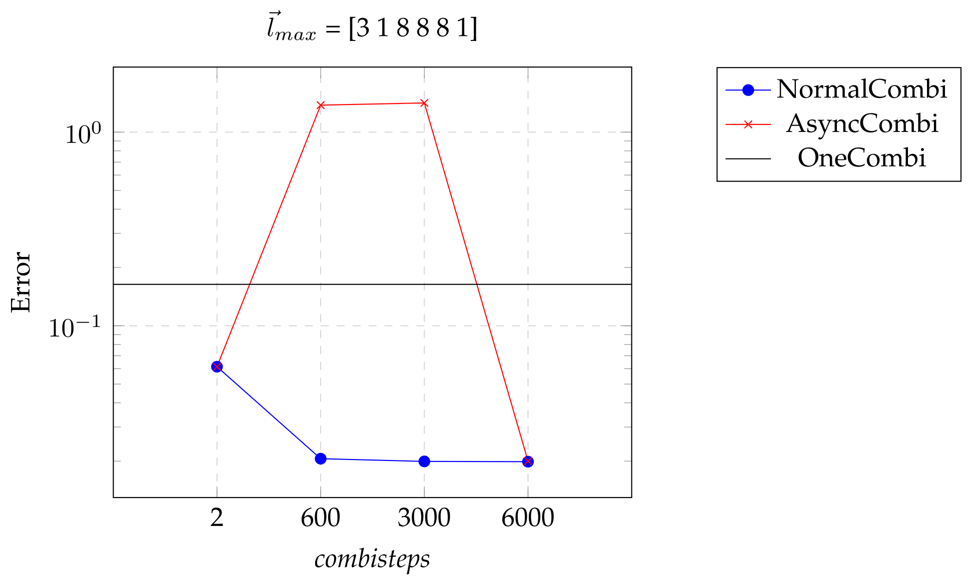


Figure 4.4.: Error Variation with *combisteps*

nation technique, the normal combination technique and with the case of single combination step at the end. Furthermore, the trials were carried out for three different modes of reduced combination: the existing reduced combination mode, where values of component grids not part of the reduced sparse grid were set to zero on communication; a fixed reduced combination mode, where the values of non participating grids were left untouched, but asynchronous combinations were not adapted using the implementation mentioned in section 3.4; finally a fixed asynchronous mode, with asynchronous combinations adapted for reduced combination.

It can be seen that if the values of non-participating component grids were set to zero on every combination, then the error value obtained is worse than carrying out a single combination. However, for asynchronous combination technique this would still give a better result than carrying out proper reduced combination, but without properly adapting the asynchronous combination steps. It can also be seen, that since the two asynchronous method utilize values of different time steps, letting the non-participatory component grids retain the current value would lead to different error levels. The proper implementation of reduced combination for the aforementioned $\vec{l}^{R}_{max}$ however shows same error results as a full combination, thus providing the expected optimization.

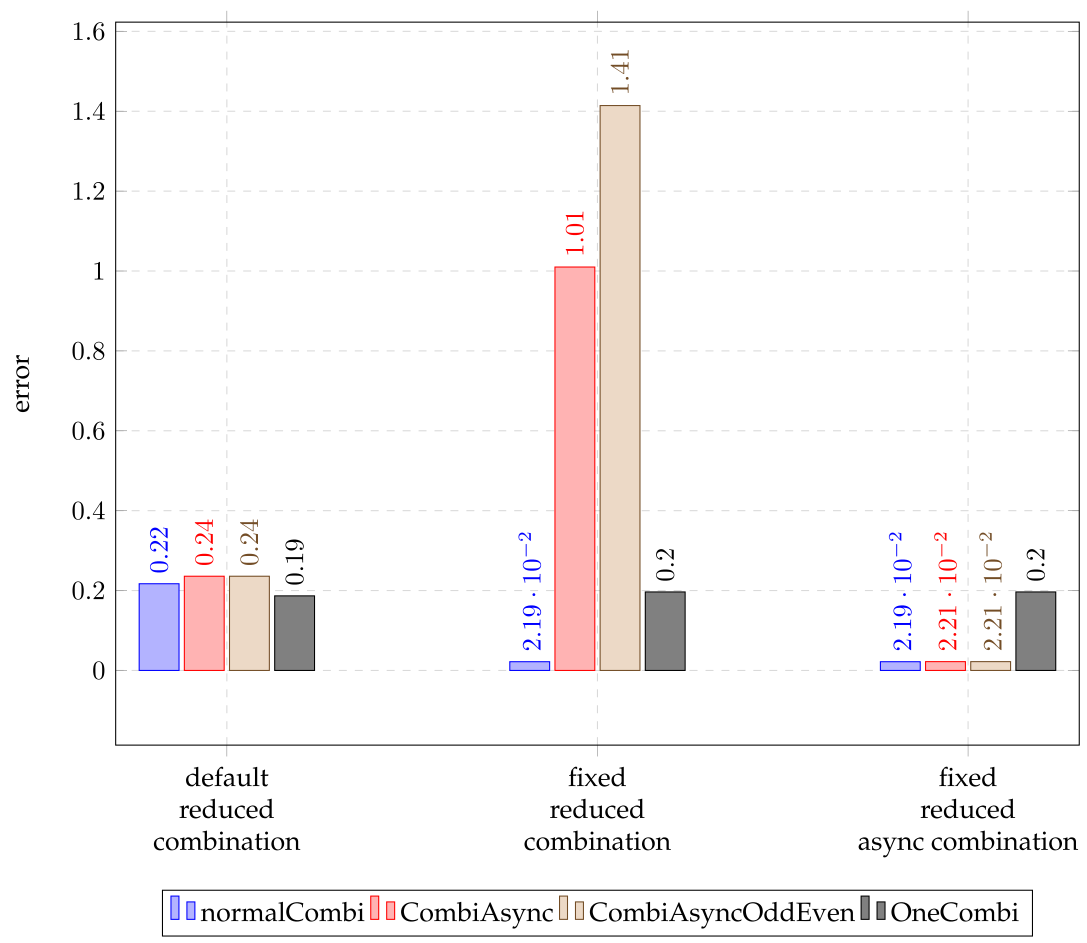


Figure 4.5.: Different iterations of reduced combination

For further tests with reduced combination it was required to change the test parameters a bit as having $\vec{l}_{max}$ as [3 1 7 7 7 1] and $\vec{l}_{min}$ as [3 1 5 5 5 1] allowed for only two levels of reduction. Hence further tests were carried out by setting $\vec{l}_{max}$ to [3 1 7 7 7 1] and $\vec{l}_{min}$ to [3 1 4 4 4 1]. This change also required that there be 8 process group managers with 64 processes each and the p parameter was appropriately set to [1 1 1 8 8 1]. Figure **??** shows the error obtained over various combinations of $\vec{l}^R_{max}$ i.e., [0 0 1 1 1 0], [0 0 2 2 2 0] and [0 0 2 2 2 0] and $\vec{l}^R_{min}$ fixed at [0 0 0 0 0 0]. Test were carried for both asynchronous and normal combination technique. A single complete sparse combination at the end was used as reference.

As seen the reduced combination works only for $\vec{l}_{max}$ = [0 0 1 1 1 0] for all combination technique. Going beyond this level cause the error values, to drop below the level of even single combination step. Thus, reduced combination can provide only one level of performance optimization (at least in case of GENE) and may not be an worthy option, when looking for timing improvements as it would be better to carry out just one full combination step at the end, and arrive at a better result.

