IMPROVING PERFORMANCE OF ITERATIVE SOLVERS ON MODERN ARCHITECTURES

BY

LUKAS SPIES

DISSERTATION

Submitted in partial fulfillment of the requirements
for the degree of Doctor of Philosophy in Computer Science
in the Graduate College of the
University of Illinois Urbana-Champaign, 2024

Urbana, Illinois

ABSTRACT

Over the past decade or two massive changes have occurred both in terms of hardware and software for high performance computing. Large heterogeneous machines are commonly in use today, presenting new challenges for scientific algorithms. In this thesis we will focus on the performance of iterative algorithms and explore several different aspects of working on modern architectures. In the first part we present a novel halo exchange library that is designed specifically for modern heterogeneous architectures and illustrate how it is not only easy to use but also flexible and, most importantly, highly performant. In the second chapter we consider various relaxation schemes for preconditioning a GMRES solver for the Stokes equations, with a particular focus on their performance on GPUs. We present a few different schemes but mostly focus on two of them, Vanka and Braess-Sarazin. We show how, when carefully designed, Vanka is capable of outperforming Braess-Sarazin on the GPU, something that to our knowledge has never been achieved before. In the final part we move from the Stokes equations to the Reynolds-Averaged Navier-Stokes equations that arise in the context of wind turbine modeling. Our focus is on an algorithm that has been of renewed interest in recent years, restricted additive Schwarz (RAS) paired with ILU. After analyzing our implementation of RAS and ILU, we design a new solver that incorporates RAS+ILU as relaxation scheme for an AMG cycle. The AMG cycle is then used as preconditioning for some of the GMRES solves as part of a new solver we design to solve the RANS equations. We conclude by extending our solver with homotopy, making it capable of self-tuning for finding a possible continuation path for solving hard problems.

ACKNOWLEDGMENTS

This thesis would not have been possible without the help and support of so many people. First, I want to thank my advisor, Dr. Luke Olson, for his support, guidance, and patience throughout my studies. I also want to thank my Masters advisor Dr. Scott MacLachlan for his helpful insights and collaboration. I am also grateful to Dr. Andrew Reisner, Dr. Amanda Bienz, Dr. David Moulton, Dr. Paul Fischer, Dr. Alexey Voronin, and so many others for being willing to collaborate or lend an ear to listen and give feedback to various ideas. I want to thank all my family and friends that encouraged me in difficult times, always pretended to be interested in my latest research even if they did not understand it. And most of all I want to thank my fabulous and amazing wife, Amanda Foster, the love of my life. I would never have made it to the end without her unending moral support and encouragement. This thesis is a testament to all the amazing people mentioned here and so many beyond. I have been blessed by having met so many amazing people and see so much of the world throughout the past few years.

TABLE OF CONTENTS

CHAPTER 1: INTRODUCTION

The focus of this work is on improving performance of algorithms, in particular iterative solvers, on modern architectures used to solve the Stokes and Navier-Stokes equations. This thesis consists of three parts: First, we present a new halo exchange library targeting large heterogeneous machines. Second, we investigate various relaxation schemes with a particular focus on Vanka and Braess-Sarazin, and explore how utilizing existing structure in the discretization allows us to take advantage of better memory access patterns. Lastly, we explore the Reynolds-Averaged Navier-Stokes (RANS) equations and provide an overview of existing work on solving these equations and their limitations in the context of wind turbine modeling. We develop and analyze a new solver based on existing and new strategies to obtain a reliable and flexible solver, with a particular focus on AMG paired with Restricted Additive Schwarz (RAS) and ILU to precondition some of our linear solves.

1.1 DATA TRANSFER ACROSS LARGE HETEROGENEOUS MACHINES

Exchanging data on supercomputers (typically in the form of halo data) is an important part of many algorithms (in particular iterative algorithms) that make use of distributed memory. Solving this task often involves code written specifically for some application, duplicating the work for every new application. In order to achieve good performance, the code needs to be carefully hand-tuned to take advantage of some more advanced capabilities of MPI and the underlying hard- and software, introducing an increasing risk of bugs and costing time. We discuss different techniques and novel ideas in order to maximize performance of data transfer across both CPUs and GPUs using MPI, either on its own or in combination with CUDA, HIP, and OpenCL. These techniques are implemented inside a halo exchange library called *Tausch* that is a drop-in solution for existing code and easy to work with for new code. We highlight the benefits of using Tausch by both real-world applications and a performance model.

1.2 SMOOTHERS FOR STOKES EQUATIONS

The Stokes equations arise from a linearization of the Navier-Stokes equations, modeling fluid flow with small advective inertial forces relative to viscous forces and typically have a small Reynolds number, $Re \ll 1$. Using GMRES as a solver for the Stokes equations, we consider various different types of relaxation schemes used within a multigrid preconditioner, with a particular focus on Braess-Sarazin and Vanka. We illustrate how we are able to take advantage of a highly structured discretization and data structures that are implemented in a highly structured way to improve the overall performance. We show how Vanka is capable of outperforming Braess-Sarazin and how it is indeed a competitive algorithm, in particular on the GPU.

1.3 RAS+ILU FOR THE REYNOLDS-AVERAGED NAVIER-STOKES EQUATIONS

The Reynolds-Averaged Navier-Stokes equations are time-averaged equations that model the motion for fluid flow, in particular describing turbulent flows. They are based on the concept of Reynolds decomposition by decomposing the Navier-Stokes equations into a fluctuating and time-averaged (mean) quantity. The RANS equations include a nonlinear Reynolds stress term that yields a system that is not closed, but requires so-called closure models to model this quantity. For the purposes of this study we assume the Reynolds-stress tensor to be zero (thus we essentially obtain the Navier-Stokes equations) to allow us to focus on our preconditioner and overall solver development. We explore these equations in the context of wind farms and illustrate how wind flows around a wind turbine and the turbulences that arise from such wind flow. The turbulences add additional difficulties to the underlying equations, as they introduce shock-like features in the system. Additionally, the resulting equations come with large Reynolds numbers, typically $O(10^2)$ to $O(10^3)$. We give an overview of some of the commonly used solvers and explore their suitability in solving our model problem. We then design a new solver that incorporates RAS paired with ILU for preconditioning some of the linear solves and analyze its performance. We explore the use of homotopy or continuation methods to improve the performance of this solver and to help us push our solver to solving more difficult problems than we could do without this approach. Lastly, we combine everything we learned into the development of a dynamic solver that is capable of self-tuning its homotopy parameters to find a working continuation path from an initial problem to the actual target problem.

CHAPTER 2: DATA TRANSFER ACROSS LARGE HETEROGENEOUS MACHINES

Halo exchanges are a part of many if not most applications and often proves to be a bottleneck. There are many existing solutions implementing such halo exchanges in various contexts, languages, and programming models. However, most of them are limited to either their specific programming model (e.g., Kokkos [1]), use case (e.g., DTK [2]), or are simply out-of-date (e.g., GCL [3]).

2.1 REQUIREMENTS

We identified four design requirements that are essential for a halo exchange library:

1. Ease of use: It should be straight forward to be incorporated into existing code or added on to new code.
2. Flexible: It should support any type of geometry and data, ideally allowing for different data types to be combined into single messages.
3. Heterogeneous: It should support both CPUs and GPUs, and any combination of thereof.
4. Performant: The exchange operation should target efficiency, and performance expectations should be clearly defined.

2.2 EXISTING WORK

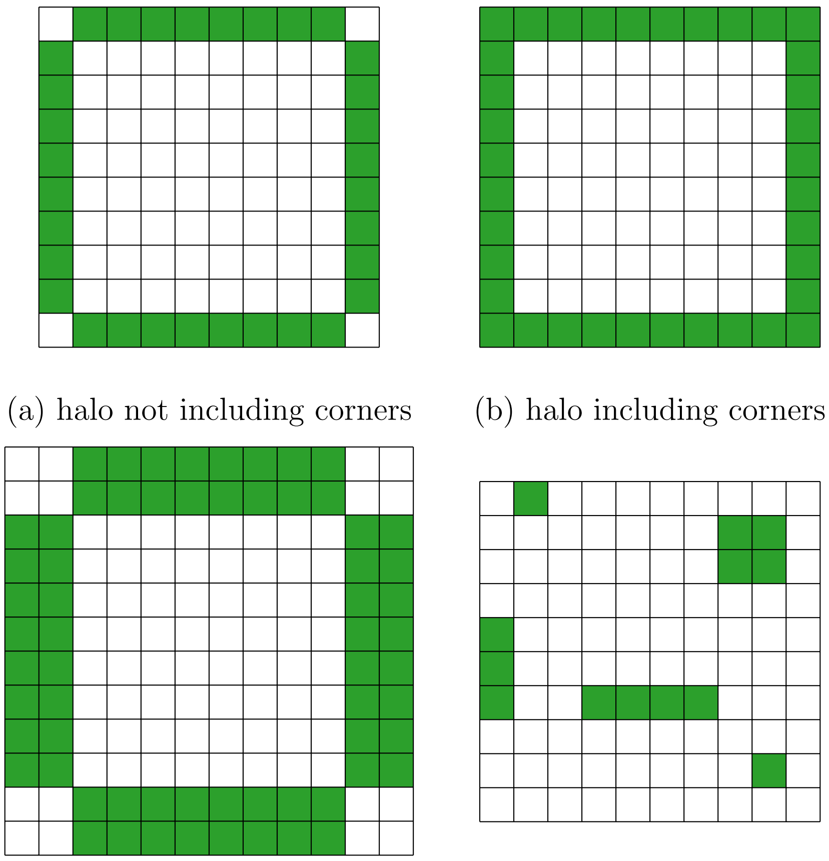
There are several existing solutions for communicating halos. We first discuss several tools that address halo exchanges in a generic way, allowing for their integration into any user code. Then we mention several existing frameworks that include their own halo handling. Most of the existing generic tools are targeted towards a specific use case or situation, with some no longer maintained. The Data Transfer Kit (DTK) [2] is designed primarily for physics applications, where geometric domains may not conform to the same physical space, potentially with mismatched parallel decomposition. These features are valuable when needed, but they also introduce unnecessary complexity. The Generic Communication Layer (GCL) [3] is a library of communication patterns where the halo exchange operation is divided into different layers that can be tweaked and updated independently. It is a templated header-only C++ library and allows for flexibility in how it can be used, leading to a more complex API. GCL is described as "old code" in its GitHub repository [4], with its last update in 2017. We are not aware of any applications making use of GCL.

Raja [5, 6] is a library of C++ software abstractions aiming to enable architecture and programming model portability for high performance applications. It also provides constructs for efficient packing and unpacking of data on different computing devices, although it does not facilitate any actual communication. Tempi [7] is another approach that specifically targets MPI+CUDA, improving the performance of MPI using CUDA buffers. This design is achieved through MPI and can thus be easily combined with other tools and libraries that use MPI. A different approach is taken by Kokkos [1], where a new programming model is developed that offers local mapping and execution on a variety of architectures. It does not handle halo exchanges and defers to other codes and frameworks for those. Tpetra [8] is a package for Trilinos [9] implementing linear algebra objects that uses Kokkos for local operations and provides the necessary code for facilitating halo exchanges. PETSc also provides its own handling of halo exchanges as part of its distributed arrays (DMDA). All of these come with their own programming models and require the user's code to adapt to that. Thus, they require the use of their own custom data structures and also typically necessitate large code rewrites. Finally, the MiniGhost [10, 11] application in the Mantevo Project [12] is written in Fortran and serves as stand-alone code to explore and experiment different programming models. It was last updated in 2016.

2.3 OUR SOLUTION

We developed a new halo exchange library (called *Tausch*) that introduces several novel ideas to achieve the goals set out above. It provides a high-level API for halo exchanges using MPI [13], CUDA [14], HIP [15], and OpenCL [16]. The user determines the traffic pattern, where the data to be sent lives in memory and where the received data is to be written to in memory. Tausch then offers various strategies to achieve a high-performance exchange of the specified data, whether the data comprises a halo or any other type of data. It is a header-only library, thus relieves the user of having to precompile and link to an additional library, and it allows for maximum inlining of its member functions. It is written in C++ with a fully compatible C API, and a Fortran interface is also available.

To begin, we first define the notion of a *halo* in the exchange of data. A halo is any structured or unstructured area that is used but typically not owned by the local process. In most applications a halo would lie along the edges of a domain, though this is not a requirement for Tausch. We refer to data that needs to be sent to another partition's halo region as the *send halo* and, conversely, data that needs to be updated locally with values received from another partition as the *receive halo.* fig. 2.1 illustrates various types of halos, both structured and unstructured, all of which can be handled by Tausch.


(a) halo not including corners (b) halo including corners

(c) halo of width 2 (d) unstructured halo

Figure 2.1: Examples of different halos, with the halos highlighted in green.

2.4 OPTIMIZATION STRATEGIES

Tausch employs several strategies to improve the performance of halo exchanges. First, it uses a compressed format to store halo information instead of storing a full vector indices. The compressed storage used by Tausch is optimized for structured halo regions, however, it will work for any halo region form or shape.

The user can either directly specify the halo regions using the compressed format, or make use of a convenience function that takes in a set of indices of halo data and converts it into the compressed format. In the latter case, Tausch decomposes that region up into rectangular subregions corresponding to how the data is laid out in memory. Such a region does not necessarily translate to a rectangular region in the physical setup. Each such subregion is defined using these 4 integers (see fig. 2.2):

1. Starting index of the region;
2. number of consecutive values (i.e., number of columns);
3. frequency of consecutive values (i.e., number of rows); and
4. stride between the sets of consecutive values.

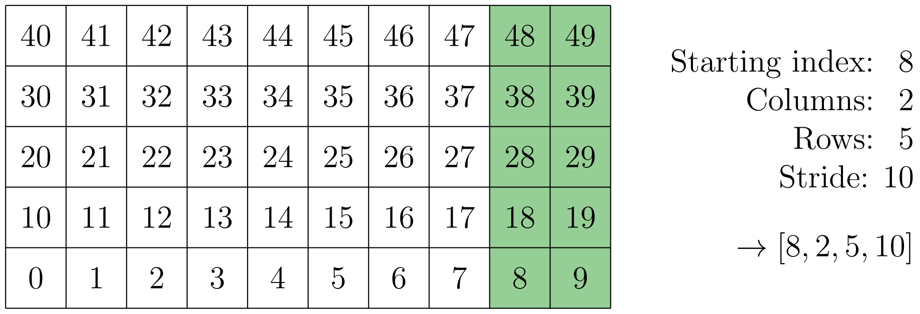


Figure 2.2: Example of compressed storage: 10 integers (40 bytes) stored using 4 integers (16 bytes), halo region highlighted in green.

Using a compressed form allows highly efficient memory operations using `memcpy`, but also using strided copies in OpenCL, HIP, and CUDA. Additionally, the memory requirement of storing halo information is drastically reduced, particularly in cases of structured data. In the example given in fig. 2.2 the compressed storage requires $^2/_5$ of the memory required for a full set of halo indices, and the effect increases with larger halo regions. Yet, in the case of unstructured halo regions, the memory requirement of using the compressed storage might increase if the region does not easily decompose into into rectangular subregions.

The rectangular subregions found by Tausch do not necessarily correspond to rectangular regions in the mesh. Instead, in a slightly more abstract sense, they correspond to rectangular regions in the memory — e.g., the example shown in fig. 2.3 illustrates how a 10×2 rectangular region in the mesh is detected as 20×1 rectangular region in the memory.

The same concept extends to three dimensions, where a three dimensional volume is interpreted by Tausch as a two or possibly one dimensional memory region. It is also possible to directly pass the halo region information in compressed form to Tausch instead of vectors of indices. In the case of the example shown in fig. 2.3 both representations (10×2 and 20×1) are valid.

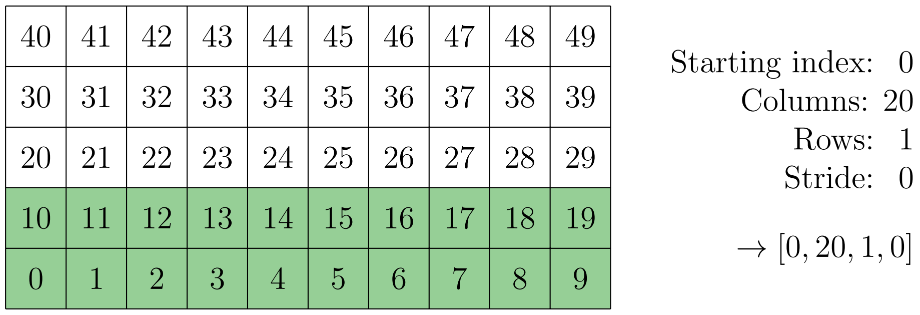


Figure 2.3: Example of rectangular region not corresponding directly to mesh region, halo region highlighted in blue.

In general, a one dimensional compression is preferential to a two dimensional one, as it allows the use of fewer `memcpy` operations and thus offers better performance. Since a halo inherently corresponds to the surface of the domain (i.e., at most two dimensional), a three dimensional compression of the halo brings little to no benefit while increasing the overhead of the actual compression step.

In addition to the compressed storage of halo indices, it supports various communication strategies to boost performance if allowed by the underlying hardware and software.