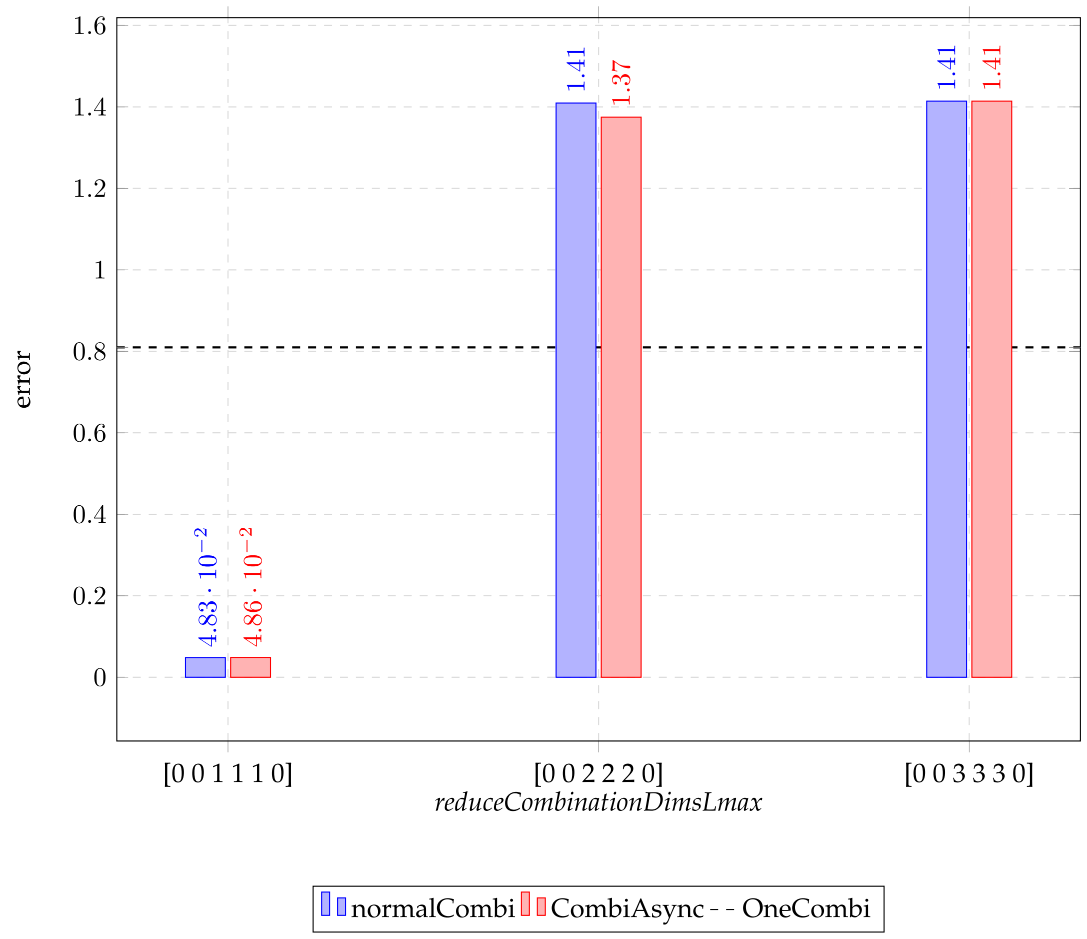


Figure 4.6.: Different iterations of reduced combination

4.2.2. Timing Analysis

The timing tests are crucial in validating the overall performance of the asynchronous techniques. To adequately check the behavior of these schemes, it was deemed essential to construct a larger problem, which as a consequence would have significant communication time. However, the amount of cores and time that could be allotted set restraints on how big the problem could get. Hence, considering both the aforementioned reasons the tests were carried for $\vec{l}_{max}$ = [3 1 11 11 11 1], [3 1 12 12 12 1] and [3 1 13 13 13 1], with $\vec{l}_{min}$ fixed at [3 1 4 4 4 1]. The process groups and their processes were set to t 4 and 64 respectively. With parallelization vector p appropriately set to [1 1 1 8 8 1]. The total execution steps was taken as 200 for all the cases.

Figure 4.7 shows the variation of execution time with number of combination steps for $\vec{l}_{max}$ = [3 1 13 13 13 1]. It can be seen that the execution time for all three techniques are around the same time range. The Asynchronous Odd-Even technique was always faster than the other Asynchronous method, owing to one less de-hierarchizing step.

As previous tests that were carried out with GENE explicitly state that as number of process groups and processes increase the hierarchization and de-hierarchization process undergo significant reduction in execution time, but the combination time remains more or less unchanged [17]. It is important that the combination time and more importantly the time taken for sparse grid reduction be subject to comparison.

Figure 4.8 shows one instance of deconstructed time sequences of the different combination technique for 20 combination steps. For all three methods it can be observed that though combination consumes a significant part of the total execution time, the majority of its time

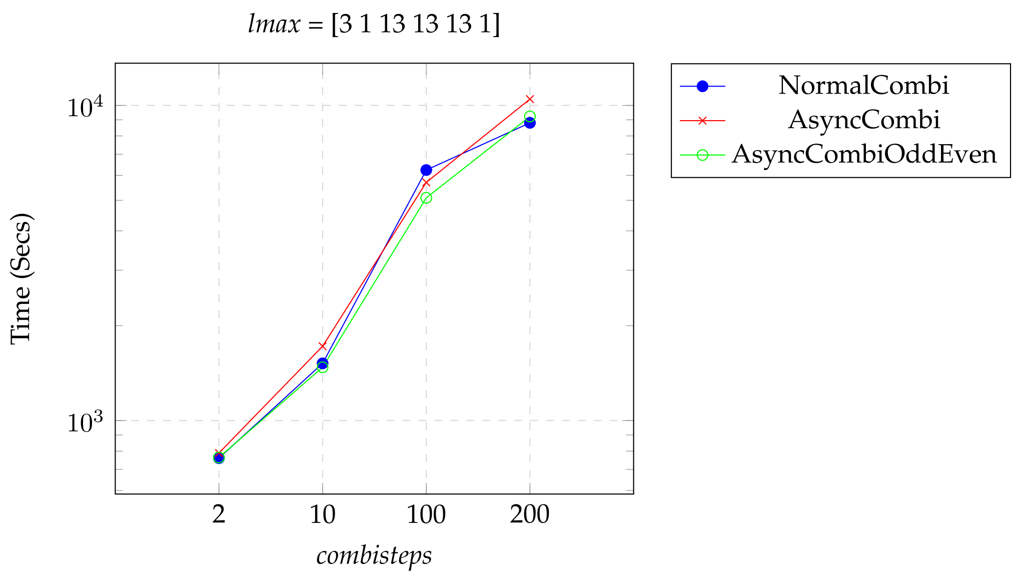


Figure 4.7.: Total Execution Time vs *combisteps*

however is spent on the hierarchization and de-hierarchization steps. The global reduce however forms only a tiny part of the same. Despite that, the glaring timing difference between normal combination against the asynchronous technique for this block can be clearly observed. Normal combination spends nearly 5 times more time than the asynchronous technique in this block.

This factor of ≈ 5 was consistently observed for values of $\vec{l}_{max}$. The figure 4.9 reinforces the same. Leading to the conclusion that that it is possible to simulate significantly large problems, at 5 times faster pace using the asynchronous techniques. Provided that the hierarchization and de-hierarchization steps times are suitably reduced using appropriate number of processes and process groups.

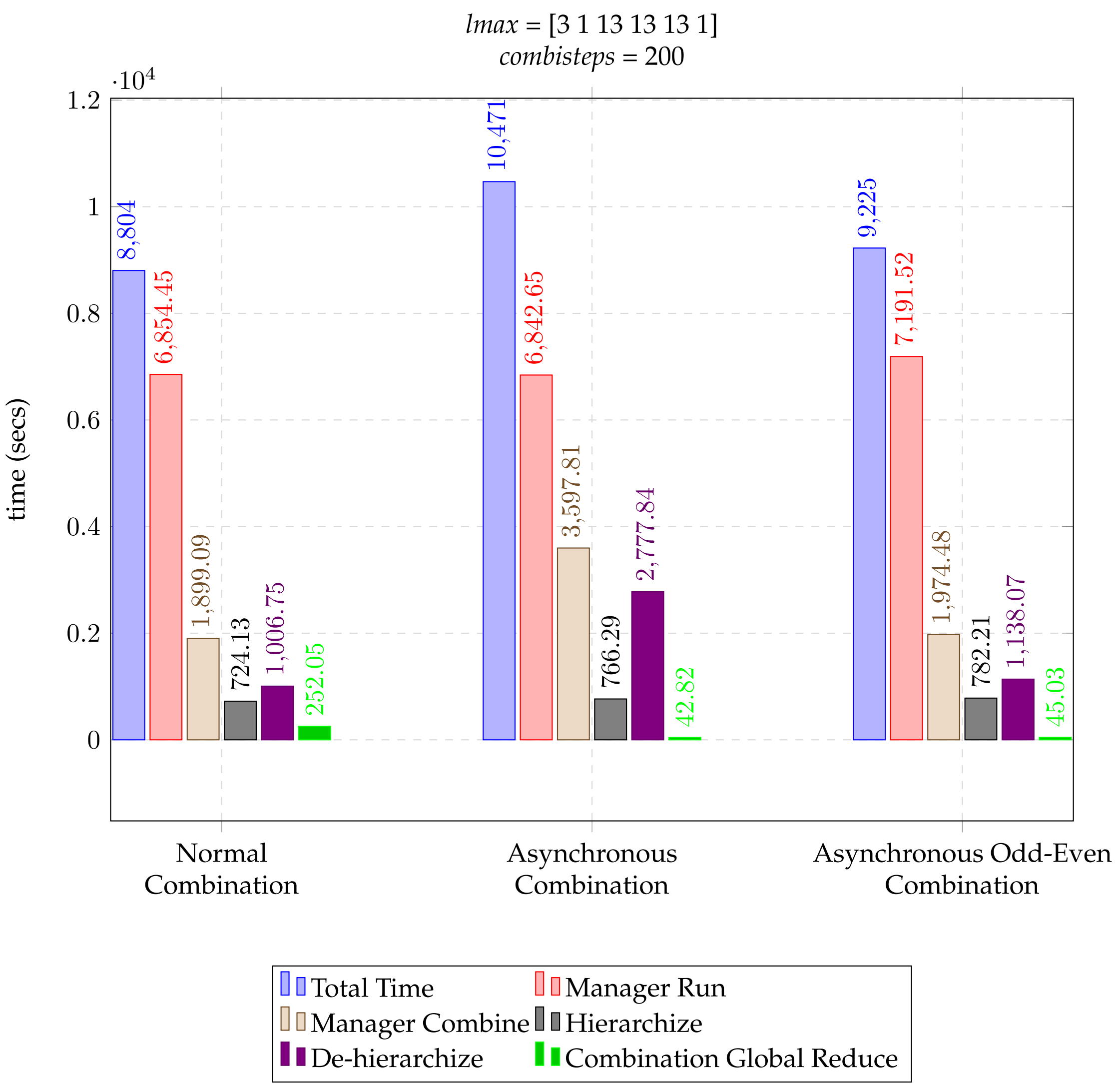


Figure 4.8.: Deconstructed execution time of various combination sequence

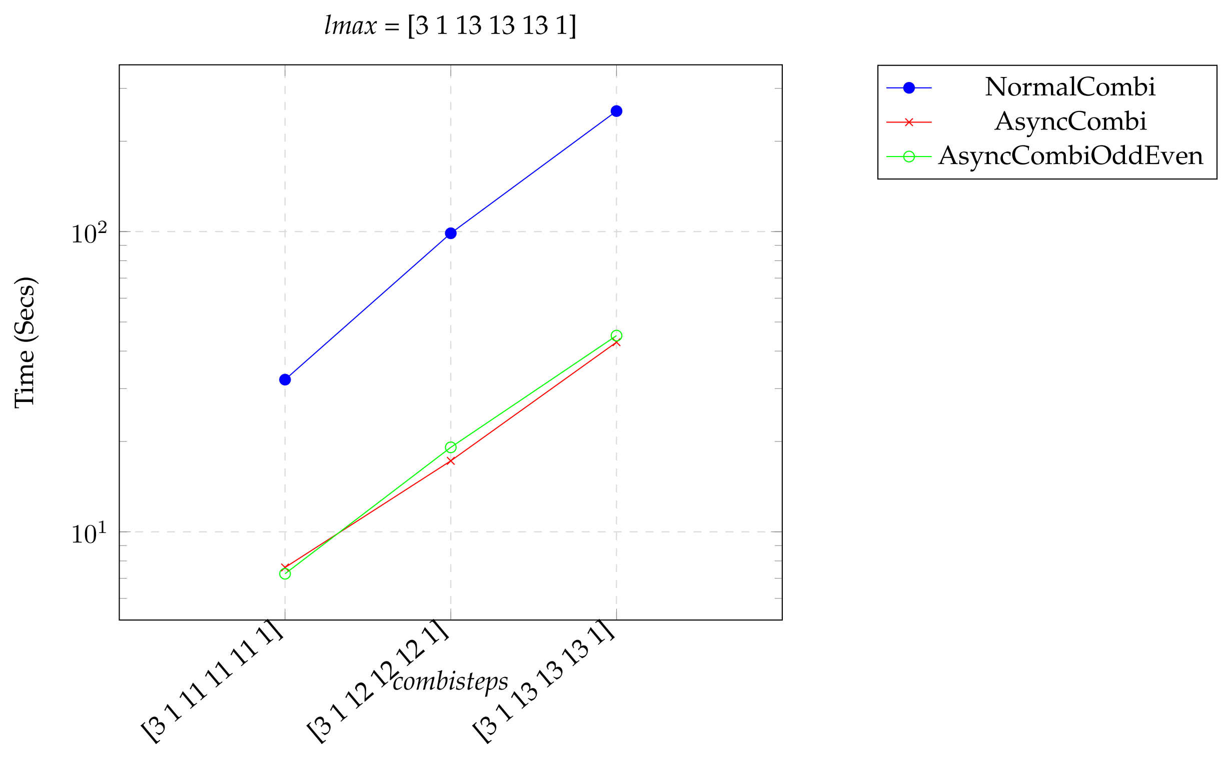


Figure 4.9.: Combination Time vs *lmax*

5. Conclusion

In this work, asynchronous/non-blocking combination techniques capable of solving time-dependent high dimensional PDEs on a large scale HPC systems were presented. Experiments with the software GENE demonstrate that here presented asynchronous techniques are capable of significantly improving the execution speed of conventional combination technique, while still maintaining the accuracy.

This work, provides the mathematical basis for the changes required in non-blocking combination technique so that its accuracy closely follows those of a blocking combination. It was shown that apart from carrying out non-blocking communication, it is necessary to append the component grid values with a suitable correction factor. This factor was derived to be the difference between the current and the previous component grid values.

The major contribution of this work is the design and successful implementation of two different combination techniques. The two techniques were demonstrated to have equal levels of accuracy. The techniques differed in two major aspect, memory and execution time. It was shown that it is possible to create a non-blocking communication technique by storing component grid values of only one time step, but this requires an additional de-hierarchization step for successful implementation. Whereas it is also possible to implement an asynchronous algorithm with same number of hierarchization and de-hierarchization steps as a normal combination technique, but this would require that component grid values of two time steps be stored.

This work, also proposes two different variants of asynchronous combinations based on how the execution flow is handled when sparse grid reduce hasn't commenced even after on full calculation step. One proposal was to explicitly wait and carry out the required number of combination, while the other was to keep skipping the combination steps till the previous sparse grid reduce commences. However, since the problems used to test weren't suitably large, the dynamic variant of skipping combinations couldn't be tested out.

An alternative time reducing combination technique called Reduced Combination was also evaluated in this work. The necessary modification to asynchronous combination technique to integrate this method was also covered. It was also found that Reduced Combination technique are capable of only one level of optimization. Going beyond the first level was found to render the accuracy obtained from Reduced Combination to be below those obtained by carrying out a single complete combination.