- Derived datatypes: Instead of copying the data to be sent into dedicated send buffers, it can be sent off directly from the originating buffer by using derived datatypes in MPI. This allows skipping the intermediate step of copying data into/out of a dedicated send/receive buffer. However, it also means that the locations of the data-to-be-sent cannot be altered until the send operation has completed.

- Persistent communication channels: Instead of creating new communication channels each time they are required, it is possible to establish persistent communication channels. In particular for iterative algorithms, this allows reusing the existing channels every iteration reducing the potential overhead.

- Single-copy and multi-copy: Data between the CPU and GPU can be copied with a single memory copy (beneficial for data that is split across multiple regions) or with multiple copies, one for each individual region (beneficial for data that mostly belongs to the same region).

- CUDA-aware MPI: If the underlying architecture supports CUDA-aware MPI, use thereof has the potential to avoid going through the CPU completely when moving data between different GPUs.

- Remote memory access (RMA): Instead of two-sided communication (sending and receiving data explicitly) an MPI window can be defined that makes certain memory on a remote rank locally available. In order to exchange data remote data can be read from and written to by the local rank without the remote rank being involved. This can improve the performance by reducing the total number and overhead introduced by explicit calls to `send` and `recv` for each halo region.

- Neighborhood collectives: By default the various halo regions need to be sent of and received individually. As an alternative it is possible to make use of neighborhood collectives (`MPI_Ineighbor_alltoallv`) to exchange all the data at the same time. Depending on the implementation and hardware in use this can result in further optimized data movements.

The choice of the optimal optimization strategies is highly dependent on use case and available hardware. Tausch provides the method `testForBestCommunication` to test all possible strategies and their combinations to find the best choice. The result is returned as a vector with two elements, the first one containing the value of the `enum` specifying the sending communication strategy, and the second entry the `enum` for the receiving communication strategy. It finds the best choice by performing halo exchanges for small and large problem sizes and recording which ones perform better than the others. It is possible to specify the range of problem sizes to test over in order to tune it for more specific applications.

A sample run of this test on a local machine with a single MPI rank returns the output as shown in listing 2.1. As can be seen, and as one would intuitively expect, when on the same

```
** Testing communication strategies

  > Testing Default (send) / Default (recv) ... 18.3825 ms
  > Testing Default (send) / DerivedMpiDatatype (recv) ... 18.1975 ms
  > Testing Default (send) / MPIPersistent (recv) ... 18.4347 ms
  > Testing TryDirectCopy (send) / TryDirectCopy (recv) ... 0.235729 ms
  > Testing DerivedMpiDatatype (send) / Default (recv) ... 16.4172 ms
  > Testing DerivedMpiDatatype (send) / DerivedMpiDatatype (recv) ... 15.6828 ms
  > Testing DerivedMpiDatatype (send) / MPIPersistent (recv) ... 16.3059 ms
  > Testing MPIPersistent (send) / Default (recv) ... 18.3637 ms
  > Testing MPIPersistent (send) / DerivedMpiDatatype (recv) ... 17.9299 ms
  > Testing MPIPersistent (send) / MPIPersistent (recv) ... 18.3672 ms
  > Testing Collectives (send) / Collectives (recv) ... 20.1225 ms

** Best strategy combo:

              Sending: TryDirectCopy
            Receiving: TryDirectCopy
Average time required: 0.235729 ms

```

Listing 2.1: Testing communication strategies on single MPI rank.

MPI rank the best choice for communication is to not go through MPI at all but simply copy the data directly as both the sender and receiver share the same memory space. Re-running this test on the same local machine but with multiple MPI ranks yields the slightly different result shown in listing 2.2. In this case, running with 12 MPI ranks, making use of RMA

```
 ** Testing communication strategies

  > Testing Default (send) / Default (recv) ... 59.752 ms
  > Testing Default (send) / DerivedMpiDatatype (recv) ... 56.5195 ms
  > Testing Default (send) / MPIPersistent (recv) ... 52.6566 ms
  > Testing DerivedMpiDatatype (send) / Default (recv) ... 51.6321 ms
  > Testing DerivedMpiDatatype (send) / DerivedMpiDatatype (recv) ... 53.9124 ms
  > Testing DerivedMpiDatatype (send) / MPIPersistent (recv) ... 50.9173 ms
  > Testing MPIPersistent (send) / Default (recv) ... 52.6983 ms
  > Testing MPIPersistent (send) / DerivedMpiDatatype (recv) ... 55.0589 ms
  > Testing MPIPersistent (send) / MPIPersistent (recv) ... 50.4296 ms
  > Testing RMA (send) / RMA (recv) ... 26.0214 ms
  > Testing Collectives (send) / Collectives (recv) ... 55.0887 ms

** Best strategy combo:

              Sending: RMA
            Receiving: RMA
Average time required: 26.0214 ms

```

Listing 2.2: Testing communication strategies on 12 MPI ranks.

for one-sided communication yields the fastest communication time.

This method by default prints its results to the screen, however, besides an MPI communicator as first argument the methods takes as second argument an optional boolean. If that boolean is set to false then no output will be generated and the method simply returns its result. This allows for calling this strategy finder during the setup stage of some code without polluting the output log.

2.5 PERFORMANCE MODEL

In order to evaluate the performance of our new implementation we employ the max-rate performance model [17],

$$T = t_c + r\alpha + \frac{kn}{\min(R_N, kR_C)} \tag{2.1}$$

where t_c is the time for copying the data into/out of dedicated send/receive buffers, r is the number of messages a rank is sending, α is the latency introduced by MPI per message, k is the number of processes, n is the number of bytes sent per process, R_N is the injection bandwidth, and R_C is the rate that can be achieved by each process in sending or receiving a message. The values for α, R_N and R_C can be found in table 2.1, with the values obtained through experiment. Note that R_N only impacts the rendezvous protocol.

protocol	α [s]	R_N [B/s]	R_C [B/s]
short	1.38×10^{-6}	—	3.81×10^{9}
eager	2.26×10^{-6}	—	2.36×10^{9}
rendezvous	1.14×10^{-5}	2.28×10^{9}	1.77×10^{10}

Table 2.1: Max-rate model parameters for Lassen, obtained through experiment.

In order to compare the performance of Tausch to the performance model we consider a test that performs a three-dimensional halo exchange. Figure 2.4 shows a visualization of this halo exchange test in three dimensions, and algorithm 2.1 describes the actual algorithm that is being used. The test code implementing this example is run on the Lassen supercomputer.

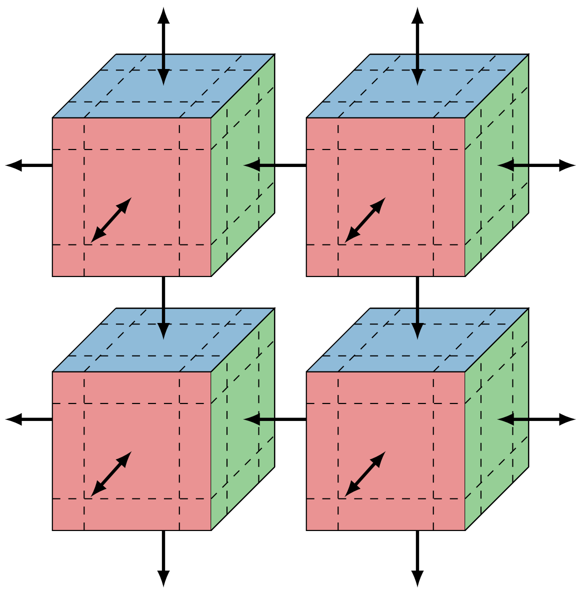


Figure 2.4: Visualization of three-dimensional halo exchange used as test case

We present a performance evaluation for test runs on both the CPU and the GPU. Note that the only difference between those two is in the copying the halo data into their dedicated send buffers, for the GPU test runs this involves calls to `cudaMemcpy` or `hipMemcpy`.

fig. 2.5 shows the comparison of the performance model and the test code. The colored regions show the range of values (min/max) across all ranks, the lines show the average timings. In order to get a handle on the average expected performance, the parameters for the performance model are for inter-node communication (i.e., using the network) and for copying only consecutive chunks of memory (without stride). Thus, the model will be

Algorithm 2.1: Algorithm of test code

```
1  Create data buffers
2  Compose halo information
3  n_test ← number of tests
4  n_timing ← number of timings per operation
5  for test←1, n_test do
6      MPI_Barrier
7      Start pack timer
8      for t←1, n_timing do
9          Pack halo data to be sent off
10         into dedicated send buffer
11     end
12     Stop pack timer
13     Start communication timer
14     for t←1, n_timing do
15         Send data off to neighbors
16         using MPI_Isend
17         Receive data from neighbors
18         using MPI_Irecv + MPI_Wait
19     end
20     Stop communication timer
21     Start unpack timer
22     for t←1, n_timing do
23         Unpack received halo data
24         out of dedicated receive buffer
25     end
26     Stop unpack timer
27 end
```

slightly too optimistic for memory copies, especially for the larger problem sizes. For the smaller problem sizes data located at some stride still falls within one or just a few cache lines resulting in a performance that is near ideal. On the other hand, the communication prediction of the model is slightly pessimistic, ranks that lie on the same node and/or socket will result in faster communication performance than the predicted performance. Overall, the prediction by the performance model will be an average of the best and worst performance between any two ranks.

The minimum values for the test runs are the fastest time for doing a halo exchange between any two ranks, and likely stem from two ranks living on the same socket. Conversely, the maximum values likely stem from two ranks living on different nodes that are far apart. From the results we see that the modeled and actual performance closely align.

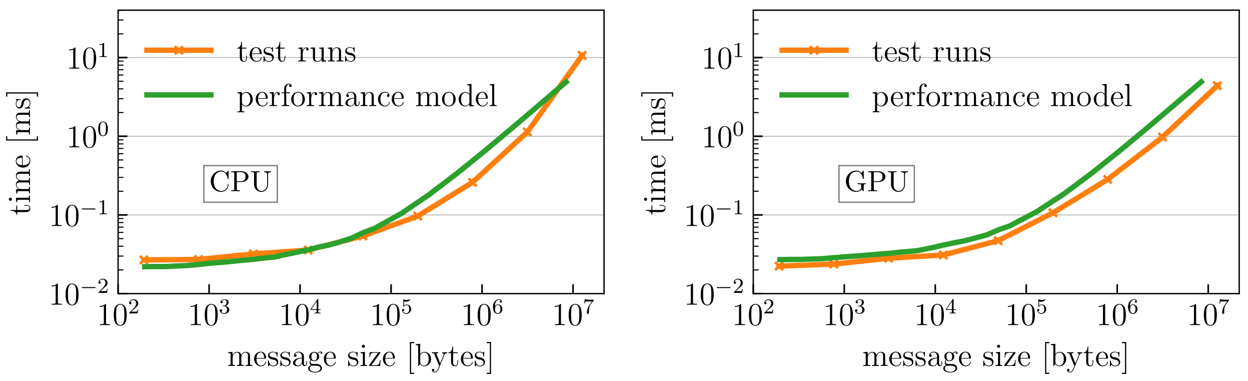
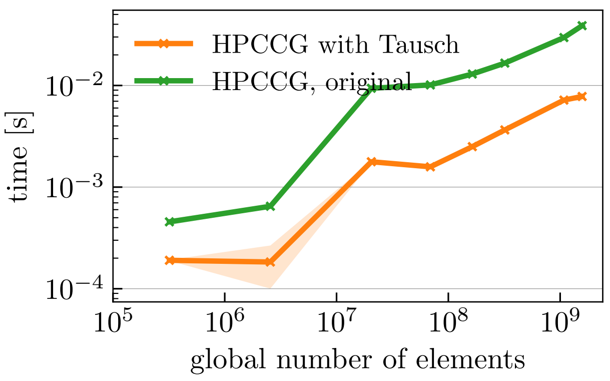


Figure 2.5: Performance Model on CPU using 320 ranks across 8 nodes (left) and GPU using 32 ranks across 8 nodes (right) running on Lassen.

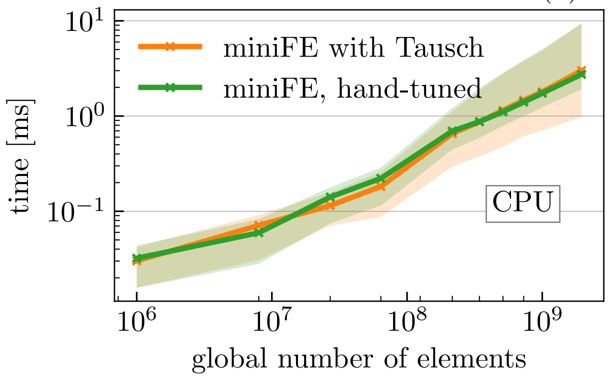
2.6 RESULTS

We used our library in different applications exercising both the CPU and GPU aspects of halo exchanges and compared its performance to the original code. The first two applications come from the Mantevo mini application suite, HPCCG [18] and miniFE [19]. The CPU-only application HPCCG is a simple conjugate gradient code that generates a 27-point finite difference stencil for a 3D chimney domain on an arbitrary number of processors. The slightly more recent mini application miniFE provides implementations of an unstructured finite elements code for CPUs and GPUs on various architectures. Both mini applications provide their own halo exchange implementations, both optimized to varying degrees. fig. 2.6 presents a comparison of that original implementation to a modified version that uses Tausch to facilitate all required exchanges. As can be seen from fig. 2.6, Tausch on the CPU at least matches if not outperforms the existing solution. This can be achieved without much effort as Tausch requires typically no more than 4-5 lines (seldom more) to be used, whereas a highly tuned existing halo exchange code requires many tens of lines of code and careful design to not only actually achieve the desired performance but also minimize the risk of accidental bugs introduced. On the GPU, Tausch is very close to the existing solution, within much less than a factor of 2, up to matching the existing performance for the larger problem sizes. The ease of use of Tausch and near-matching performance is a clear argument in favor of using Tausch. It is worth pointing out that the use of Tausch in the CPU and GPU version is nearly identical, with the only difference being the buffer pointers and the enabling of CUDA-aware MPI (a single line of code).

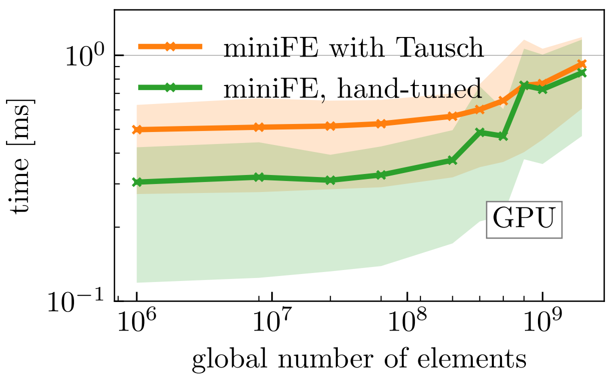
In addition to the two mini application presented above, Tausch is also used within Cedar [20], a robust, variational multigrid library implementing BoxMG on large scale parallel systems. There, it replaces a legacy halo exchange library (MSG [21]), for both the



(a) HPCCG



(b) miniFE (CPU)



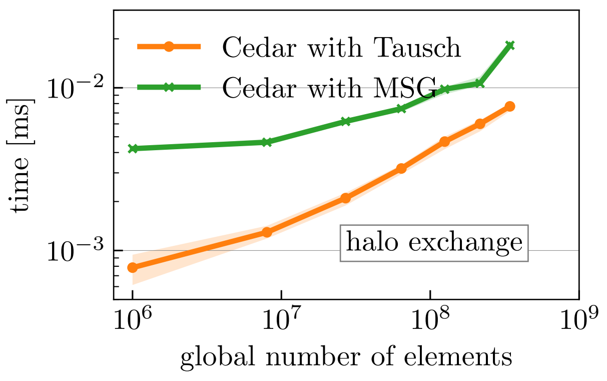
(c) miniFE (GPU)

Figure 2.6: Tausch in HPCCG (a) and miniFE (b and c)

halo data itself and the associated stencil. fig. 2.7 shows the halo/stencil exchange operations inside Cedar for both the old implementation and our new library. Within Cedar, Tausch provides structured communication for the solver in two and three dimensions. In addition to providing performant halo communication with predictable performance, Tausch enables parallel plane relaxation with coarse-grid problems redistributed on subcommunicators. Prohibited in the past by the legacy communication library, Tausch supports many non-interfering instances. This is used to create thousands of instances of Tausch for large 3D solves with minimal overhead [22]. Since at least around 30% of the time in Cedar is spent in communication [23], this improvement leads to significant performance gains overall.

2.7 CONCLUSION

In this chapter we discussed techniques and novel ideas for halo exchanges targeting large heterogeneous machines. We presented a new halo exchange library called *Tausch*, and all the benefits it offers both in terms of performance but also ease of use. Measuring its performance against a three dimensional performance model showed that its performance lies



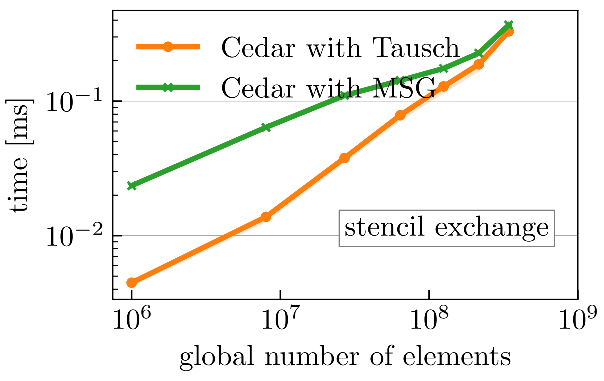


Figure 2.7: Tausch vs MSG in Cedar

within expectations. Using Tausch within the three sample applications HPCCG, miniFE, and Cedar confirmed that result by illustrating the competitive performance exhibited by Tausch by at least matching if not outperforming the previous performance of halo exchanges by up to an order of magnitude.

CHAPTER 3: SMOOTHERS FOR THE STOKES EQUATIONS

Finite-element discretizations are a popular choice for coupled systems such as magnetohydrodynamics (MHD), or the Stokes or Navier-Stokes equations. Even though solvers for finite-element discretizations of such saddle-point problems are well established, designing efficient and scalable solvers on emerging computing architectures for such systems remains an ongoing challenge [24–26].

In this chapter, we focus on preconditioned Krylov methods for the linear or linearized systems of equations that arise in solving such problems. There are two main classes of preconditioners for such systems: preconditioners based on block-factorization approaches [27–29] and those based on monolithic multigrid principles [30, 31]. Within each class there is considerable variability in their building blocks, such as the choice of relaxation scheme in monolithic multigrid, including Braess-Sarazin [32, 33], Vanka [34], and Schur-Uzawa [35] relaxation.

From existing studies [31, 36–38] some insight can be gained into which algorithms are preferable in serial (low-performance) computing settings, but there are relatively few fair comparisons of these approaches in literature [39–42], in particular for geometric multigrid on modern architectures, such as GPUs. This is particularly important given changes in prevailing HPC architectures in the past two decades. In [37, 43] it is observed that monolithic multigrid with Vanka relaxation leads to scalable performance mathematically and that additive variants of Vanka are well-suited for implementation on modern GPUs but, to our knowledge, no performance studies support this claim. As we discuss below, one main difficulty in getting good performance out of the Vanka algorithm is the high cost of memory movement for forming the various Vanka patches and updating the global solution, which requires a careful approach to achieve good performance. Similar issues have recently been considered using related additive Schwarz relaxation schemes within multigrid applied to the Poisson equation [44], where it was found that, with optimization of memory caching and reducing communication between patches, additive Schwarz methods built around cell-centric patches are capable of outperforming optimized point-Jacobi-based relaxation schemes.

More broadly, the parallel scalability of multigrid algorithms on modern architectures faces many challenges related to indirection and increased coarse-grid communication costs [45, 46]. This makes data locality and the cost of data movement a central issue, but one that can be solved by carefully exploiting structure in the problem as is done, e.g., in black box multigrid (BoxMG) algorithms [23, 47, 48]. In this work, we use highly structured meshes that allow us to encode various information about the data and how it is accessed in the structure itself, similarly to the BoxMG paradigm. Notably, we work with a structured matrix representation and implement algorithms that take full advantage of this, in order to minimize memory accesses and maximize floating point operations (arithmetic intensity).

In this chapter, we first introduce the Stokes equations as our model problem and provide an overview of their structure and resulting discretization. We then introduce two different preconditioners for the FGMRES algorithm used to solve such equations, monolithic multigrid and the upper Block-Triangular preconditioner. For monolithic multigrid, we introduce three different relaxation schemes, Braess-Sarazin, Vanka, and Schur-Uzawa. As Braess-Sarazin and Vanka are two common choices for relaxation schemes, we focus our performance analysis on these, noting that Schur-Uzawa can be implemented with the same kernels as Braess-Sarazin. After illustrating which kernels are the biggest contributors to each algorithm, we then break the performance analysis into two parts: common kernels (matrix-vector and array operations) and Vanka-specific kernels (forming patches, updating the global solution, solving patch systems). For each part, the arithmetic intensity, performance, and runtime are analyzed in order to gain a full understanding of the algorithms and how they compare. This leads to a roofline model to investigate how much better the kernels could be doing, if at all. To finish our performance analysis, we compare a full solve of the Stokes equations using FGMRES preconditioned with both a Block-Triangular preconditioner and a multigrid V-cycle preconditioner with Braess-Sarazin, Vanka, and Schur-Uzawa as relaxation schemes. We show that Vanka is, indeed, a competitive algorithm on modern architectures with careful design. We also show that simply porting a performant CPU implementation of Vanka directly to the GPU is not sufficient for achieving competitive performance.

3.1 THE STOKES EQUATIONS AND THEIR DISCRETIZATION

3.1.1 Problem Setup

Fluid flow where viscous forces are much greater than advective inertia is called Stokes flow. In nature, flow with such properties occurs in many places, e.g., in geodynamics or in the swimming movement of microorganisms. The equations of motions arising from this flow are called the Stokes equations and can be viewed as a simplification of the steady-state Navier-Stokes equations in the limit of small Reynolds number, $Re \ll 1$. They are not only well-suited to be solved by iterative solvers [49], but they also serve as a suitable prototype for a wide range of models that lead to saddle-point structure.

Specifically, we consider the incompressible Stokes equation with constant viscosity ν in the unit-square domain $\Omega = [0,1]^2 \in \mathbb{R}^2$. The equations are given by

$$-\nu \nabla^2 \mathbf{u} + \nabla p = \mathbf{f}, \tag{3.1}$$

$$\nabla \cdot \mathbf{u} = 0. \tag{3.2}$$

Dirichlet boundary conditions on velocity are enforced on all edges of the domain, but no boundary conditions are imposed on pressure. Here, we consider no-flux boundary conditions,

$$\mathbf{u} \cdot \mathbf{n} = 0 \text{ on } \partial\Omega, \tag{3.3}$$

where $\mathbf{n}$ is the outward pointing normal vector. Thus, we define the Hilbert space $\mathbf{H}_0^1(\Omega)$ as

$$\mathbf{H}_0^1(\Omega) = \{\mathbf{v} \in \mathbf{H}^1(\Omega) : \mathbf{v} \cdot \mathbf{n} = 0 \text{ on } \partial\Omega\} \tag{3.4}$$

and $L^2(\Omega)/\mathbb{R}$ as the quotient space of equivalence classes of functions in $L^2(\Omega)$ differing by a constant. The weak form is then defined as: Find $(\mathbf{u}, p) \in \mathbf{H}_0^1(\Omega) \times L^2(\Omega)/\mathbb{R}$ such that

$$a(\mathbf{u}, \mathbf{v}) + b(\mathbf{v}, p) = (\mathbf{f}, \mathbf{v}) \quad \forall \mathbf{v} \in \mathbf{H}_0^1(\Omega) \tag{3.5}$$

$$b(\mathbf{u}, q) = 0 \qquad \forall q \in L^2(\Omega)/\mathbb{R} \tag{3.6}$$

where

$$a(\mathbf{u}, \mathbf{v}) = \nu \int_\Omega \nabla \mathbf{u} : \nabla \mathbf{v}, \tag{3.7}$$

$$b(\mathbf{v}, p) = \int_\Omega p \nabla \cdot \mathbf{v}. \tag{3.8}$$

We work with a manufactured solution [42, 50] that satisfies the properties above, given by

$$\mathbf{u}(x,y) = \begin{cases} x(1-x)(2x-1)(6y^2-6y+1) \\ y(y-1)(2y-1)(6x^2-6x+1) \end{cases} \tag{3.9}$$

$$p(x,y) = x^2 - 3y^2 + \frac{8}{3}xy, \tag{3.10}$$

with $\mathbf{f}$ computed to satisfy (3.1). A visualization of this solution is found in fig. 4.2.

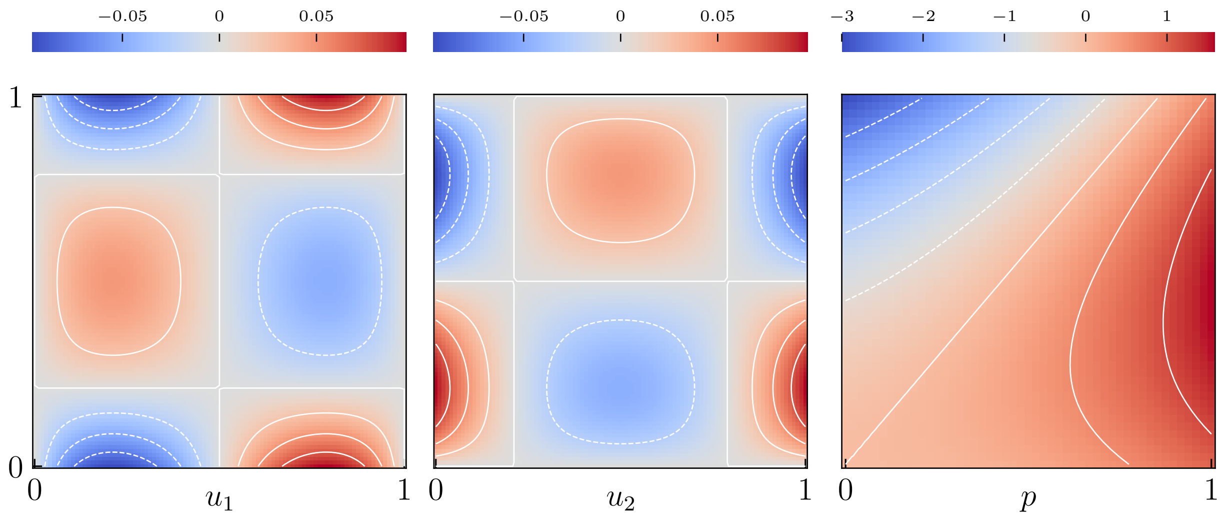


Figure 3.1: Visualization of three components of the manufactured solution.

3.1.2 Discretization

We consider the standard Q2–Q1 Taylor-Hood mixed finite-element discretization on a uniform grid for discretizing the system in eqs. (3.1) and (3.2). For the velocity, this uses Q2 elements, with biquadratic polynomials for each component on each element as a basis. For the pressure, this uses Q1 elements, with bilinear polynomials on each element as a basis. Both velocities and pressures are required to be continuous across element boundaries. An illustration of the degrees of freedom in these elements is shown in fig. 3.2. The resulting Q2

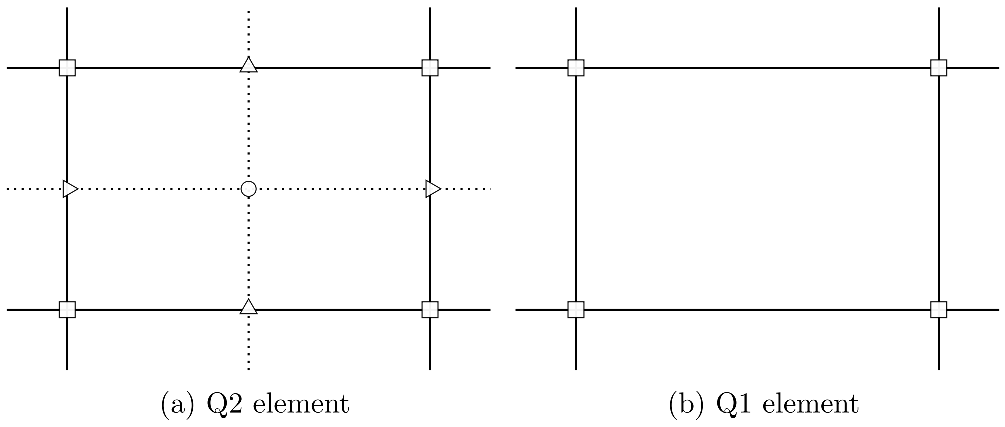


Figure 3.2: Illustration of the degrees of freedom for a Q2 and Q1 element, with different types of degrees of freedom identified by different shapes.

and Q1 elements have a total of nine and four degrees of freedom per element, respectively. Such a discretization of the Stokes equations directly relates back to the weak form as shown in eqs. (3.5) and (3.6), defining the matrices L and B by

$$L_{i,j} = a(\boldsymbol{\psi}_j, \boldsymbol{\psi}_i) \tag{3.11}$$

$$B_{k,j} = b(\boldsymbol{\psi}_j, \phi_k). \tag{3.12}$$

It is important to note that the introduction of a basis gives us two "views" on the finite-element approximations, writing $\mathbf{u} = \sum_i u_i \boldsymbol{\psi}_i$ and $p = \sum_k p_k \phi_k$, so we can consider the functions $\mathbf{u}$ and p directly, or think about their coefficients in the basis expansion, $\{u_i\}$ and $\{p_k\}$. In what follows, we follow the usual convention of overloading the notation $\mathbf{u}$ and p to denote both the functions themselves and the vectors of basis coefficients, with the distinction typically clear from context, in that $L\mathbf{u}$ is the matrix acting on the basis coefficients, while $a(\mathbf{u}, \mathbf{v})$ is the bilinear form evaluated on the function. With this matrix representation, the solution of the weak form can be expressed as that of the linear system

$$\begin{bmatrix} L & B^T \\ B & 0 \end{bmatrix} \begin{bmatrix} \mathbf{u} \\ p \end{bmatrix} = \begin{bmatrix} \mathbf{f} \\ 0 \end{bmatrix} \tag{3.13}$$

where L is the discretized Laplacian, B and B^T are the discretized divergence of $\mathbf{u}$ and gradient of p, respectively, $\mathbf{u}$ and p are the discretized velocity and pressure components of the solution, and $\mathbf{f}$ is the velocity component of the right-hand side. In what follows, to save space, we will write the 2×2 block matrix in eq. (3.13) as A.

Such a system is challenging to solve, as it is symmetric but indefinite due to the zero block in the lower right-hand corner of the system matrix. This causes many common iterative methods (e.g., simple stationary methods like Jacobi and Gauss-Seidel) to not work as they typically involve inverting the diagonal of the system matrix. Study of numerical methods for solution of such saddle-point systems is a well-established discipline [28]. One possible solution to these challenges is the use of block preconditioners, based on the block LU factorization of the coupled system. This approach has been well-developed for discretizations of the Stokes equations [27, 28]. However, existing studies [42] suggest that monolithic preconditioners can be more efficient. Thus, we also consider monolithic multigrid as preconditioner for FGMRES. We note that monolithic multigrid preconditioners are generally not symmetric and positive definite and, so, they cannot be used directly as preconditioners for MINRES; however, the added computational work for orthogonalization in FGMRES is more than made up for by the quick convergence of the monolithic multigrid approach.

3.1.3 Structured Matrix Representation

Any iterative solver naturally depends on calculations of matrix-vector products for the block-structured matrix in eq. (3.13). In general, such calculations require indirect addressing, when arbitrary numbers of elements can be adjacent to each node of the mesh, leading to irregular communication patterns. However, when we restrict the mesh to have logically rectangular structure (meaning that each node is at the intersection of four edges, and is adjacent to four elements), then applying the discretization matrix can be done in a stencil-wise fashion, where each degree of freedom requires information from at most a 2×2 element patch. This allows storing the system matrices in a highly efficient data structure by numbering the degrees of freedom in lexicographic order. For the Q1 discretization, this ordering is natural, since the only degrees of freedom occur at the nodes in the mesh, that can be labeled lexicographically by (x, y)-indices. For the Q2 discretization, we separate the degrees of freedom into four sets, given by those associated with the nodes of the mesh, then those at midpoints of the x and y edges, and, finally, those associated with the cell centers. Figure 3.3 shows a local numbering of those degrees of freedom on the 2×2 patch around a node.

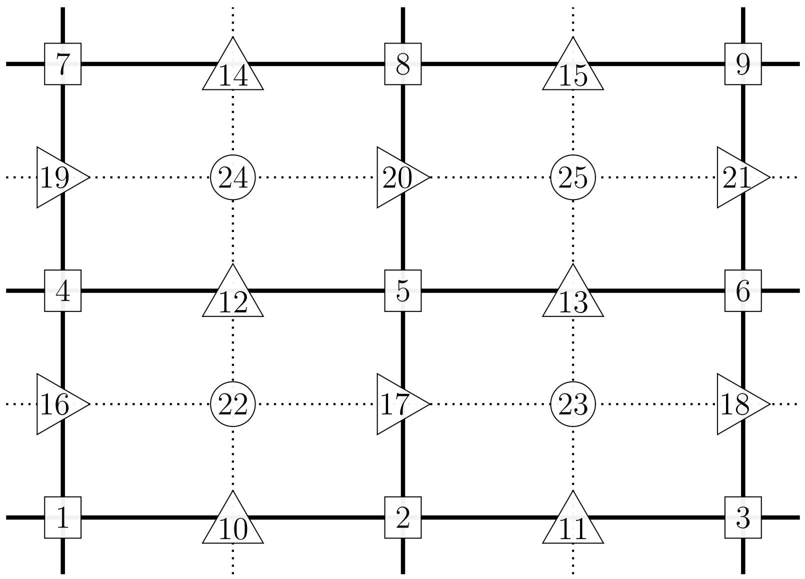


Figure 3.3: Local numbering of degrees of freedom around nodal degree of freedom 5.

With this ordering, the system matrix is stored as an array of arrays, where the first (outer) array index corresponds to the row in the matrix associated with that degree of freedom, and the inner arrays stored at each outer index contain a fixed number of entries corresponding to the number of degrees of freedom surrounding the respective degree of freedom, with the respective columns implicitly stored by the positions in the inner array.