Limitations of Asynchronous combination techniques were also demonstrated and discussed in this work. This includes the degradation in accuracy when solving certain problems like Advection-Diffusion, in case of carrying out fewer asynchronous combinations steps in comparison to calculation steps. It was also found that for certain problems like GENE inaccurate combination steps may not cause complete degradation of accuracy, but would still lead to starkly different accuracy levels when compared to normal combination.

As we move to exa-scale computing, the problems being solved will get bigger and though the calculations will get faster, the communication will still serve as a bottleneck. The asynchronous combination techniques implemented in this work can aid in mitigating this problem. As they are shown to have the same accuracy levels of normal combination technique, but have communication time 5 times lower.

6. Future Work

Many different tests and experiments have left for the future, as they couldn't be tested out due to lack of time (i.e large experiments would need days to finish a single run) and resources (for instance the time and cores made available on linux cluster where the tests were carried out, was not enough to run bigger experiments)

Tests can be performed on problems where the communication step consumes more time than the calculation ones. This would help to establish the time improvement achieved via asynchronous combination technique. This would also be useful to see the behavior of the proposed dynamic asynchronous combination technique.

The correction factor that was appended to component grid values after an asynchronous combination was taken considering only the first order of Taylor expansion. It would be interesting to see if higher orders provide any improvement in accuracy.

The degradation in accuracy seen in Advection-Diffusion example also needs further investigation. A numerical basis for the occurrence would help to decide if asynchronous combination technique can be utilized for the particular problem or not.

Appendix

A. Implemented Code

A.1. ProcessGroupWorker

```
void ProcessGroupWorker::combineUniformAsync() {
  /*In case there is only one combine do normal combine*/
  if(combiParameters_.getNumberOfCombinations() == 1){
    combineUniform();
  }
  else{
    if(Task::isFirstCombiSequence){
      combineUniformAsyncInitHierarchizeReduce();
      Task::isFirstCombiSequence = false;
    }
    else if(isDistributedGlobalReduceAsyncCompleted()){
      combineUniformAsyncHierarchizeUpdate();
      if(currentCombi_ + 1!=
       ↪  combiParameters_.getNumberOfCombinations()){
        combineUniformAsyncInitHierarchizeReduce();
      }
    }
  }
}

bool ProcessGroupWorker::isDistributedGlobalReduceAsyncCompleted(){
  int numGrids = combiParameters_.getNumGrids();
  int finishedReduce = 0;
  int flag = 0;

  //comment the next line for dynamica
  MPI_Waitall(numGrids,Task::requestAsync,MPI_STATUSES_IGNORE);
  MPI_Testall(numGrids, Task::requestAsync, &flag,
 ↪  MPI_STATUSES_IGNORE);

  return flag;
}


void ProcessGroupWorker::combineUniformAsyncInitHierarchizeReduce(){
#ifdef DEBUG_OUTPUT
    MASTER_EXCLUSIVE_SECTION{
      std::cout << "start combining \n";
    }
#endif

```

```
  Stats::startEvent("combine init");

  // each pgrouproot must call reduce function
  //assert(tasks_.size() > 0);
  if(tasks_.size() == 0){
    std::cout << "Possible error: task size is 0! \n";
  }
  assert( combiParametersSet_ );

  //we assume here that every task has the same number of grids
  int numGrids = combiParameters_.getNumGrids();
  DimType dim = combiParameters_.getDim();
  LevelVector lmin = combiParameters_.getLMin();
  LevelVector lmax = combiParameters_.getLMax();
  const std::vector<bool>& boundary =
   ↪  combiParameters_.getBoundary();

  // the dsg can be smaller than lmax because the highest subspaces
   ↪  do not have
  // to be exchanged
  // todo: use a flag to switch on/off optimized combination

  reduceSparseGridCoefficients(lmax,lmin,

                               ↪  combiParameters_.getNumberOfCombinations(),currentCombi_,
                              combiParameters_.getLMinReductionVector(),
                              combiParameters_.getLMaxReductionVector());
  /*for (size_t i = 0; i < lmax.size(); ++i)
        if (lmin[i] > 1)
          lmin[i] -= 01;
  for (size_t i = 0; i < lmax.size(); ++i)
      lmax[i] = std::max(lmin[i],lmax[i] - 2);
  */
#ifdef DEBUG_OUTPUT
  MASTER_EXCLUSIVE_SECTION{
    std::cout << "lmin: "<< lmin << std::endl;
    std::cout << "lmax: "<< lmax << std::endl;
  }
#endif

  //delete old dsgs
  for(int g=0; g<combinedUniDSGVector_.size(); g++){

    if (combinedUniDSGVector_[g] != NULL)
      delete combinedUniDSGVector_[g];
  }
  combinedUniDSGVector_.clear();
  // erzeug dsgs
  combinedUniDSGVector_.resize(numGrids);
  for(int g=0; g<numGrids; g++){
      combinedUniDSGVector_[g] = new
       ↪  DistributedSparseGridUniform<CombiDataType>
                      (dim, lmax, lmin, boundary,
                       ↪  theMPISystem()->getLocalComm());
    }
    // todo: move to init function to avoid reregistering
    // register dsgs in all dfgs
    for (Task* t : tasks_) {
      for(int g=0; g<numGrids; g++){

        DistributedFullGrid<CombiDataType>& dfg =
         ↪  t->getDistributedFullGrid(g);

        dfg.registerUniformSG(*(combinedUniDSGVector_[g]));
      }
    }
    Stats::stopEvent("combine init");
    Stats::startEvent("combine hierarchize");

    real localMax(0.0);
    //std::vector<CombiDataType> beforeCombi;
    for (Task* t : tasks_) {
      t->fullgridVectorBeforeCombi = new
       ↪  std::vector<CombiDataType>[numGrids];
      for(int g=0; g<numGrids; g++){

        DistributedFullGrid<CombiDataType>& dfg =
         ↪  t->getDistributedFullGrid(g);
        std::vector<CombiDataType> datavector(dfg.getElementVector());
        t->fullgridVectorBeforeCombi[g] = datavector;
        //beforeCombi = datavector;
        // compute max norm
        /*
        real max = dfg.getLpNorm(0);
        if( max > localMax )
          localMax = max;
          */

        // hierarchize dfg
        DistributedHierarchization::hierarchize<CombiDataType>(
            dfg, combiParameters_.getHierarchizationDims() );

        // lokales reduce auf sg ->
        dfg.addToUniformSG( *combinedUniDSGVector_[g],
                            combiParameters_.getCoeff( t->getID() ) );
#ifdef DEBUG_OUTPUT
        std::cout << "Combination: added task " << t->getID() <<
        " with coefficient " << combiParameters_.getCoeff( t->getID()
         ↪  ) <<"\n";
#endif
      }
```

```
  }
  Stats::stopEvent("combine hierarchize");

  Stats::startEvent("combine global reduce init");

  Task::bufAsync = new std::vector<CombiDataType>[numGrids];
  Task::requestAsync = new MPI_Request[numGrids];
  Task::subspaceSizes = new std::vector<int>[numGrids];

  for(int g=0; g<numGrids; g++){
    CombiCom::distributedGlobalReduceAsyncInit(
     ↪  *combinedUniDSGVector_[g],
        Task::subspaceSizes[g], Task::bufAsync[g],
         ↪  Task::requestAsync[g]);
  }
  Stats::stopEvent("combine global reduce init");

  Stats::startEvent("combine dehierarchize");

  for (Task* t : tasks_) {
    for(int g=0; g<numGrids; g++){

      // get handle to dfg
      DistributedFullGrid<CombiDataType>& dfg =
       ↪  t->getDistributedFullGrid(g);


      // dehierarchize dfg
      DistributedHierarchization::dehierarchize<CombiDataType>(
          dfg, combiParameters_.getHierarchizationDims() );
    }
  }
  Stats::stopEvent("combine dehierarchize");
}

void ProcessGroupWorker::combineUniformAsyncHierarchizeUpdate(){
  //std::vector<CombiDataType> afterCombi;
  Stats::startEvent("combine global reduce extract");
  int numGrids = combiParameters_.getNumGrids();

  for(int g=0; g<numGrids; g++){
    CombiCom::distributedGlobalReduceAsyncExtractSubspace(
      *combinedUniDSGVector_[g], Task::subspaceSizes[g],
       ↪  Task::bufAsync[g] );
  }
  Stats::stopEvent("combine global reduce extract");

  Stats::startEvent("combine dehierarchize");
  for (Task* t : tasks_) {
    for(int g=0; g<numGrids; g++){

      // get handle to dfg
      DistributedFullGrid<CombiDataType>& dfg =
       ↪  t->getDistributedFullGrid(g);
      //t->prevTimeStepDfg = t->getDistributedFullGrid(g);
      std::vector<CombiDataType>
       ↪  gridNextTimestep(dfg.getElementVector());
      // extract dfg vom dsg
      dfg.extractFromUniformSG( *combinedUniDSGVector_[g] );

      // dehierarchize dfg
      DistributedHierarchization::dehierarchize<CombiDataType>(
          dfg, combiParameters_.getHierarchizationDims() );

      std::vector<CombiDataType>& gridAfterCombi =
       ↪  dfg.getElementVector();

      for(int i=0; i< gridAfterCombi.size();i++){
        gridAfterCombi[i] += (gridNextTimestep[i]
                                      - t->fullgridVectorBeforeCombi[g][i]);
      }
    }
  }

  for(Task* t : tasks_){
    delete [] t->fullgridVectorBeforeCombi;
  }
  delete [] Task::subspaceSizes;
  delete [] Task::bufAsync;
  delete [] Task::requestAsync;

  Stats::stopEvent("combine dehierarchize");
}
```

A.2. CombiCom

```
  template<typename FG_ELEMENT> void
   ↪  CombiCom::distributedGlobalReduceAsyncInit(
  DistributedSparseGridUniform<FG_ELEMENT>& dsg,
  std::vector<int>& subspaceSizes,
  std::vector<FG_ELEMENT>& bufAsync, MPI_Request &requestAsync) {
  // get global communicator for this operation
  MPI_Comm mycomm = theMPISystem()->getGlobalReduceComm();

  assert(mycomm != MPI_COMM_NULL);

  /* get sizes of all partial subspaces in communicator
   * we have to do this, because size information of uninitialized
↪  subspaces
```

```
   * is not available in dsg. at the moment this information is only available
   * in dfg.
   */
   subspaceSizes.resize(dsg.getNumSubspaces());

  for (size_t i = 0; i < subspaceSizes.size(); ++i) {
    // MPI does not have a real size_t equivalent. int should work in most cases
    // if not we can at least detect this with an assert
    assert(dsg.getDataSize(i) <= INT_MAX);

    subspaceSizes[i] = int(dsg.getDataSize(i));
  }

  MPI_Allreduce( MPI_IN_PLACE, subspaceSizes.data(), int(subspaceSizes.size()),
                 MPI_INT, MPI_MAX, mycomm);

  // check for implementation errors, the reduced subspace size should not be
  // different from the size of already initialized subspaces
  int bsize = 0;

  for (size_t i = 0; i < subspaceSizes.size(); ++i) {
    bool check = (subspaceSizes[i] == 0 || dsg.getDataSize(i) == 0
                  || subspaceSizes[i] == int(dsg.getDataSize(i)));

    if (!check) {
      int rank;
      MPI_Comm_rank( MPI_COMM_WORLD, &rank);
      std::cout << "l = " << dsg.getLevelVector(i) << " " << "rank = " << rank
                << " " << "ssize = " << subspaceSizes[i] << " " << "dsize = "
                << dsg.getDataSize(i) << std::endl;
      assert(false);
    }

    bsize += subspaceSizes[i];
  }

  // put subspace data into buffer
  bufAsync.clear();
  bufAsync.resize(bsize,FG_ELEMENT(0));
  {
    typename std::vector<FG_ELEMENT>::iterator buf_it = bufAsync.begin();

    for (size_t i = 0; i < dsg.getNumSubspaces(); ++i) {
      std::vector<FG_ELEMENT>& subspaceData = dsg.getDataVector(i);

      // if subspace does not exist on this process this part of the buffer is
      // left empty
      if (subspaceData.size() == 0) {
        buf_it += subspaceSizes[i];
        continue;
      }

      for (size_t j = 0; j < subspaceData.size(); ++j) {
        *buf_it = subspaceData[j];
        ++buf_it;
      }
    }
  }

  MPI_Datatype dtype = abstraction::getMPIDatatype(
                                  abstraction::getabstractionDataType<FG_ELEMENT>()
  MPI_Iallreduce( MPI_IN_PLACE, bufAsync.data(), bsize, dtype, MPI_SUM, mycomm,
                  &requestAsync);

}

template<typename FG_ELEMENT>
void CombiCom::distributedGlobalReduceAsyncExtractSubspace(
  DistributedSparseGridUniform<FG_ELEMENT>& dsg,
  std::vector<int>& subspaceSizes,
  std::vector<FG_ELEMENT>& bufAsync) {
  // extract subspace data

    typename std::vector<FG_ELEMENT>::iterator buf_it = bufAsync.begin();

    for (size_t i = 0; i < dsg.getNumSubspaces(); ++i) {
      std::vector<FG_ELEMENT>& subspaceData = dsg.getDataVector(i);

      // this is very unlikely but can happen if dsg is different than
      // lmax and lmin of combination scheme
      if(subspaceData.size() == 0 && subspaceSizes[i] == 0)
        continue;

      // this happens for subspaces that are only available in component grids
      // on other process groups
      if( subspaceData.size() == 0 && subspaceSizes[i] > 0 ){
        subspaceData.resize( subspaceSizes[i] );
      }

      // wenn subspaceData.size() > 0 und subspaceSizes > 0
      for (size_t j = 0; j < subspaceData.size(); ++j) {
        subspaceData[j] = *buf_it;
        ++buf_it;
      }
    }
  }
} /* namespace combigrid */
```