Thus, the inner arrays have a fixed length of 9 for Q1 elements and a fixed length of 25 for the nodal degree of freedoms for Q2 elements (with the ordering given in fig. 3.3). Such a matrix format corresponds to a hybrid of the ELLPACK and the SELL-n matrix formats [51, 52], as we use padding to always store the full stencil and, thus, have a consistent row length, but we also use a (variable) block size n with each type of degree of freedom forming its own block. Exploiting the structure of our setup enables us to avoid having to store any indices or pointers, as this information is encoded in the structure itself. Thus we are able to minimize the memory storage needed and maximize the performance, as only a small amount of memory needs to be loaded and read for any operation. fig. 3.4 illustrates how the matrix entries for the different types of degrees of freedom are stored in memory, with each box corresponding to one inner array.

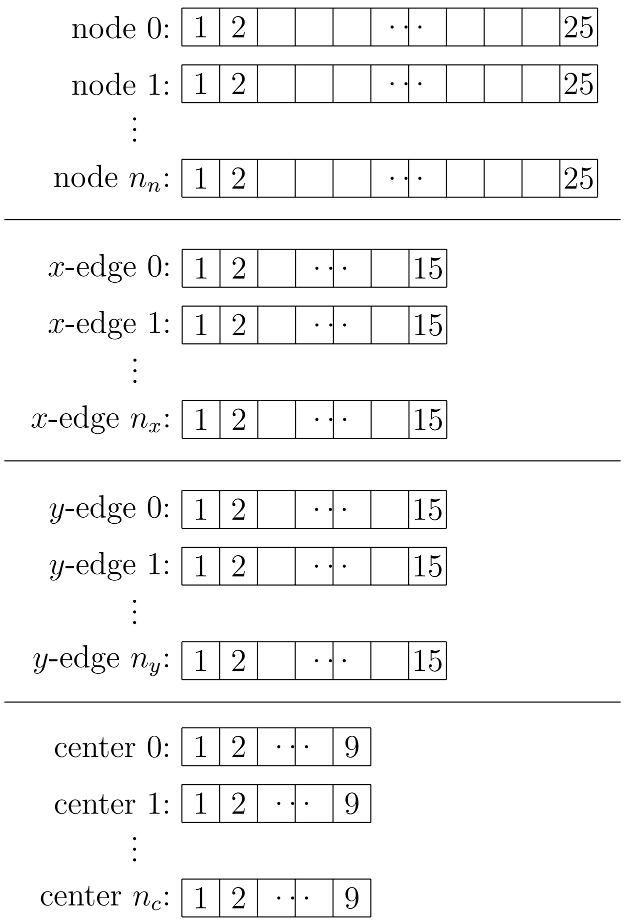


Figure 3.4: Illustration of how matrix entries are stored in memory for the four different groups of degrees of freedom.

This approach translates directly to higher dimensions. For a 3D discretization of this type on "brick" elements, we would extend the above to $2 \times 2 \times 2$ element patches with

$5 \times 5 \times 5 = 125$ nodal degrees of freedom (5 in each dimension). These can be labeled in an analogous way and, consequently, stored similarly in a single array of arrays, with outer index corresponding to the rows in the system matrix, and inner index corresponding to the local numbering around each degree of freedom.

3.2 MULTIGRID

Multigrid methods are based on the notion that standard (but slow-to-converge) iterative methods are generally effective at reducing oscillatory errors in a discrete approximation, but that the subspace of smooth and slow-to-converge errors of such an iteration is better treated by a complementary process [53, 54]. A natural approach to reduce the smooth errors is with correction from a coarser-grid realization of the same discretized problem, where those smooth modes can be accurately resolved by recursively applying the same iterations to problems with fewer degrees of freedom, until some suitably coarse version of the problem is found where a sparse direct solver can be effectively applied. For higher-order discretizations and systems of PDEs, such an error classification breaks down [55], but the multigrid principle remains effective, in that we can define relaxation schemes that effectively damp a large portion of the error in a given approximation, and the remaining error can be effectively corrected from a coarse grid.

The standard multigrid solution algorithm is known as the V-cycle, since it traverses a given hierarchy of meshes from the given finest grid to the coarsest, then back to the finest. In the "downward" sweep of the traversal (from fine-to-coarse), on each level, an initial approximation (generally a zero vector) is improved by a specified relaxation scheme. Then, the residual associated with that improved approximation is calculated and restricted to the next coarsest grid, where the scheme recurses. On the "upward" sweep, the current approximation is improved by interpolating a correction back from the next coarser grid, then running relaxation again, before proceeding to the next finer grid. For transferring residuals and corrections between grids, we define a single interpolation operator that maps from a coarse grid to the next finer grid, and use its transpose as a restriction operator. In this work, we follow the standard geometric multigrid approach of using the finite-element interpolation operators, that naturally map from coarse-grid versions of the Q2 and Q1 spaces to their fine-grid analogues. Algorithm 3.1 presents an algorithmic overview of the multigrid V-cycle for the Stokes equations, following the convention that level 0 is the coarsest grid in the hierarchy, and we are interested in the solution on some given fine grid, for fixed $l > 0$.

While traditional relaxation schemes, such as (weighted) Jacobi or Gauss-Seidel are effective for elliptic problems, they generally cannot be applied directly to saddle-point systems,

Algorithm 3.1: Multigrid V-cycle for Stokes equations

```
1  function MG($A_l$, $\mathbf{u}_l$, $p_l$, $\mathbf{f}_l$, $g_l$, $l$):
2      Relax on $\mathbf{u}_l$ and $p_l$
3      Compute residual: $\begin{bmatrix} \mathbf{r}_{\mathbf{u},l} \\ r_{p,l} \end{bmatrix} = \begin{bmatrix} \mathbf{f}_l \\ g_l \end{bmatrix} - A_l \begin{bmatrix} \mathbf{u}_l \\ p_l \end{bmatrix}$
4      Restriction: $\begin{bmatrix} \mathbf{r}_{\mathbf{u},l-1} \\ r_{p,l-1} \end{bmatrix} = P_{l-1}^T \begin{bmatrix} \mathbf{r}_{\mathbf{u},l} \\ r_{p,l} \end{bmatrix}$
5      if $l$ is 1 then
6          $\begin{bmatrix} \mathbf{e}_{\mathbf{u},0} \\ e_{p,0} \end{bmatrix} = A_0^{-1} \begin{bmatrix} \mathbf{r}_{\mathbf{u},0} \\ r_{p,0} \end{bmatrix}$
7      else
8          $\begin{bmatrix} \mathbf{e}_{\mathbf{u},l-1} \\ e_{p,l-1} \end{bmatrix} = \mathrm{MG}(A_{l-1}, \mathbf{0}, 0, \mathbf{r}_{\mathbf{u},l-1}, r_{p,l-1}, l-1)$
9      end
10     Correction: $\begin{bmatrix} \mathbf{u}_l \\ p_l \end{bmatrix} = \begin{bmatrix} \mathbf{u}_l \\ p_l \end{bmatrix} + P_{l-1} \begin{bmatrix} \mathbf{e}_{\mathbf{u},l-1} \\ e_{p,l-1} \end{bmatrix}$
11     Relax on $\mathbf{u}_l$ and $p_l$
```

due to the zero block in the matrix. Thus, specialized relaxation schemes are commonly developed and analyzed for the Stokes equations. In this work, we consider four different preconditioning approaches, comparing monolithic multigrid with Braess-Sarazin [32, 33], Vanka [34], and Schur-Uzawa [35] relaxation with an upper Block-Triangular preconditioner. We focus in particular on the former two, Braess-Sarazin and Vanka, as these are known to lead to effective monolithic multigrid methods, but also expose key kernels that are reused in the implementation of the latter two. We next provide an overview of all four algorithms, before focusing on aspects of implementation and performance when implementing these approaches on the GPU.

3.2.1 Braess-Sarazin Relaxation Scheme

The Braess-Sarazin iteration is based on an approximation of the block factorization of the system matrix in (3.13),

$$\begin{bmatrix} L & B^T \\ B & 0 \end{bmatrix} = \begin{bmatrix} L & 0 \\ B & \hat{S} \end{bmatrix} \begin{bmatrix} I & L^{-1}B^T \\ 0 & I \end{bmatrix}, \tag{3.14}$$

for $\hat{S} = -BL^{-1}B^T$. The original algorithm [32] proposed replacing the matrix, L, in the above by a scaled version of its diagonal, tD, for scalar t, and updating the current approximation by an under-relaxed solve of the saddle-point system with this replacement,

$$\begin{bmatrix}\mathbf{u}\\ p\end{bmatrix}^{new} = \begin{bmatrix}\mathbf{u}\\ p\end{bmatrix}^{old} + \omega_{BS}\begin{bmatrix}tD & B^T\\ B & 0\end{bmatrix}^{-1}\begin{bmatrix}\mathbf{r_u}\\ r_p\end{bmatrix}^{old} \tag{3.15}$$

where $\mathbf{r_u}$ and r_p are the respective residuals. For simplicity, we fix $\omega_{BS} = 1$ in what follows. This is equivalent to computing the (unweighted) updates, $\delta\mathbf{u}$ and δp, as approximate solutions of the block-factorized approximation to the system matrix,

$$\begin{bmatrix}tD & 0\\ B & S\end{bmatrix}\begin{bmatrix}I & \frac{1}{t}D^{-1}B^T\\ 0 & I\end{bmatrix}\begin{bmatrix}\delta\mathbf{u}\\ \delta p\end{bmatrix} = \begin{bmatrix}\mathbf{r_u}\\ r_p\end{bmatrix} \tag{3.16}$$

where $S = -\frac{1}{t}BD^{-1}B^T$ is the Schur complement of the approximated system. Equation (3.16) can be rewritten as two equations

$$S\delta p = r_p - \frac{1}{t}BD^{-1}\mathbf{r_u}, \tag{3.17}$$

$$\delta\mathbf{u} = \frac{1}{t}D^{-1}(\mathbf{r_u} - B^T\delta p). \tag{3.18}$$

The *inexact* variant of Braess-Sarazin [33] only approximately computes the solution to eq. (3.17), using standard weighted Jacobi (or other algorithms) to approximate the inverse of S (see also [56]. The full algorithm is given in algorithm 3.2.

Algorithm 3.2: Braess-Sarazin relaxation

```
1 Approximately solve $S\delta p = r_p - \frac{1}{t}BD^{-1}\mathbf{r_u}$ for $\delta p$ by relaxation.
2 Compute $\delta\mathbf{u} = \frac{1}{t}D^{-1}(\mathbf{r_u} - B^T\delta p)$.
3 Update $p^{new} = p^{old} + \omega_{BS}\delta p$.
4 Update $\mathbf{u}^{new} = \mathbf{u}^{old} + \omega_{BS}\delta\mathbf{u}$.
```

3.2.2 Vanka Relaxation Scheme

Vanka relaxation, in contrast, applies a block overlapping Schwarz iteration to the global saddle-point system. In this approach, we define sets of "patches" (or "subdomains" in the usual Schwarz notation) corresponding to 2×2 blocks of elements, where we take a single pressure degree of freedom at the central vertex and all velocity degrees of freedom on the associated (neighboring) elements, see fig. 3.5. For each patch, we define a restriction operator, V_i, that extracts degrees of freedom from the global matrix to those present on local patch i, and use this to restrict the system matrix to the ith patch, as

$$A_i = V_i A V_i^T \tag{3.19}$$

The Vanka algorithm is defined by looping over the patches and solving

$$A_i \begin{bmatrix} \delta\mathbf{u}_i \\ \delta p_i \end{bmatrix} = V_i \begin{bmatrix} \mathbf{r}_\mathbf{u} \\ r_p \end{bmatrix} \tag{3.20}$$

exactly for $\delta\mathbf{u}_i$ and δp_i, which are then used to update the global approximate solution in a weighted additive manner. The Vanka algorithm is given in algorithm 3.3. We note that

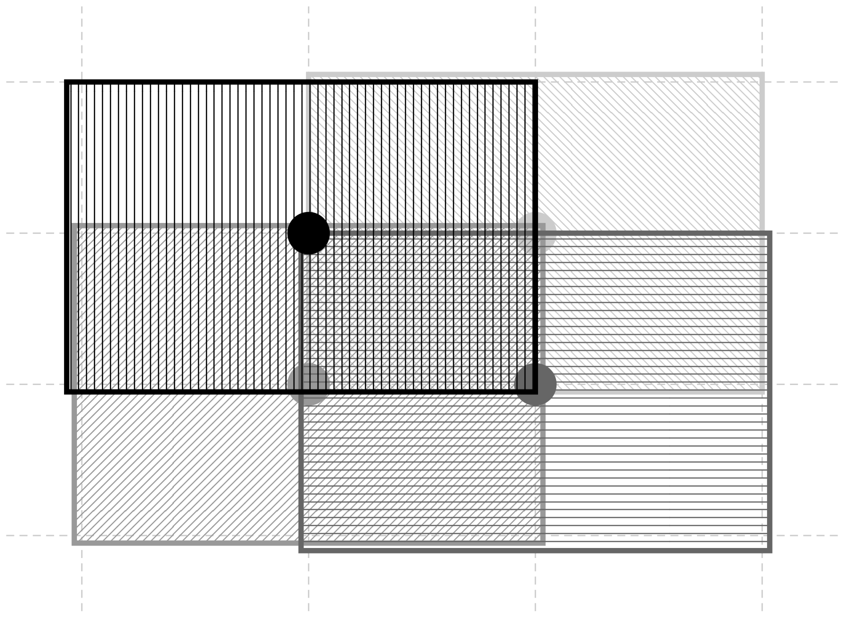


Figure 3.5: Illustration of overlapping 2×2 Vanka patches

we solve the patch systems exactly by inverting each patch matrix ahead of time, as they do not change between iterations.

Algorithm 3.3: Vanka relaxation (additive)

```
1 for $i \leftarrow 1$ to $N$ do
2     Solve $A_i \begin{bmatrix} \delta\mathbf{u}_i \\ \delta p_i \end{bmatrix} = V_i \begin{bmatrix} \mathbf{r}_\mathbf{u} \\ r_p \end{bmatrix}$ for $\begin{bmatrix} \delta\mathbf{u}_i \\ \delta p_i \end{bmatrix}$.
3 end
4 Update $\begin{bmatrix} \mathbf{u}^{new} \\ p^{new} \end{bmatrix} = \begin{bmatrix} \mathbf{u}^{old} \\ p^{old} \end{bmatrix} + \Sigma_{i=1}^{N} V_i^T W_i \begin{bmatrix} \delta\mathbf{u}_i \\ \delta p_i \end{bmatrix}$ where $W_i$ is the matrix with the weights.
```

3.2.3 Schur-Uzawa Relaxation Scheme

The Schur-Uzawa iteration is derived in a similar way to the Braess-Sarazin iteration. It is again based on an approximation of the factorization of the system matrix in (3.13),

$$\begin{bmatrix} L & B^T \\ B & 0 \end{bmatrix} = \begin{bmatrix} L & 0 \\ B & \hat{S} \end{bmatrix} \begin{bmatrix} I & L^{-1}B^T \\ 0 & I \end{bmatrix}, \tag{3.21}$$

for $\hat{S} = -BL^{-1}B^T$. As in Braess-Sarazin, we again replace the matrix, L, by some approximation that is easy to invert, again denoted tD, but also drop the upper-triangular term from this factorization, resulting in an inexact system computing updates, $\delta\mathbf{u}$ and δp, as approximate solutions of the block system,

$$\begin{bmatrix} tD & 0 \\ B & \hat{S} \end{bmatrix} \begin{bmatrix} \delta\mathbf{u} \\ \delta p \end{bmatrix} = \begin{bmatrix} \mathbf{r_u} \\ r_p \end{bmatrix}. \tag{3.22}$$

The system in (3.22) can be rewritten as two equations

$$tD\delta\mathbf{u} = \mathbf{r_u} \tag{3.23}$$

$$S\delta p = B\delta\mathbf{u} - r_p \tag{3.24}$$

that are solved for $\delta\mathbf{u}$ by directly inverting tD and for δp by standard weighted Jacobi to approximate the inverse of $S = -\frac{1}{t}BD^{-1}B^T$. The full algorithm is given in algorithm 3.4.