A.3. AdvectionExample

```
int main(int argc, char** argv) {
  MPI_Init(&argc, &argv);

  /* when using timers (TIMING is defined in Stats), the Stats class
   ↪  must be
   * initialized at the beginning of the program. (and finalized in
↪  the end)
   */
  Stats::initialize();

  // read in parameter file
  boost::property_tree::ptree cfg;
  boost::property_tree::ini_parser::read_ini("ctparam", cfg);

  // number of process groups and number of processes per group
  size_t ngroup = cfg.get<size_t>("manager.ngroup");
  size_t nprocs = cfg.get<size_t>("manager.nprocs");

  // divide the MPI processes into process group and initialize the
  // corresponding communicators
  theMPISystem()->init( ngroup, nprocs );

  // this code is only executed by the manager process
  WORLD_MANAGER_EXCLUSIVE_SECTION {
    /* create an abstraction of the process groups for the manager's
     ↪  view
     * a pgroup is identified by the ID in gcomm
     */
    ProcessGroupManagerContainer pgroups;
    for (size_t i = 0; i < ngroup; ++i) {
      int pgroupRootID(i);
      pgroups.emplace_back(
          std::make_shared< ProcessGroupManager > ( pgroupRootID )
                              );
    }

    // create load model
    LoadModel* loadmodel = new LinearLoadModel();

  /* read in parameters from ctparam */
  DimType dim = cfg.get<DimType>("ct.dim");
  LevelVector lmin(dim), lmax(dim), leval(dim);
  IndexVector p(dim);
  combigrid::real dt;
  size_t nsteps, ncombi;
  cfg.get<std::string>("ct.lmin") >> lmin;
  cfg.get<std::string>("ct.lmax") >> lmax;
  cfg.get<std::string>("ct.leval") >> leval;
  cfg.get<std::string>("ct.p") >> p;
  ncombi = cfg.get<size_t>("ct.ncombi");
  dt = cfg.get<combigrid::real>("application.dt");
  nsteps = cfg.get<size_t>("application.nsteps");

  // todo: read in boundary vector from ctparam
  std::vector<bool> boundary(dim, true);

  // check whether parallelization vector p agrees with nprocs
  IndexType checkProcs = 1;
  for (auto k : p)
    checkProcs *= k;
  assert(checkProcs == IndexType(nprocs));

  /* generate a list of levelvectors and coefficients
   * CombiMinMaxScheme will create a classical combination scheme.
   * however, you could also read in a list of levelvectors and
↪ coefficients
   * from a file */
  CombiMinMaxScheme combischeme(dim, lmin, lmax);
  combischeme.createAdaptiveCombischeme();
  std::vector<LevelVector> levels = combischeme.getCombiSpaces();
  std::vector<combigrid::real> coeffs = combischeme.getCoeffs();

  // output combination scheme
  std::cout << "lmin = " << lmin << std::endl;
  std::cout << "lmax = " << lmax << std::endl;
  std::cout << "CombiScheme: " << std::endl;
  std::cout << combischeme << std::endl;

  // create Tasks
  TaskContainer tasks;
  std::vector<int> taskIDs;
  for (size_t i = 0; i < levels.size(); i++) {
    Task* t = new TaskExample(dim, levels[i], boundary, coeffs[i],
                              loadmodel, dt, nsteps, p);
    tasks.push_back(t);
    taskIDs.push_back( t->getID() );
  }

  // create combiparameters
```

```
    CombiParameters params(dim, lmin, lmax, boundary, levels,
     ↪  coeffs, taskIDs,
                              ncombi, 1 );
    params.setParallelization(p);
    // create abstraction for Manager
    ProcessManager manager(pgroups, tasks, params);

    // the combiparameters are sent to all process groups before the
    // computations start
    manager.updateCombiParameters();

    std::cout << "set up component grids and run until first
     ↪  combination point"
              << std::endl;

    /* distribute task according to load model and start computation
     ↪  for
     * the first time */
    Stats::startEvent("manager run first");
    manager.runfirst();
    Stats::stopEvent("manager run first");

    for (size_t i = 0; i < ncombi; ++i) {
      Stats::startEvent("combineAsync");
      manager.combineAsync();
      Stats::stopEvent("combineAsync");

      // evaluate solution and
      // write solution to file
      std::string filename("out/solution_" + std::to_string(ncombi)
       ↪  + ".dat" );
      Stats::startEvent("manager write solution");
      manager.parallelEval( leval, filename, 0 );
      Stats::stopEvent("manager write solution");

      std::cout << "run until combination point " << i+1 <<
       ↪  std::endl;

      // run tasks for next time interval
      Stats::startEvent("manager run");
      manager.runnext();
      Stats::stopEvent("manager run");
    }

  // send exit signal to workers in order to enable a clean program
   ↪  termination
    manager.exit();
  }

  // this code is only execute by the worker processes
  else {
    // create abstraction of the process group from the worker's
    ↪  view
    ProcessGroupWorker pgroup;

    // wait for instructions from manager
    SignalType signal = -1;

    while (signal != EXIT)
      signal = pgroup.wait();
  }

  Stats::finalize();

  /* write stats to json file for postprocessing */
  Stats::write( "timers.json" );

  MPI_Finalize();

  return 0;
}
```

B. Input Parameters

B.1. Advection-Diffusion

The *ctparam* file containing the input parameters consist of three sections namely: **ct**, **application** and **manager**. All three sections contain values in agreement with their names. The **ct** contains the *combination technique* parameters, namely *dim* which specifies the dimension of the problem, $\vec{l}_{\text{min}}$ and $\vec{l}_{\text{max}}$ which specify the minimum and the maximum size of the level vectors along each dimension. The parameter *leval* is used to specify the level at which the full grid evaluation should take place. Finally, parameter *p* denotes the parallelization vector while *ncombi* specifies the number of combinations that should occur in that test sequence. The **application** section consists of *dt*. which is the euler time step ΔT to be used in calculation phase, and *nsteps* which determines how many of such calculation steps should be run between each combination steps. The **manger** section contain *ngroup* and *nprocs* which denotes the number of groups and processes per each group respectively.

```
ctparam
[ct]
dim = 2
lmin = 3 3
lmax = 10 10
leval = 5 5
p = 1 2
ncombi = 10

[application]
dt = 1e-3
nsteps = 100

[manager]
ngroup = 2
nprocs = 2
```

Figure B.1.: Adevction-Diffusion example ctparam file

B.2. GENE

The *ctparam* file of GENE has two additional sections **preproc** and **faults** in addition to the previous three sections of **ct**, **application** and **manager**. The parameters enclosed in the **ct** section include: *readspaces*, which denotes how the rest of the parameters in the file should be read. *dim* which corresponds to the number of dimensions of the problem. *lmin* and *lmax* which correspond to $\vec{l}_{\text{min}}$ and $\vec{l}_{\text{max}}$ respectively. While, *leval* and *leval2* state the level vectors to be used for two separate full grid evaluation after the sparse grid computation. The parameters *fg_file_path* and *fg_file_path2* specify where the respective evaluation should be stored. parameter *p* denotes number of processes to be used per dimension, whereas *boundary* indicates which dimension has boundary points, and *hierarchization_dims* is used to establish if the following dimension would be hierarchized or not. *ncombi* specifies the total number of combination steps. The are also parameters *reduceCombinationDimsLmin* and *reduceCombinationDimsLmax* determine the maximum and the minimum extent of the dimension to be used to facilitate reduced combination.

The section **application** contains parameters pertaining to GENE. *dt*. which is the euler time step ΔT which is considered only if its a linear simulation where the adaptivity is switched off. *nsteps* corresponds to the number of calculation steps that should take place between each combination steps. Its also possible to set time between each combinations instead of fixed steps using the parameter *combitime*. the parameters *shat*, *kymin* and *lx* correspond to $\hat{s}$, $k_{y_{min}}$ and l_x respectively. *numspecies* denotes number of species to be used in the simulation. While, *GENE_local* and *GENE_nonlinear* are used to flag if the simulation is local and
or non-linear.

The **preproc** sections contains the location of the necessary libraries and executables to be used by GENE. *basename* specifies the name to be used as the root name for all the folders that would be created as a result of simulation. *executable* provides the location the gene executable file. *mpi* marks the kind of MPI module that should be used. *sgpplib* and *tasklib* are used to point towards the libraries pertaining to sgpp and the local tasks respectively. Finally, *startscript* points to the batch file that would initiate the entire simulation.

The **manager** section like before, states the number of process groups and processes to be used in the simulation via *ngroup* and *nprocs* respectively.

The **faults** section like the name suggest contains parameters pertaining to the fault tolerance segment. *num_faults* fixes the number of faults that is acceptable. Finally, the parameters *iteration_faults* and *global_rank_faults* are used to denote the vector of time steps at which processes fail and global rank of process that fails respectively.

```
ctparam
[ct]
#last element has to be 1 -> specify species with special field
dim = 6
lmin = 3 1 4 4 4 1
lmax = 3 1 13 13 13 1
leval = 3 1 4 4 4 1
leval2 = 3 1 13 13 13 1
p = 1 1 1 8 8 1
ncombi = 200
readspaces = 1
fg_file_path = ../plot.dat
fg_file_path2 = ../plot2.dat
boundary = 1 0 1 1 1 0
hierarchization_dims = 0 0 1 1 1 0
reduceCombinationDimsLmin = 0 0 0 0 0 0
reduceCombinationDimsLmax = 0 0 1 1 1 0

[application]
dt = 0.005
nsteps = 1
combitime = 10000000
shat = 0.7960
kymin = 0.3000
lx = 4.18760
numspecies = 1
GENE_local = T
GENE_nonlinear = F

[preproc]
basename = ginstance
executable = ./gene_new_machine
mpi = mpiexec
sgpplib = $HOME/sgpp
tasklib = $HOME/lib
startscript = start.bat

[manager]
ngroup = 4
nprocs = 64

[faults]
num_faults = 0
iteration_faults = 2 4 7 8
global_rank_faults = 2 1 7 0
```