Algorithm 3.4: Schur-Uzawa relaxation

```
1 Compute $\delta\mathbf{u} = \frac{1}{t}D^{-1}\mathbf{r_u}$.
2 Approximately solve $S\delta p = B\delta\mathbf{u} - r_p$ for $\delta p$ by relaxation.
3 Update $p^{new} = p^{old} + \delta p$.
4 Update $\mathbf{u}^{new} = \mathbf{u}^{old} + \delta\mathbf{u}$.
```

3.2.4 Block-Triangular Preconditioner

The Block-Triangular preconditioner is also based on an approximation of the system matrix in (3.13), but we now consider an alternate form with unit block diagonal for the lower-triangular factor,

$$\begin{bmatrix} L & B^T \\ B & 0 \end{bmatrix} = \begin{bmatrix} I & 0 \\ BL^{-1} & I \end{bmatrix} \begin{bmatrix} L & B^T \\ 0 & \hat{S} \end{bmatrix}, \tag{3.25}$$

with $\hat{S} = -BL^{-1}B^T$. While Braess-Sarazin and Schur-Uzawa relaxation use simple approximations to L and $\hat{S}$ to approximate the inverse for relaxation within a multigrid cycle, it is more common to use multigrid on the blocks when using the block preconditioner directly. Here, as is common [27], we first approximate $\hat{S}$ by a mass matrix, $-M$, on the pressure space, then apply multigrid to this approximation. With this, we compute updates, $\delta\mathbf{u}$ and δp, as approximate solutions of the upper-triangular approximation to the system matrix,

$$\begin{bmatrix} L & B^T \\ 0 & -M \end{bmatrix} \begin{bmatrix} \delta\mathbf{u} \\ \delta p \end{bmatrix} = \begin{bmatrix} \mathbf{r_u} \\ r_p \end{bmatrix}, \tag{3.26}$$

using multigrid V-cycles to approximately invert L and M. The system in (3.26) can be rewritten as two equations

$$-M\delta p = r_p \tag{3.27}$$

$$L\delta\mathbf{u} = \mathbf{r_u} - B^T\delta p, \tag{3.28}$$

leading to the full algorithm given in algorithm 3.5.

Algorithm 3.5: Block-Triangular preconditioner

```
1 Approximately solve $M\delta p = -r_p$ for $\delta p$ using multigrid on $M$.
2 Approximately solve $L\delta\mathbf{u} = \mathbf{r_u} - B^T\delta p$ for $\delta\mathbf{u}$ using multigrid on $L$.
3 Update $p^{new} = p^{old} + \delta p$.
4 Update $\mathbf{u}^{new} = \mathbf{u}^{old} + \delta\mathbf{u}$.
```

3.3 OUR IMPLEMENTATION

We have implemented the outer FGMRES iteration and a multigrid V-cycle with the three relaxation schemes, Vanka, Schur-Uzawa, and Braess-Sarazin, as well as the Block-Triangular preconditioner, in C++, with custom data structures that provide structured matrix implementations of the required matrix and vector operations, as discussed above. This is achieved by using operator overloading to allow the optimization of the code for different architectures while preserving a clean implementation of the high-level algorithms. Underlying the custom data structures are standard STL vectors of `double` data type. Additionally, support for CUDA and OpenCL requires only a switch of the backend implementation while the high-level algorithm implementation remains largely untouched.

The resulting implementation yields an efficient solution algorithm for the incompressible

Stokes equations in two dimensions on both the CPU and the GPU. We limit our attention to optimizing implementations for a single CPU node or single GPU and focus on comparing performance using the different algorithms, taking advantage of the underlying structure. In principle, similar performance is expected for other discretizations of Stokes and other saddle-point problems, in two and three dimensions, but studying performance in these contexts is left for future work. We also do not consider extending this work to MPI-based parallelism or multi-GPU systems.

3.4 EXISTING WORK

John and Tobiska [57] investigate the performance of multigrid paired with Braess-Sarazin for solving the Stokes equations using P1-P0 finite elements. In the case of a W-cycle, the improvement in error reduction is approximately linear with the number of smoothing steps. For a V-cycle, convergence increases in general with increasing level of refinement. The work shows that multigrid paired with Braess-Sarazin is indeed a robust and reliable preconditioner.

Larin et al. [58] compares use of a coupled multigrid method with Vanka and Braess-Sarazin type relaxation schemes, along with preconditioned MINRES and an inexact Uzawa method. The focus is on solving the Stokes equations on the then-current hardware and architectures. The results show that all four methods are robust with respect to variations in parameters. The conclusion is that a multigrid W-cycle paired with diagonal Vanka results in the most efficient solver in terms of CPU time.

More recently, Adlet et al. [42] compare of a fully-coupled monolithic multigrid paired with Braess-Sarazin or Vanka as relaxation scheme, and a block-factorization preconditioner similar to the one we presented here. On CPU-only systems, multigrid paired with Vanka results in the best scaling and lowest iteration count. Yet, these solvers require significantly more work per iteration than the other preconditioners. As a result, multigrid paired with Braess-Sarazin yields the best time-to-solution on CPUs for the problems studied in that work.

3.5 PERFORMANCE ANALYSIS

By far the most costly component of the monolithic multigrid-preconditioned FGMRES solver is the relaxation scheme within the multigrid V-cycle. Thus, we focus our performance analysis on the implementations of two of the relaxation schemes, Vanka and Braess-Sarazin, alone, noting that multigrid with Schur-Uzawa relaxation reuses only components from that using Braess-Sarazin, while the Block-Triangular preconditioner also reuses primarily kernels from Braess-Sarazin relaxation as well.

Studying the performance of these methods requires careful analysis of memory movement and access. The standard metric for this is *arithmetic intensity*, which quantifies the relationship between floating-point operations and memory reads and writes. Another important quantity is the *FLOP rate*, which describes how many floating-point operations are performed in a given time frame. The runtime of the kernels (and how they relate to one-another) indicates the importance of each kernel when it comes to studying the performance. We first study and optimize the component kernels individually, before comparing performance of our four preconditioners for the FGMRES solver for the Stokes equations. For measuring the various metrics to evaluate and compare implementations on the GPU, we use NVIDIA's Nsight Compute[1] and Nsight Systems[2] software.

3.5.1 Test System

The system we use for each test is the Delta supercomputer[3] located at the National Center for Supercomputing Applications (NCSA). It is equipped with NVIDIA A100 GPUs that have a measured peak double-precision floating-point performance of 9472.34 GFLOP/s, 80GB on-chip memory, and a measured GPU memory bandwidth of 1264.42 GB/s, measured using the CS roofline toolkit[4]. For our final comparison of algorithms, we also run on the CPU nodes of the Delta supercomputer, that carry dual 64-core AMD 7763 processors with a base frequency of 2.45 GHz (max boost frequency of 3.5 GHz) and a per socket memory bandwidth of 204.8 GB/s, although we only consider serial runs on a single core here.

3.5.2 Kernels

To start the analysis of the different kernels, we first present an overview of each kernel and get a sense of how much they contribute to the overall runtime. Although the problem itself and the final parameter choices can have an important impact on the performance of a kernel, we compare kernels for a generic case here to provide a baseline.

Algorithm 3.6 shows the Braess-Sarazin algorithm in slightly different form than algorithm 3.2, to focus on the kernels involved for the various steps. These kernels are color-coded

[1]NVIDIA Nsight Compute: `https://developer.nvidia.com/nsight-compute`
[2]NVIDIA Nsight Systems: `https://developer.nvidia.com/nsight-systems`
[3]Delta supercomputer: `https://delta.ncsa.illinois.edu/`
[4]CS Roofline Toolkit: `https://bitbucket.org/berkeleylab/cs-roofline-toolkit`

for ease of comparing with the cost breakdown for a single iteration of Braess-Sarazin shown in fig. 3.6, indicating how much each kernel contributes to the overall runtime. (Noting that the percentages may not sum to 100%, due to rounding.) As is seen (and expected), most of the runtime is consumed by the matrix-vector operations. Many of these kernels are reused in the other solvers. We note in particular that the weighted Jacobi kernel for the pressure solve, used both for Braess-Sarazin and Schur-Uzawa relaxation (and in the Block-Triangular preconditioner) contributes very little to the overall runtime (less than 2%).

Algorithm 3.6: Braess-Sarazin with kernel breakdowns

```
1 Compute current residuals, $\mathbf{r_u}$ and $r_p$.
    Q2 matrix * Q2 vector
    Q2Q1 matrix * Q2 vector
    Q2Q1 matrix * Q1 vector
    array operations
2 Form right hand side of eq. (3.17).
    Q2 matrix * Q2 vector
    Q2Q1 matrix * Q2 vector
    array operations
3 Use Jacobi to compute approximation of $\delta p$ in eq. (3.17).
    weighted Jacobi
4 Use $\delta p$ to compute $\delta \mathbf{u}$ in eq. (3.18).
    Q2 matrix * Q2 vector
    Q2Q1 matrix * Q1 vector
    array operations
5 Update global solution with $\delta \mathbf{u}$ and $\delta p$.
    array operations
```

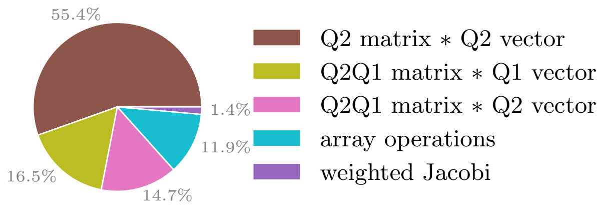
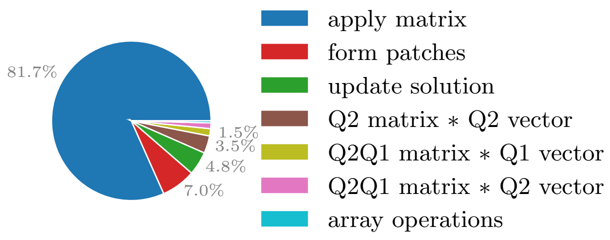


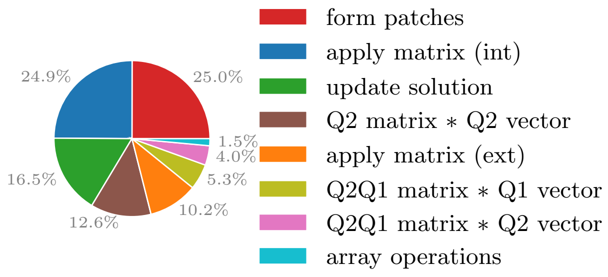
Figure 3.6: Braess-Sarazin: Kernels and their proportion of runtime

A similar kernel-focused restatement of algorithm 3.3 is given in algorithm 3.7, with the various kernels color-coded to correspond to timing breakdown for a single iteration shown in fig. 3.7. The left figure in fig. 3.7 shows the runtime for what we call "simple Vanka", where we do not take advantage of the fact that, for many settings, the Vanka submatrices are identical for most patches and can, thus, be stored once and used many times. This approach results in more than 75% of the runtime being spent applying the patch matrix inverses as each patch needs to load its own Vanka submatrix from global memory. The right figure in fig. 3.7 shows the results using a "tuned Vanka" implementation, where patches that have identical submatrices take advantage of fast shared memory to optimize memory

```
Algorithm 3.7: Vanka (tuned) with kernel breakdowns
1 Compute current residuals, $\mathbf{r_u}$ and $r_p$.
    Q2 matrix * Q2 vector
    Q2Q1 matrix * Q2 vector
    Q2Q1 matrix * Q1 vector
    array operations
2 Form patch right hand sides of eq. (3.20)
    Q2 matrix * Q2 vector
    Q2Q1 matrix * Q2 vector
    Q2Q1 matrix * Q1 vector
    array operations
    form right hand side
3 Apply inverses of patch matrices to patch right hand sides.
    apply matrix inverse (int)
    apply matrix inverse (ext)
4 Update global solution.
    update global solution
```



(a) simple Vanka



(b) tuned Vanka

Figure 3.7: Vanka: Kernels and their proportion of runtime, both (a) simple and (b) tuned Vanka.

accesses and, in turn, improve performance. For a uniform grid as considered here, fig. 3.8 sketches the grouping of patches into those that have a submatrix in common. Here, there are special cases for patches adjacent to the edges or corners of the mesh, including those associated with nodes on the boundary and those distance one from the boundary (where some degrees of freedom in the patch have Dirichlet boundary conditions applied), and a general case for all patches at nodes at least distance two from the boundary. This approach results in only about 40% of the overall runtime being taken up by applying patch matrix inverses. In total, just over three quarters of the runtime in the tuned approach is used for the four unique-to-Vanka operations. The other portion is contributed by the same simple matrix-vector operations as in the analysis of Braess-Sarazin. In what follows, we focus on tuned Vanka in our performance analysis and show a comparison of tuned and simple Vanka as part of our final comparison of relaxation schemes within multigrid-preconditioned FGMRES.

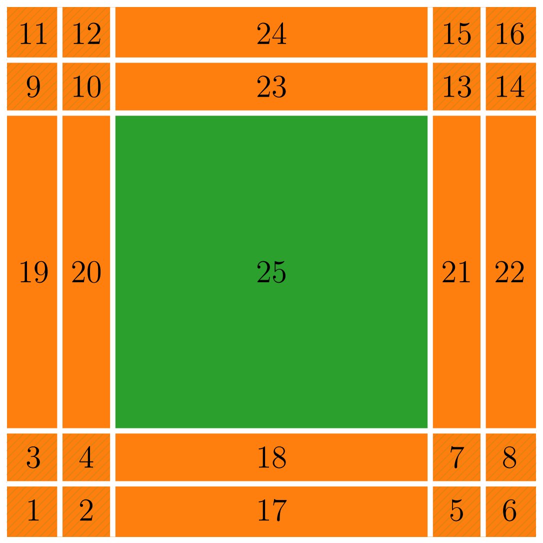


Figure 3.8: Vanka: groups of shared patch matrices

3.5.3 Vanka Patch Matrices

GPUs are built around multithreaded streaming multiprocessors. Whenever a kernel is launched from the host, all threads are grouped together into smaller thread blocks, which are then enumerated and distributed to available multiprocessors. All threads within a thread block are executed concurrently, and all blocks can be executed concurrently. Threads within a block are able to access local shared memory and can be synchronized. Additionally, all threads are able to access global memory. Choosing the right size of block is essential for good performance, as a too small block size leads to streaming multiprocessors that remain partially idle, whereas a too large block size leads to an imbalanced load over all of the streaming multiprocessors. CUDA is designed with a maximum of 1024 possible threads per thread block.

Making use of shared and local memory as much as possible within a CUDA thread block allows us to optimize memory accesses further, as data that is used repeatedly can be cached in memory that is faster than global memory. This is of particular importance for the Vanka algorithm, as many of the Vanka patches share the same patch matrices, as discussed above. In total, there are 25 different patch matrices, as depicted in fig. 3.8, for a constant-coefficient Stokes problem on a uniform mesh, independent of the number of elements. The shaded orange areas in fig. 3.8 denote single patches that have their own unique patch matrix. The normal orange areas are one-dimensional areas along element edges that share the same patch matrix, and the green area is the two-dimensional interior region of the domain, where all patches share the same submatrix. Within any one of these regions, we can load the patch matrix into shared memory once, to be used by all threads in the block.

kernel	reads	writes	flops
Q2 array plus/minus array	$2n$	n	n
Q2 array times scalar	$n+1$	n	n
Q2 matrix $*$ Q2 vector	$50+160\ell+128\ell^2$	n	$25+80\ell+64\ell^2$
Q2Q1 matrix $*$ Q2 vector	$50+100\ell+50\ell^2$	m	$25+50\ell+25\ell^2$
Q2Q1 matrix $*$ Q1 vector	$18+60\ell+50\ell^2$	n	$9+30\ell+25\ell^2$
Braess-Sarazin: weighted Jacobi	$2m+1$	m	$2m$
Vanka: Form Patch RHS	$51+102\ell+51\ell^2$	$51+102\ell+51\ell^2$	0
Vanka: Apply matrix inverse	$2652+5304\ell+2652\ell^2$	$51+102\ell+51\ell^2$	$2601+5202\ell+2601\ell^2$
Vanka: Update global solution	$52+104\ell+52\ell^2$	$52+104\ell+52\ell^2$	$52+104\ell+52\ell^2$

Table 3.1: Theoretical reads [double], writes [double] and flops of the various operations with $\ell+1$ as the number of nodal degrees of freedom in one dimension, n as the total number of velocity degrees of freedom, and m as the total number of pressure degrees of freedom.

3.5.4 Arithmetic Intensity

The *arithmetic intensity* of a kernel is defined as the ratio of how many floating point operations (flops) are performed per byte read/written. The algorithms for both Vanka and Braess-Sarazin involve various general vector and matrix-vector operations. In addition, Braess-Sarazin requires a weighted Jacobi application, and Vanka requires operations to extract the current residuals, to apply the patch matrix inverses, and to update the global solution. Table 3.1 denotes the counts of all reads, writes, and floating points operations (flops) for the various kernels, obtained by counting the operations in the algorithms. Note that the Q1 array operations are similar to the Q2 array operations leading to equivalent arithmetic intensity and performance values.

Based on the values in table 3.1, we compute the arithmetic intensity of the various kernels. On the GPU (using CUDA), we use the following formula,

$$\mathrm{AI} = \frac{\text{flops}}{32(\text{sectors read} + \text{sectors written})}. \tag{3.29}$$

where the reads and writes are counted per sector. One sector consists of a total of 32 bytes and, thus, we multiply by that constant in order to recover the byte count. The performance

kernel	AI	performance
array plus/minus array	0.0417	52.684
array times scalar	0.0625	79.026
Q2 matrix $*$ Q2 vector	0.0606	76.633
Q2Q1 matrix $*$ Q2 vector	0.0613	77.477
Q2Q1 matrix $*$ Q1 vector	0.0578	73.175
Braess-Sarazin: Jacobi	0.0833	105.368
Vanka: Form Patches	0.0	0.0
Vanka: Apply matrix	0.120	152.088
Vanka: Update solution	0.125	157.744

Table 3.2: Theoretical AI [flops/byte] and performance [GFLOP/s], calculated for a 512×512 element patch.

of each operation is then computed by

$$\text{perf} = \frac{\text{flops}}{\max\left(\frac{32(\text{sectors read + written})}{\text{bandwidth}}, \frac{\text{flops}}{\text{peak perf}}\right)} \tag{3.30}$$

The theoretical arithmetic intensity and performance computed this way is shown in table 3.2. This analysis, however, has its limitations. In practice, we expect the actual arithmetic intensity and performance to be better, as values are typically not read from or written to memory one-by-one. Instead, a memory range is typically loaded all at once, allowing us to reuse values. Additional strategies, like using shared memory for Vanka patches with the same patch matrix, further optimize the memory accesses, increasing both the arithmetic intensity and performance. Similarly, varying the size of CUDA blocks also has an effect on these quantities.