Figure B.2.: GENE example ctparam file

C. Additional Plots

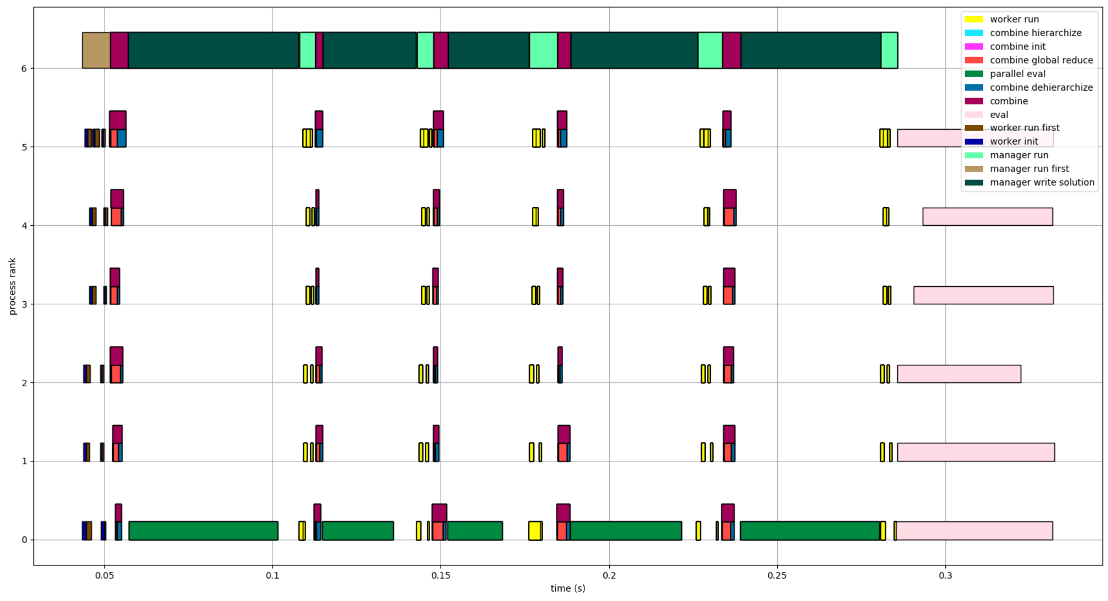


Figure C.1.: Timeline of all processes of normal combination technique

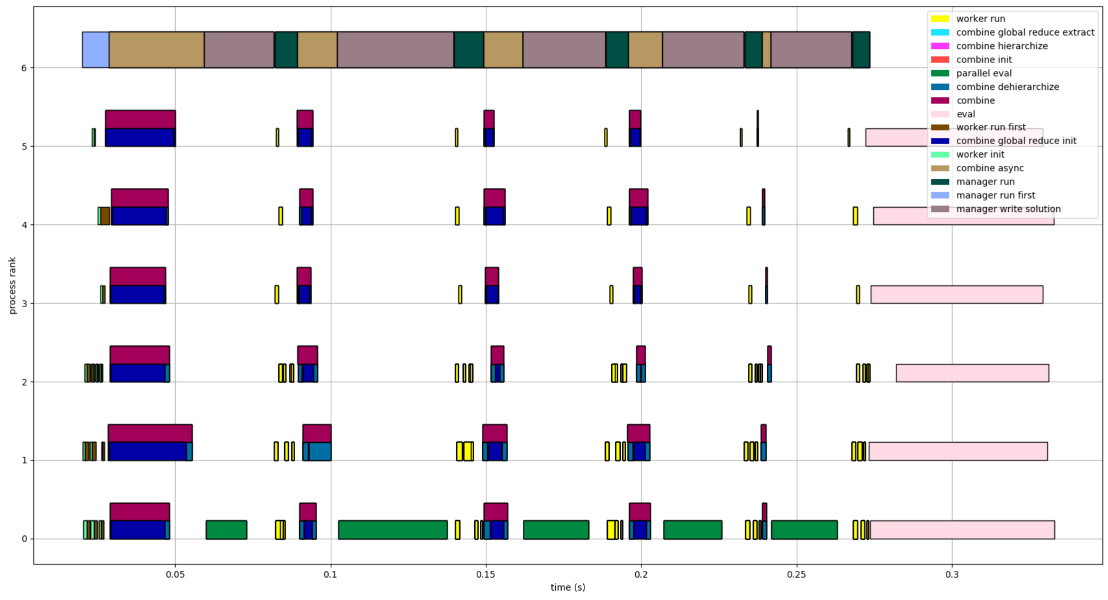


Figure C.2.: Timeline of all processes of asynchronous combination technique

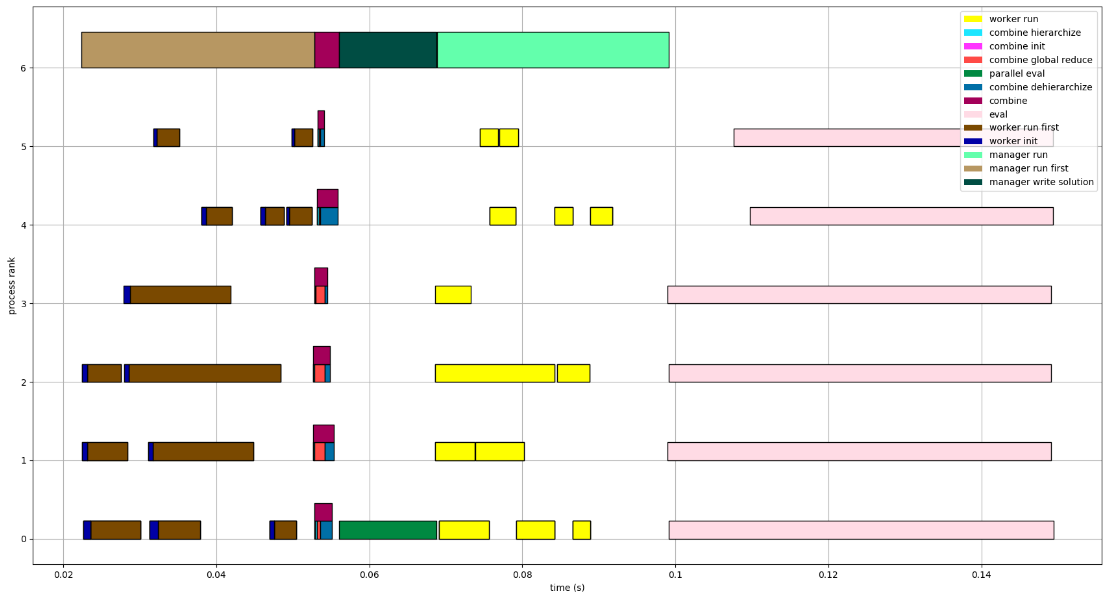


Figure C.3.: Timeline of all processes normal combination technique of only one combination step