3.5.5 Common Kernels

For simplicity, we group the kernels into two classes. First, we examine those kernels that are common to both Vanka and Braess-Sarazin relaxation, involving matrix-vector products and array operations. Following this, we analyze the Vanka-specific kernels.

The common kernels are listed at the top of fig. 3.9, where we break down the matrix-vector products producing Q2 vectors into those that compute values at the nodes, denoted by n, the x- and y-edge midpoints, denoted by x and y, respectively, and the cell centers, denoted by c. The color-coding of these kernels matches that in fig. 3.6. For these kernels, we can choose the CUDA block size in an attempt to improve performance. Figure 3.9 shows how the arithmetic intensity, performance, and runtime vary for the various common kernels with varying CUDA block size.

We first note that the measured arithmetic intensities are indeed better than the theoretical values described in table 3.2, albeit generally not far off. Additionally, the measured arithmetic intensity does not vary much (or at all) with varying CUDA block size. This is due to the nature of the underlying memory operations, as the structured matrix data structures used here already optimize the loading and writing of memory. Due to the global nature of the kernels, they are not able to take advantage of shared memory on the GPU. The performance and runtime, however, are impacted by the CUDA block size, with increases in the performance leading to decreases in runtime. Over all results, we see differences in performance of up to a factor of 5 as we vary the block size. Choosing the best overall parameter comes down to selecting the best block size for the kernels that contribute the most to each algorithm.

For Braess-Sarazin, the Q2 matrix by Q2 vector multiplication makes up more than 50% of the overall runtime and, thus, choosing the best parameter for the 4 kernels within this operation has the largest impact on the overall runtime of the algorithm. For both problem sizes, the best (or near-best, within 2% of the best) runtime for these 4 kernels is achieved for a CUDA block size of 12×12. Analyzing the other common kernels yields a very similar picture. Thus, all of the common kernels achieve peak (or near-peak) performance for a CUDA block size of 12×12, which we choose for the Braess-Sarazin algorithm for which these kernels dominate the cost. We confirmed that these are the best choices by timing a full iteration of the algorithm for both problem sizes. Table 3.3 provides a concise overview of the best parameters for both algorithms.

3.5.6 Vanka-Specific Kernels

For the Vanka-specific kernels, we perform a similar analysis as for the common kernels. For all four kernels, we vary the thread block size from 4×4 to 16×16. Figure 3.10 shows how the arithmetic intensity, performance, and runtime varies with this parameter, again matching the color-coding used in fig. 3.7. Here, we notice that the measured arithmetic intensity is higher than the theoretical analysis in table 3.2, in particular for the kernels applying the patch inverses. This is expected, as we take advantage of fast shared memory for storing the shared patch matrices, which is not accounted for in that analysis. We note, however, that the arithmetic intensity does not vary much with block size, remaining largely constant. The kernel updating the global solution has a comparatively low arithmetic intensity, as it consists largely of memory movements and only very few floating-point operations.

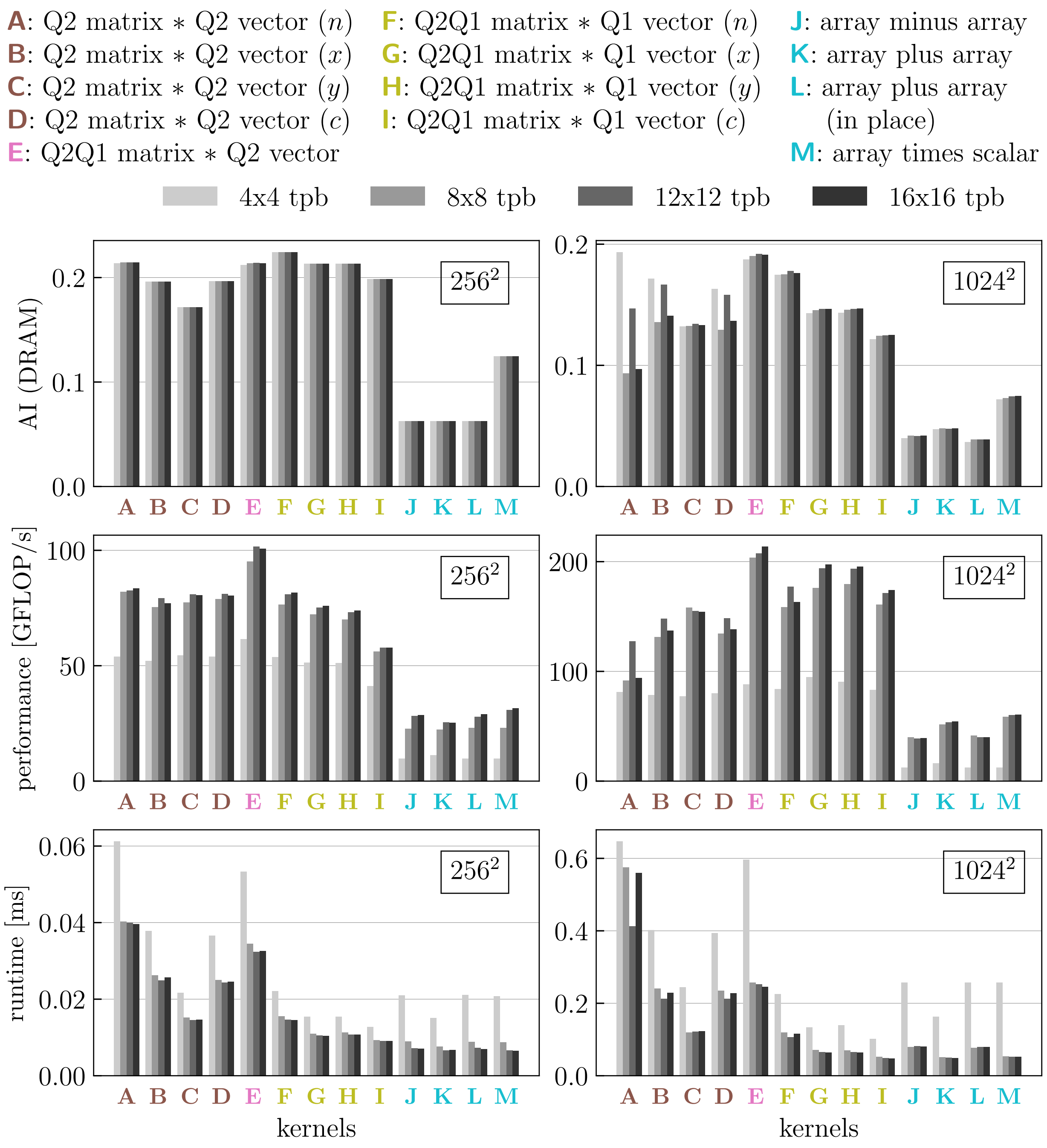


Figure 3.9: Common kernels: CUDA block size vs. AI, performance, and runtime. 256^2 elements in left column, 1024^2 elements in right column.

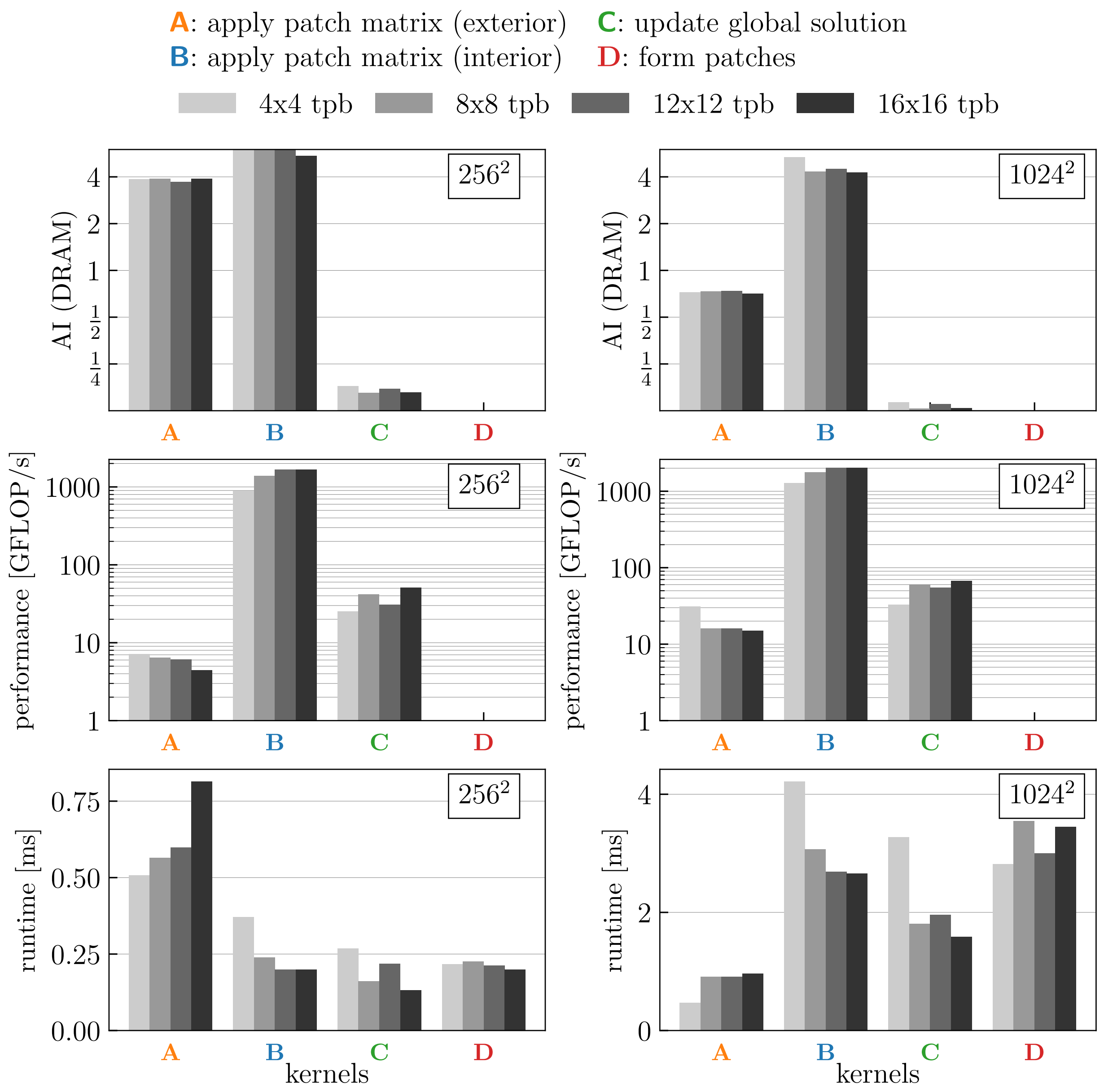


Figure 3.10: Vanka-specific kernels: thread-block size vs. AI, performance, and runtime. 256^2 elements in left column, 1024^2 elements in right column.

Similarly, the kernel forming the various Vanka patches does not contain any floating-point operations, resulting in zero arithmetic.

Analyzing the performance of the four kernels shows a rather similar picture, with the thread block size causing only small variations in the performance. Even though the kernel for the exterior patches and the kernel for the interior patches have a very similar arithmetic intensity, they differ widely in terms of performance, by up to 2 orders of magnitude. This is due to the comparatively high amount of work to be done for the interior patches. Once again, the kernel for forming the Vanka patches has a performance of 0 GFLOP/s, as it does not contain any floating point operations.

Both of these metrics, the arithmetic intensity and performance, are useful for evaluating the different kernels, but the effective runtime is the defining criteria for which any set of parameters is, ultimately, the best choice. Even though the kernels applying the patch matrices to the exterior patches (A) has a much lower performance than the kernel applying the patch matrices to the interior patches (B), the runtime of (A) for the smaller problem size is only about a factor of 3 larger than that for (B). For the larger problem size, the runtime of (A) is much lower than that for (B), by a factor of about 8. This is due to the overall relatively small amount of computations required for (A), as the exterior regions only grow linearly with the grid size in each dimension, whereas the interior region grows quadratically with (one-dimensional) grid size. Here, we can also see that the proportional runtime for the kernels (B), (C), and (D) is very much comparable, as already indicated in the kernel runtime breakdown in fig. 3.7.

Next, we investigate the effect of “grouping” computational threads, by passing more than one Vanka patch off to single CUDA thread within any one of the regions where the Vanka patches share the same patch matrix. This reduces the number of overall threads that need to be launched, while potentially further improving the memory accesses required. Figure 3.11 shows the runtime of the two sets of kernels applying the patch matrices for the four different thread block sizes, grouping patches together in groups of 1 to 64 patches per thread. From fig. 3.11, we see that increasing the number of patches per thread typically does not lead to a faster runtime; at best, the performance remains relatively constant. Thus, we do not pursue this any further and remain using one thread per patch.

The four Vanka-specific kernels make up more than 75% of the overall runtime of a Vanka iteration (see fig. 3.7), with each kernel taking up roughly the same proportion of overall runtime. To avoid unnecessary complexity in the code, we choose a single CUDA block size to use for the entire algorithm and all connected kernels. For the smaller problem size, a CUDA block size of 8×8 is not the optimal choice for many of the individual kernels, but all four kernels exhibit near-peak performance for this CUDA block size. For the larger problem

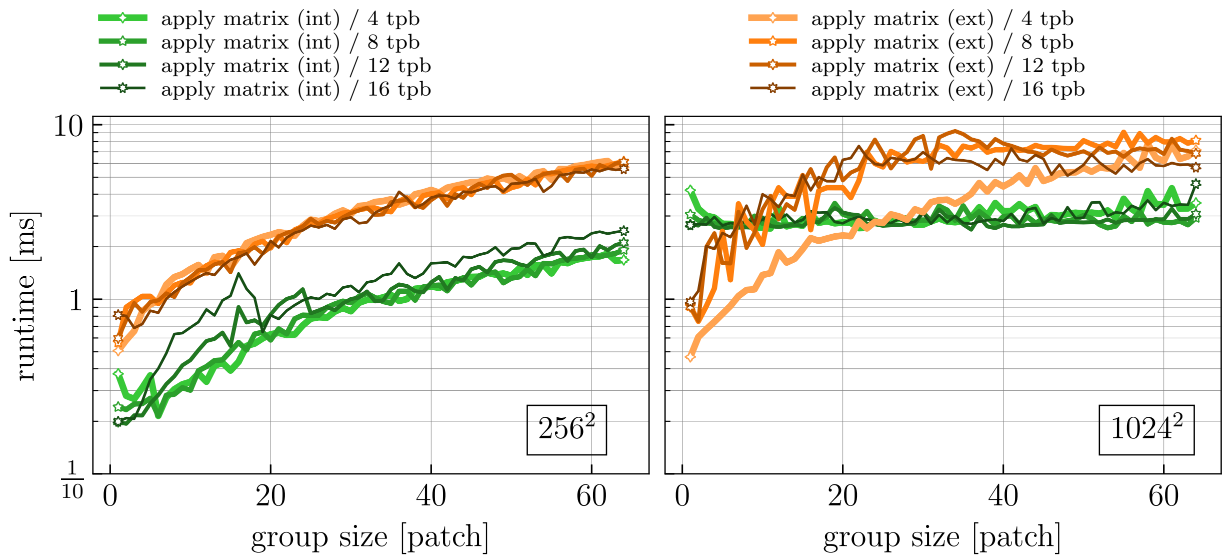


Figure 3.11: Vanka-specific kernels: Group size vs. runtime for 256^2 elements (left) and 1024^2 elements (right).

size, the best choice of CUDA block size is 12×12. We have also confirmed that these are the best choices by timing a full iteration of the algorithm for both problem sizes. Table 3.3 provides a concise overview of the best parameters for both algorithms.

Algorithm	# elements	threads/block
Braess-Sarazin	256^2	12×12
	1024^2	12×12
Vanka	256^2	8×8
	1024^2	12×12

Table 3.3: Best parameter choices for both algorithms.