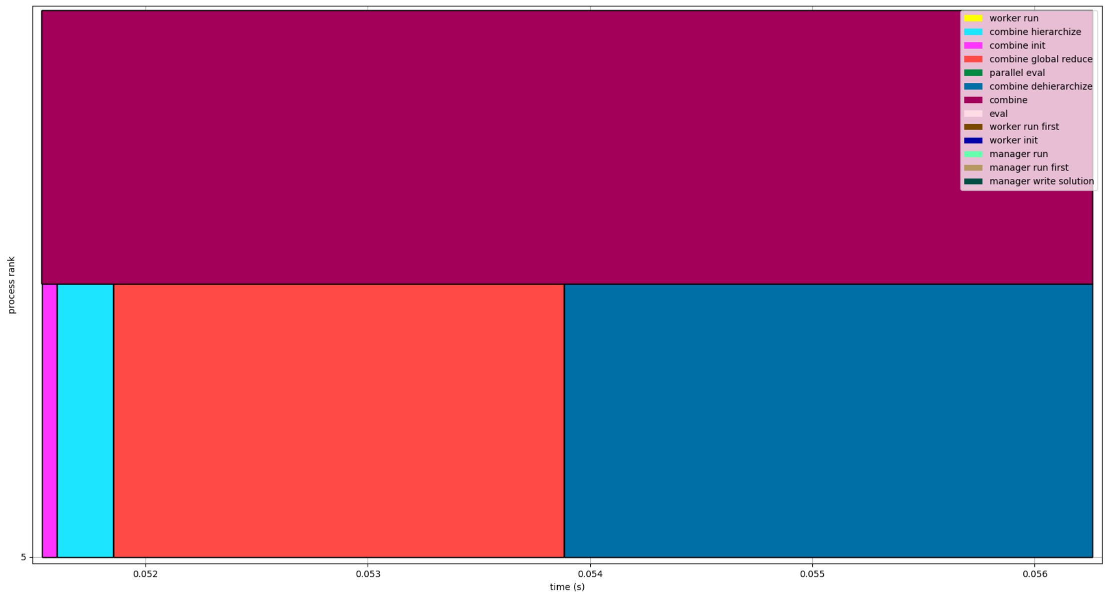


Figure C.4.: Timeline of one normal combination step

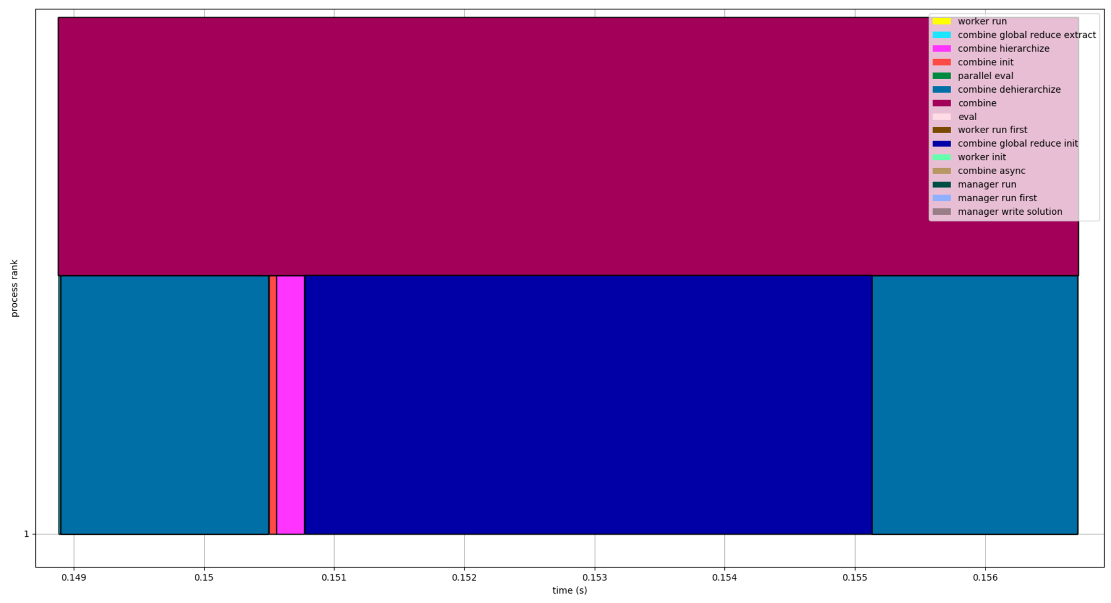


Figure C.5.: Time line of one asynchronous combination Step

Bibliography

[1] H Bungartz, Michael Griebel, and Ulrich Rüde. Extrapolation, combination, and sparse grid techniques for elliptic boundary value problems. *Computer methods in applied mechanics and engineering*, 116(1-4):243–252, 1994.

[2] Hans-Joachim Bungartz, M Griebel, D Röschke, and C Zenger. *Pointwise convergence of the combination technique for Laplace's equation*. Technische Universität München. Institut für Informatik, 1993.

[3] Hans-Joachim Bungartz and Michael Griebel. Sparse grids. *Acta numerica*, 13:147–269, 2004.

[4] Franck Cappello, Al Geist, Bill Gropp, Laxmikant Kale, Bill Kramer, and Marc Snir. Toward exascale resilience. *The International Journal of High Performance Computing Applications*, 23(4):374–388, 2009.

[5] Franck Cappello, Al Geist, William Gropp, Sanjay Kale, Bill Kramer, and Marc Snir. Toward exascale resilience: 2014 update. *Supercomputing frontiers and innovations*, 1(1):5–28, 2014.

[6] Jack Dongarra, Pete Beckman, Terry Moore, Patrick Aerts, Giovanni Aloisio, Jean-Claude Andre, David Barkai, Jean-Yves Berthou, Taisuke Boku, Bertrand Braunschweig, et al. The international exascale software project roadmap. *International Journal of High Performance Computing Applications*, 25(1):3–60, 2011.

[7] Georg Faber. Über stetige funktionen. *Mathematische Annalen*, 66(1):81–94, 1908.

[8] Christian Feuersänger. *Sparse grid methods for higher dimensional approximation.* PhD thesis, University of Bonn, 2010.

[9] Jochen Garcke. Sparse grids in a nutshell. In *Sparse grids and applications*, pages 57–80. Springer, 2012.

[10] Jochen Garcke et al. Sparse grid tutorial. *Mathematical Sciences Institute, Australian National University, Canberra Australia*, page 7, 2006.

[11] Thomas Gerstner and Michael Griebel. Numerical integration using sparse grids. *Numerical algorithms*, 18(3-4):209, 1998.

[12] Richard L Graham, Timothy S Woodall, and Jeffrey M Squyres. Open mpi: A flexible high performance mpi. In *International Conference on Parallel Processing and Applied Mathematics*, pages 228–239. Springer, 2005.

[13] Michael Griebel. *A parallelizable and vectorizable multi-level-algorithm on sparse grids*. Technische Universität, 1990.

[14] Michael Griebel, Michael Schneider, and Christoph Zenger. *A combination technique for the solution of sparse grid problems*. Citeseer, 1990.

[15] William Gropp and Ewing Lusk. User's guide for mpich, a portable implementation of mpi, 1996.

[16] William Gropp, Ewing Lusk, Nathan Doss, and Anthony Skjellum. A high-performance, portable implementation of the mpi message passing interface standard. *Parallel computing*, 22(6):789–828, 1996.

[17] Mario Heene. A massively parallel combination technique for the solution of high-dimensional pdes. 2018.

[18] Pieter W Hemker. Application of an adaptive sparse-grid technique to a model singular perturbation problem. *Computing*, 65(4):357–378, 2000.

[19] Willem Hundsdorfer and Jan G Verwer. *Numerical solution of time-dependent advection-diffusion-reaction equations*, volume 33. Springer Science & Business Media, 2013.

[20] Dirk Pflüger. *Spatially Adaptive Sparse Grids for High-Dimensional Problems*. Verlag Dr. Hut, München, August 2010.

[21] Dirk Pflüger, Benjamin Peherstorfer, and Hans-Joachim Bungartz. Spatially adaptive sparse grids for high-dimensional data-driven problems. *Journal of Complexity*, 26(5):508–522, 2010.

[22] Dirk Michael Pflüger. *Spatially adaptive sparse grids for high-dimensional problems*. PhD thesis, Technische Universität München, 2010.

[23] Sayantan Sur, Matthew J Koop, and Dhabaleswar K Panda. High-performance and scalable mpi over infiniband with reduced memory usage: an in-depth performance analysis. In *Proceedings of the 2006 ACM/IEEE conference on Supercomputing*, page 105. ACM, 2006.

[24] Harry Yserentant. On the multi-level splitting of finite element spaces. *Numerische Mathematik*, 49(4):379–412, 1986.

[25] Christoph Zenger and W Hackbusch. Sparse grids. 1991.