3.5.7 Roofline Model

Having analyzed the kernels above and selected the optimal thread block size, we now consider a roofline model to measure for how efficient the kernels are on a given GPU. Such models tell us whether an operation is memory or compute bound, and whether all theoretically available computing power is used. Figure 3.12 shows two roofline models, one for each of the two problem sizes, showing measured performance vs. arithmetic intensity for each kernel in a Braess-Sarazin or Vanka relaxation sweep. Here, we very clearly see that, for the larger problem size, most kernels lie right on or very close to the performance

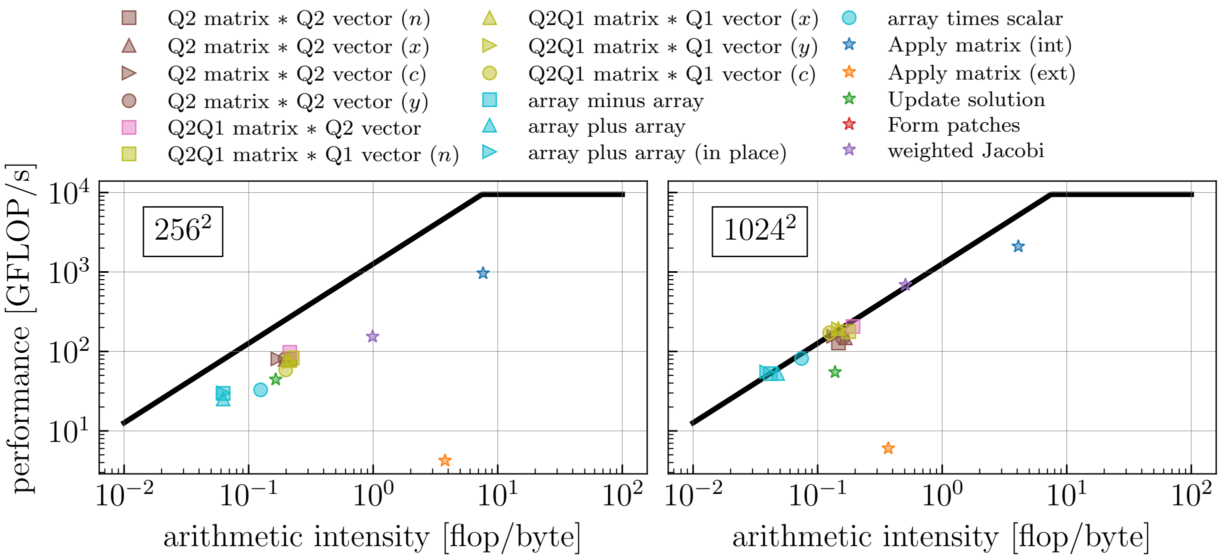


Figure 3.12: Roofline Model for all kernels

bound, meaning that they are running as fast as possible given their arithmetic intensity. In order to improve their performance, we would need to find ways to increase their arithmetic intensity. However, given the nature of these kernels and the underlying structured matrix data structures, there is not an obvious avenue to do this.

Even though most of the kernels (all the matrix-vector and array operations) are clustered together, there are four outliers in this data that we want to highlight:

1. The first outlier is the kernel corresponding to forming the Vanka patch submatrices, which is not visible in the plot, as it consists entirely of memory movements and no floating-point operations. Its arithmetic intensity and floating-point performance are, thus, 0.

2. The second outlier is the kernel applying the patch matrix inverse to the exterior patches of the domain. This has a low peak performance of only 6 GFLOP/s, as it consists of mostly small operations (16 unique patch matrices, with 8 one-dimensional regions sharing a patch matrix). It also acts on little enough data that, even for the large problem size, we do not achieve the performance expected from the roofline model.

3. The third outlier is a kernel that we mostly ignored in our analysis, the weighted Jacobi kernel. This kernel achieves a higher performance than all but one other kernel, with a peak performance of 753 GFLOP/s. However, it contributes less than 2% to the overall runtime of Braess-Sarazin relaxation, with similar percentages of runtime

expected for the other algorithms that use it. Thus, even though its performance is rather high, it has barely any measurable effect on the algorithm runtime.

4. The final outlier is the kernel applying the patch matrix inverse to the interior patches of the domain. Its peak performance is roughly 2145 GFLOP/s, almost three times as high as the next highest kernel. With this high performance, it still makes up about 20% of the overall runtime of Vanka. Thus, achieving this performance on this single kernel results in the overall Vanka algorithm achieving much better performance.

Overall, we note that most of the kernels achieve their maximum possible performance, as they lie right on the performance limit in the roofline model for the larger problem size. Due to the nature of their operations, increasing their arithmetic intensity is not possible and, thus, the performance of these algorithms cannot reasonably be expected to be increased. One avenue to consider to improve the performance of the kernels that require a disproportionately large volume of memory movement would be to try to "trade" some memory movement for increasing numbers of floating-point operations; this will be a subject for future research.

3.5.8 Overall Solver Performance

Finally, having analyzed and optimized the performance of the Vanka and Braess-Sarazin relaxation schemes, we now look to see how they compare in practice, when used as relaxation schemes inside of a multigrid V-cycle that is used as preconditioner for FGMRES applied to the Stokes equations. We will also compare their performances to the performance of FGMRES preconditioned with a multigrid V-cycle with Schur-Uzawa and preconditioned with a Block-Triangular preconditioner with multigrid V-cycles used to approximate the block inverses. The additional parameters needed for Schur-Uzawa and the Block-Triangular preconditioner have been determined through further experiment. The optimal value of t in the Schur-complement scheme is 1 with an optimal Jacobi weight of $\omega = 0.4$. For the Block-Triangular preconditioner, we determined that a total of 3 V-cycles are necessary for both the pressure update solve and velocity update solve, and the two respective weights for the weighted Jacobi relaxation are $\omega = 0.6$ for the pressure update solve, and $\omega = 1.0$ for the velocity update solve.

The choice of outer Krylov method for a linear solve requires considering many factors. Here, because the preconditioners are not guaranteed to be symmetric and positive definite, we must use a general Krylov method instead of a specialized technique like CG or MINRES. We choose to use FGMRES for two reasons. First, we find that right preconditioning is a preferable framework to left preconditioning, since it does not change the norm of the underlying minimization. Secondly, all of the chosen components lead to preconditioned FGMRES algorithms that converge in tens of iterations, so the additional memory costs for vector storage in FGMRES are feasible (even on the GPU) and preferable to the added computational cost of an extra preconditioner application that is needed in classical GMRES. We note, however, that none of the conclusions from this study would be substantially changed by using classical right-preconditioned GMRES.

We use our own implementation of FGMRES, making use of our structured data structures, and use a multigrid `V(1,1)` cycle as preconditioner, and a `V(3,3)` cycle as part of the Block-Triangular preconditioner. At each level of the multigrid algorithm, we use a sweep of either Braess-Sarazin, Vanka, or Schur-Uzawa relaxation. With the Block-Triangular preconditioner, we use three sweeps of weighted Jacobi. At the coarsest level, we use either an exact solve on the CPU or three sweeps of the relaxation scheme on the GPU.

The first comparison we consider is a comparison of Braess-Sarazin to both our tuned Vanka implementation described in this chapter and a simple Vanka implementation, shown in fig. 3.13. All of the results are for problems of size 1024^2, with the exception of the

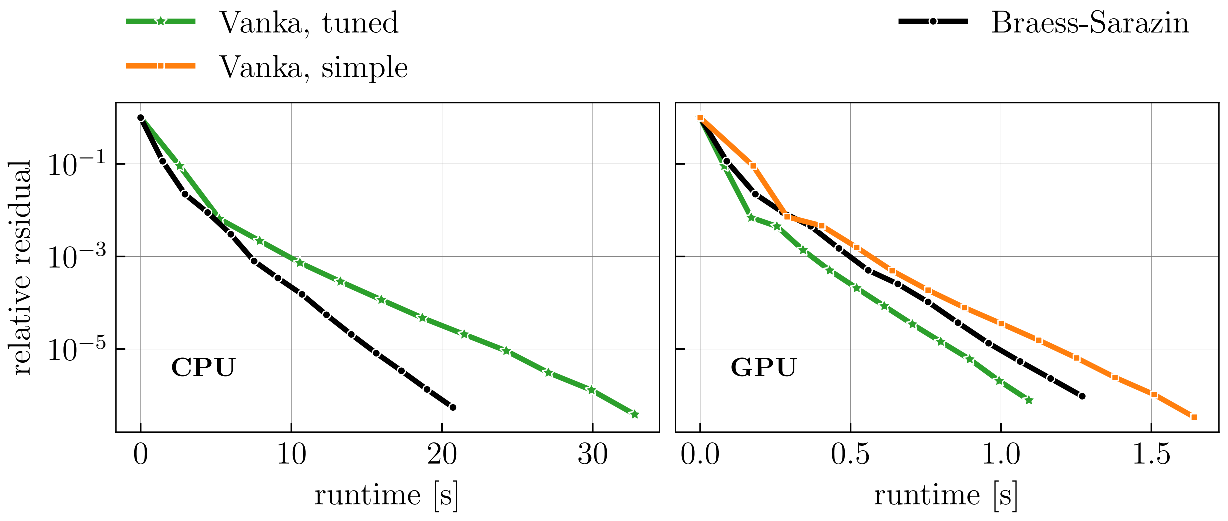


Figure 3.13: Comparing Vanka to Braess-Sarazin for a problem of size 1024^2 elements (768^2 elements for simple Vanka)

simple Vanka runs. Due to its higher memory requirements, the largest problem size that successfully ran was a problem of size 768^2 elements. However, even though simple Vanka has just over half as many elements as the other approaches, it is still not able match their performance. On the CPU, we see that multigrid with Braess-Sarazin relaxation strongly outperforms the use of Vanka relaxation. Even though multigrid with Vanka relaxation typically requires one fewer iteration to reach convergence, the work required per iteration is significantly larger than for Braess-Sarazin, resulting in multigrid with Vanka taking about twice as long. On the GPU, however, we are able to take advantage of the throughput of Vanka, resulting in a runtime that is more than 23 times smaller than on the CPU, whereas the runtime for Braess-Sarazin is only reduced by a factor of about 11. Overall, on the GPU, tuned Vanka outperforms Braess-Sarazin by about 10%.

Next, we compare Braess-Sarazin and tuned Vanka to the other two preconditioning strategies, monolithic multigrid with Schur-Uzawa and the Block-Triangular preconditioner, shown in fig. 3.14. We can see that both the multigrid preconditioner with Schur-Uzawa relax-

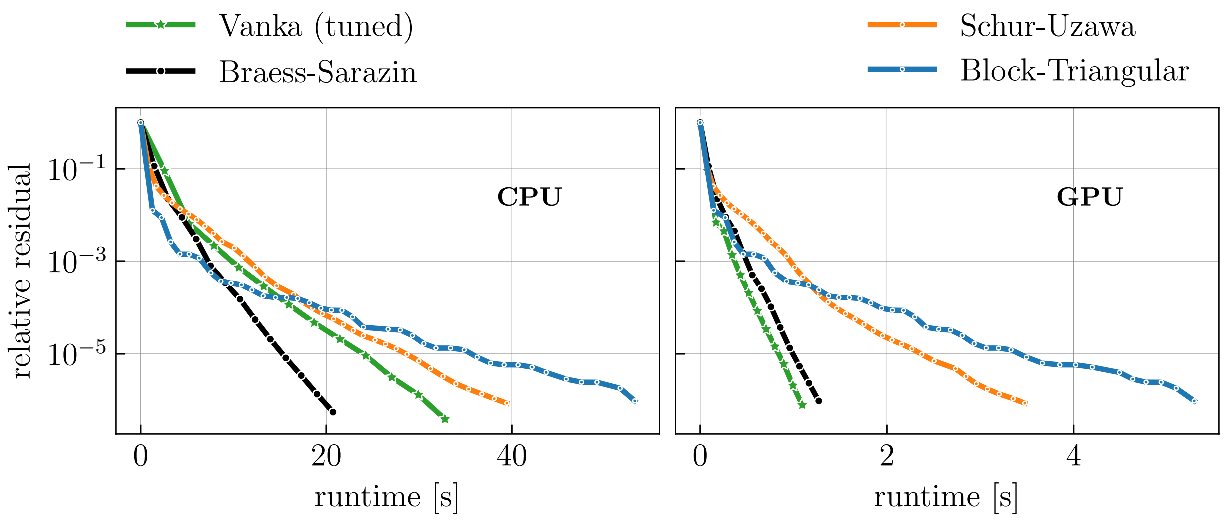


Figure 3.14: Comparing all four preconditioning strategies for a problem of size 1024^2 elements

ation and the Block-Triangular preconditioner are not able to match the performance of both Braess-Sarazin and Vanka. Initially they perform very well, in particular the Block-Triangular preconditioner, but they soon slow down requiring up to more than 3 times as long as Braess-Sarazin and Vanka (on the GPU).

The third metric to consider is the performance of our tuned Vanka implementation for two problem sizes when normalized per element on the CPU and per row of elements on the GPU; this is shown in fig. 3.15. We observe that the time for the tuned Vanka implementation (per element) remains the same no matter the problem size on the CPU, requiring about 0.03ms per element. Thus, there is no additional overhead introduced by the size of the problem. On the GPU, we are able expose the fine-grained parallelism in the tuned Vanka implementation, resulting in a constant work per row of elements at just over 1 ms. In fact, we are able to perform about 10% faster per row of elements for the larger problem size.

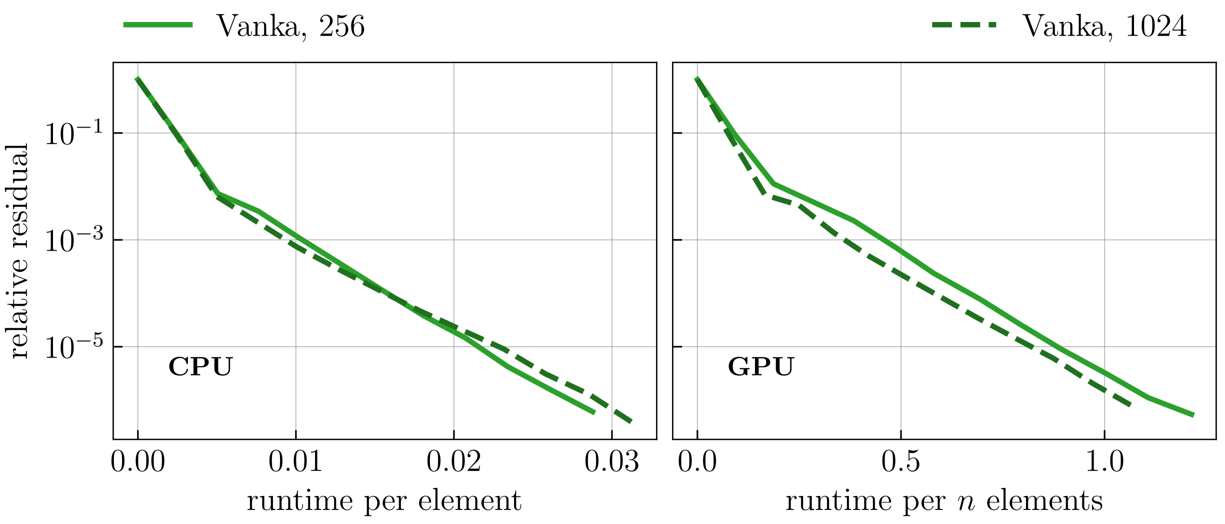


Figure 3.15: Showing the work per element (in ms) of tuned Vanka for 256^2 and 1024^2 elements.

These results show that a careful implementation of Vanka on the GPU not only results in the fastest time to convergence, but it also does so without requiring additional parameters to be set. In addition, the parallelism of Vanka makes it a clear favorite in distributed memory settings.

3.6 CONCLUSION

Several preconditioners for FGMRES are well-known to yield scalable solution algorithms for saddle-point problems, such as the Stokes equations, including both monolithic multigrid and multigrid-based block-factorization preconditioners. Here, we consider their implementation, performance, and optimization, on modern CPU and GPU architectures. Different metrics were presented and analyzed, including arithmetic intensity, performance, and runtime, for the various kernels making up these algorithms. Given a highly structured setup, we show that multigrid with Vanka relaxation can be very performant on the GPU, leading to faster convergence than when using Braess-Sarazin, both in terms of iterations (saving just 1 iteration) and runtime (up to 10% faster). This shows that using Vanka relaxation is both mathematically and computationally competitive, although a careful design of the algorithm is warranted. This also highlights the benefit of using GPUs for such algorithms, as multigrid with Vanka on the GPU is up to 23 times faster than on the CPU, while multigrid with Braess-Sarazin is up to 11 times faster.

We also presented two other preconditioning strategies, multigrid preconditioner with

Schur-Uzawa relaxation, and Block-Triangular preconditioner with multigrid and weighted Jacobi within. Both of these have been shown to not be able to compete with multigrid with Braess-Sarazin or Vanka, in particular on the GPU. In addition, they introduce additional parameters that need to be carefully chosen.

Future work includes extending this work to cases where the tuned Vanka approach is not applicable, such as for linearizations of the Navier-Stokes equations. It is also not clear how well these results generalize to other discretizations of saddle-point systems (including both higher-order discretizations using generalized Taylor-Hood elements and other discretizations, such as using discontinuous Galerkin methods). Extensions to three-dimensional incompressible flow problems and other saddle-point systems are also important future work. One such system of interest, for example, that combines some of these difficulties is the Reynolds-Averaged Navier-Stokes (RANS) equations, in the context of wind-turbine simulations.

CHAPTER 4: RAS+ILU FOR THE REYNOLDS-AVERAGED NAVIER-STOKES EQUATIONS

In this chapter model turbulence around a wind turbine. The turbulence flow can be described using the Reynolds-averaged Navier-Stokes (RANS) equations. We re-derive these equations and explore different ways the Reynolds-stress tensor can be handled. Even so, for the rest of the chapter we set the Reynolds-stress tensor to zero and work with a direct numerical simulation (DNS).

The focus of this work is on the use of restrictive additive Schwarz (RAS) paired with incomplete LU (ILU) as part of a new solver for the Navier-Stokes equations. We explore various existing approaches and see how convergence is limited for our model problem. We introduce a new solver that is capable of solving our model problem for moderate to large Reynolds numbers. After analyzing its performance we further explore homotopy (or continuation methods) and how faster convergence can be achieved. They also allow the solving of even more difficult problems. Lastly, we develop a dynamic version of our solver that is capable of self-tuning the continuation steps.

4.1 REYNOLDS-AVERAGED NAVIER-STOKES (RANS) EQUATIONS

The Reynolds-averaged Navier-Stokes (RANS) equations [59] are time-averaged equations that model fluid flow and are used to describe turbulent flows. To derive these equations, we start with the Navier-Stokes equations as described by

$$\frac{\partial \mathbf{u}}{\partial t} + \mathbf{u} \cdot \nabla \mathbf{u} = -\nabla p + \nu \nabla^2 \mathbf{u} + \mathbf{f} \tag{4.1}$$

$$\nabla \cdot \mathbf{u} = 0 \tag{4.2}$$

where $\mathbf{u}$ is the fluid velocity, p is the pressure divided by the density, and ν is the velocity.

In order to derive the RANS equations from these two equations, we start by using a Reynolds decomposition such that the equations eq. (4.1) and eq. (4.2) are decomposed into a time-averaged (mean) and a fluctuating quantity, $\mathbf{u} = \bar{\mathbf{u}} + \mathbf{u'}$ and $p = \bar{p} + p'$. From there, the system of PDEs describing turbulent flows is given as

$$\frac{\partial \mathbf{u}}{\partial t} + \overline{\mathbf{u} \cdot \nabla \mathbf{u}} = -\nabla \bar{p} + \nu \nabla^2 \bar{\mathbf{u}} + \mathbf{f} \tag{4.3}$$

$$\nabla \cdot \bar{\mathbf{u}} = 0 \tag{4.4}$$

where the mean is computed as an ensemble average,

$$\bar{\phi}^E(\mathbf{x},t) = \lim_{N\to\infty} \frac{1}{N} \sum_{k=1}^{N} \phi^k(\mathbf{x},t) \tag{4.5}$$

where ϕ is a generic flow variable, and N is the number of repeated experiments. For a turbulent spatially homogeneous flow and turbulent statistically steady state, it is assumed that the time average, $\bar{\phi}^T$, and the volume average, $\bar{\phi}^V$, is equal to the ensemble average $\bar{\phi}^E$. In eq. (4.3) we make use of the property

$$\begin{aligned}\overline{\mathbf{u}\mathbf{u}} &= \overline{(\bar{\mathbf{u}}+\mathbf{u}')(\bar{\mathbf{u}}+\mathbf{u}')} \\ &= \overline{\bar{\mathbf{u}}\bar{\mathbf{u}}} + \overline{\bar{\mathbf{u}}\mathbf{u}'} + \overline{\mathbf{u}'\bar{\mathbf{u}}} + \overline{\mathbf{u}'\mathbf{u}'} \\ &= \bar{\mathbf{u}}\bar{\mathbf{u}} + \overline{\mathbf{u}'\mathbf{u}'}\end{aligned} \tag{4.6}$$

noting that $\overline{\bar{\mathbf{u}}\mathbf{u}'} = \overline{\mathbf{u}'\bar{\mathbf{u}}}$ [60], to obtain the equation

$$\frac{\partial \mathbf{u}}{\partial t} + \bar{\mathbf{u}} \cdot \nabla \bar{\mathbf{u}} = -\nabla \bar{p} + \nu \nabla^2 \bar{\mathbf{u}} - \nabla \tau + \mathbf{f} \tag{4.7}$$

where $\tau = \overline{\mathbf{u}'\mathbf{u}'}$ is the Reynolds-stress term. This is equivalent to the Reynolds-stress tensor divided by the density. Its diagonal entries correspond to the normal stresses, and its off-diagonal components correspond to the shear stresses. Integration in time removes the time dependence, resulting in the steady-state RANS equations

$$\bar{\mathbf{u}} \cdot \nabla \bar{\mathbf{u}} = -\nabla \bar{p} + \nu \nabla^2 \bar{\mathbf{u}} - \nabla \tau + \mathbf{f} \tag{4.8}$$

$$\nabla \cdot \bar{\mathbf{u}} = 0 \tag{4.9}$$

The difficulty in solving the RANS equations as described in eq. (4.8) and eq. (4.9) is that the Reynolds-stress tensor introduces six additional unknowns, resulting in an unclosed system of equations.

4.1.1 Weak Form and Stabilization

We study the RANS equations discretized by a finite element method. Let $\mathbf{H}_0^1(\Omega)$ be the Hilbert space defined as

$$\mathbf{H}_0^1(\Omega) = \{\mathbf{v} \in \mathbf{H}^1(\Omega) : \mathbf{v} \cdot \mathbf{n} = 0 \text{ on } \delta\Omega\} \tag{4.10}$$

and define the finite-dimensional subspaces $\mathbf{X}_0^h \subset \mathbf{H}_0^1$ and $M^h \subset L_2(\Omega)$. The weak formulation of the Reynolds-averaged Navier-Stokes equations given in eqs. (4.8) and (4.9) is then given by: Find $\mathbf{u} \in \mathbf{X}_0^h$ and $p \in M^h$ such that

$$\nu\left(\nabla \cdot \mathbf{u}, \nabla \cdot \mathbf{v}\right) + (\nabla \mathbf{u}, \mathbf{v}) + (\nabla \cdot \mathbf{v}, p) + (\tau, \nabla \mathbf{v}) = (\mathbf{f}, \mathbf{v}) \quad \forall \mathbf{v} \in \mathbf{X}_0^h \tag{4.11}$$

$$(\nabla \mathbf{u}, q) = 0 \qquad \forall q \in L_2(\Omega). \tag{4.12}$$

The weak form as described in eq. (4.11) and eq. (4.12) is very similar to the weak form of the standard Navier-Stokes equations. However, the addition of the Reynolds-stress tensor requires additional steps to be taken in order to model that quantity.

4.2 MODELING THE REYNOLDS-STRESS TENSOR

An important task when working with the RANS equation is being able to model the Reynolds-stress tensor in eq. (4.7). This is done either directly as with the eddy-viscosity models, or indirectly by solving additional PDEs in closure models. There are a few main classes of models that are commonly used for this purpose [59]:

1. **Zero-equation models**: In zero equation models only the mean-velocity field is solved using a system of PDEs,

$$\frac{\partial \bar{u}_i}{\partial t} + \bar{u}_j \frac{\partial \bar{u}_i}{\partial x_j} = -\frac{\partial \bar{p}}{\partial x_i} + \nu \frac{\partial^2 \bar{u}_i}{\partial x_j \partial x_j} - \frac{\partial \tau_{ij}}{\partial x_j}, \tag{4.13}$$

$$\frac{\partial \bar{u}_i}{\partial x_i} = 0, \tag{4.14}$$

$$\tau_{ij} = \frac{2}{3} K \delta_{ij} - \nu_T \left(\frac{\partial \bar{u}_i}{\partial x_j} + \frac{\partial \bar{u}_j}{\partial x_i} \right), \tag{4.15}$$

   where K is the average kinetic energy of the velocity fluctuations, $K = \frac{1}{2}\overline{u_i' u_i'}$, and ν_T is the eddy viscosity, $\nu_T \propto \frac{l_0^2}{t_0}$, where l_0 and t_0 the turbulence length and time scales, both obtained empirically and provided algebraically. In incompressible flows, τ_{ij}' can be incorporated into the mean-pressure flow in eq. (4.13) and thus the term involving K disappears alleviating the need to compute K.

2. **One-equation models**: One-equation models extend the zero-equation models by adding an additional transport equation to calculate the turbulence kinetic energy. They are also able to account for some non-local and history effects in the definition

of the eddy viscosity. The system of equations is described as

$$\frac{\partial \bar{u}_i}{\partial t} + \bar{u}_j \frac{\partial \bar{u}_i}{\partial x_j} = -\frac{\partial \bar{p}}{\partial x_i} + \nu \frac{\partial^2 \bar{u}_i}{\partial x_j \partial x_j} - \frac{\partial \tau_{ij}}{\partial x_j}, \tag{4.16}$$

$$\frac{\partial \bar{u}_i}{\partial x_i} = 0, \tag{4.17}$$

$$\frac{\partial K}{\partial t} + \bar{u}_i \frac{\partial K}{\partial x_i} = -\tau_{ij} \frac{\partial \bar{u}_i}{\partial x_j} - C^* \frac{K^{3/2}}{l_0} + \frac{\partial}{\partial x_i}\left(\frac{\nu_T}{\sigma_K}\frac{\partial K}{\partial x_i}\right) + \nu \frac{\partial^2 K}{\partial x_i \partial x_i}, \tag{4.18}$$

$$\tau_{ij} = \frac{2}{3} K \delta_{ij} - \nu_T \left(\frac{\partial \bar{u}_i}{\partial x_j} + \frac{\partial \bar{u}_j}{\partial x_i}\right), \tag{4.19}$$

$$\nu_T = K^{1/2} l_0 \tag{4.20}$$

where K is the average kinetic energy of the velocity fluctuations, $K = \frac{1}{2}\overline{u_i' u_i'}$, ν_T is the eddy viscosity, l_0 is the turbulence length, and $\sigma_K \approx 1.0$ and $C^* = 0.166$ are two nondimensional constants.

3. **Two-equation models**: Two-equation models further extend the one-equation model by an additional transport equation. These two additional equations are solved for two independent quantities associated with turbulence, directly related to the turbulence length and time scales.

   - K-ϵ models: One such two-equation model is the so-called (K-ϵ) model where the turbulence length and time scales are constructed from K, the turbulent kinetic energy, and ϵ, the turbulent dissipation rate,

$$l_0 \propto \frac{K^{3/2}}{\epsilon}, \quad \tau_0 \propto \frac{K}{\epsilon}. \tag{4.21}$$

This leads to the system of PDEs,

$$\frac{\partial \bar{u}_i}{\partial t} + \bar{u}_j \frac{\partial \bar{u}_i}{\partial x_j} = -\frac{\partial \bar{p}}{\partial x_i} + \nu \frac{\partial^2 \bar{u}_i}{\partial x_j \partial x_j} - \frac{\partial \tau_{ij}}{\partial x_j}, \tag{4.22}$$

$$\frac{\partial \bar{u}_i}{\partial x_i} = 0, \tag{4.23}$$

$$\frac{\partial K}{\partial t} + \bar{u}_i \frac{\partial K}{\partial x_i} = -\tau_{ij} \frac{\partial \bar{u}_i}{\partial x_j} - \epsilon + \frac{\partial}{\partial x_i}\left(\frac{\nu_T}{\sigma_K}\frac{\partial K}{\partial x_i}\right) + \nu \frac{\partial^2 K}{\partial x_i \partial x_i} \tag{4.24}$$

$$\frac{\partial \epsilon}{\partial t} + \bar{u}_i \frac{\partial \epsilon}{\partial x_i} = -C_{\epsilon_1}\frac{\epsilon}{K}\tau_{ij}\frac{\partial \bar{u}_i}{\partial x_j} + \frac{\partial}{\partial x_i}\left(\frac{\nu_T}{\sigma_\epsilon}\frac{\partial \epsilon}{\partial x_i}\right) - C_{\epsilon_2}\frac{\epsilon^2}{K} + \nu \frac{\partial^2 \epsilon}{\partial x_i \partial x_i} \tag{4.25}$$

$$\tau_{ij} = \frac{2}{3}K\delta_{ij} - \nu_T\left(\frac{\partial \bar{u}_i}{\partial x_j} + \frac{\partial \bar{u}_j}{\partial x_i}\right) \tag{4.26}$$

$$\nu_T = C_\mu \frac{K^2}{\epsilon} \tag{4.27}$$

with the constants $C_{\epsilon_1} = 1.44$, $C_{\epsilon_2} = 1.92$, $C_\mu = 0.09$, $\sigma_K = 1.0$, and $\sigma_\epsilon = 1.3$.

- K-l models: The K-l models are very similar to the K-ϵ models. They are based on the solution of the transport equation for the turbulent kinetic energy K (see eq. (4.18)) and the solution of the transport equation for the integral length scale l,

$$\frac{\partial (Kl)}{\partial t} + \bar{u}_i \frac{\partial (Kl)}{\partial x_i} = \frac{\partial}{\partial x_i}\left[(\nu + \beta_1 K^{1/2} l)\frac{\partial}{\partial x_i}(Kl) + \beta_2 K^{3/2} l \frac{\partial l}{\partial x_i}\right] - \beta_3 l \tau_{ij}\frac{\partial \bar{u}_i}{\partial x_i} - \beta_4 K^{3/2}, \tag{4.28}$$

  where β_1, β_2, β_3, and β_4 are empirical constants. In particular for the case of homogeneous flows, the K-l model can be shown to be equivalent to the K-ϵ model with different values for the constants C_μ, $C_{\epsilon 1}$, and $C_{\epsilon 2}$.

- K-ω models: Another variation of the two-equation models are the K-ω models which as the K-l model are also based on the solution of the transport equation for the turbulent kinetic energy K (see eq. (4.18)), but also on the solution of the equation for the reciprocal turbulent time scale ($\omega = \frac{\epsilon}{K}$),

$$\frac{\partial \omega}{\partial t} + \bar{u}_i \frac{\partial \omega}{\partial x_i} = -\gamma_1 \frac{\omega}{K}\tau_{ij}\frac{\partial \bar{u}_i}{\partial x_j} + \frac{\partial}{\partial x_i}\left(\frac{\nu_T}{\sigma_\omega}\frac{\partial \omega}{\partial x_i}\right) - \gamma_2 \omega^2 + \nu \frac{\partial^2 \omega}{\partial x_i}\partial x_i \tag{4.29}$$

  where $\nu_T = \frac{\gamma^* K}{\omega}$, γ_1, γ_2, γ^*, and σ_ω are constants.

- Nonlinear eddy-viscosity models: The nonlinear eddy-viscosity models create closures for the Reynolds-stress tensor that are nonlinear in the mean strains,

$$\begin{aligned}\tau_{ij} =& \frac{2}{3}K\delta_{ij} - 2\frac{l_0^2}{\tau_0}\bar{S}_{ij} + \alpha_1 l_0^2\left(\bar{S}_{ik}\bar{S}_{kj} - \frac{1}{3}\bar{S}_{mn}\bar{S}_{mn}\delta_{ij}\right) \\ &+ \alpha_2 l_0^2\left(\bar{S}_{ik}\bar{W}_{jk} + \bar{S}_{jk}\bar{W}_{ik}\right) \\ &+ \alpha_3 l_0^2\left(\bar{W}_{ik}\bar{W}_{kj} - \frac{1}{3}\bar{W}_{mn}\bar{W}_{mn}\delta_{ij}\right) \\ &+ \alpha_4 l_0^2\left(\frac{\partial\bar{S}_{ij}}{\partial t} + \bar{\mathbf{u}}\cdot\nabla\bar{S}_{ij}\right),\end{aligned} \tag{4.30}$$

where $\bar{S}_{ij}$ is the mean strain-rate tensor, $\bar{W}_{ij}$ is the mean-vorticity tensor, and α_1, α_2, α_3, and α_4 are dimensionless constants.

4. **Stress-equation models**: Stress-equation models, also called τ_{ij}-ϵ or second-order models, involve the solution of modeled equations for the Reynolds-stress and dissipation-rate transport equations. They do not require approximations to the Reynolds-stress term on the right-hand side of eq. (4.8). The typical form of the PDEs to be solved in this class of models can be described as,

$$\frac{\partial\bar{u}_i}{\partial t} + \bar{u}_j\frac{\partial\bar{u}_i}{\partial x_j} = -\frac{\partial\bar{p}}{\partial x_i} + \nu\frac{\partial^2\bar{u}_i}{\partial x_j\partial x_j} - \frac{\partial\tau_{ij}}{\partial x_j}, \tag{4.31}$$

$$\frac{\partial\bar{u}_i}{\partial x_i} = 0, \tag{4.32}$$

$$\begin{aligned}\frac{\partial\tau_{ij}}{\partial t} + \bar{u}_k\frac{\partial\tau_{ij}}{\partial x_k} =& -\tau_{ij}\frac{\partial\bar{u}_j}{\partial x_k} - \tau_{jk}\frac{\partial\bar{u}_i}{\partial x_k} + \epsilon A_{ij} + KM_{ijkl}\frac{\partial\bar{u}_k}{\partial x_l} - \frac{2}{3}\epsilon\delta_{ij} \\ &+ C_s\frac{K}{\epsilon}\frac{\partial}{\partial x_k}\left(\tau_{im}\frac{\partial\tau_{jk}}{\partial x_m} + \tau_{jm}\frac{\partial\tau_{ik}}{\partial x_m} + \tau_{km}\frac{\partial\tau_{ij}}{\partial x_m}\right) \\ &+ \nu\frac{\partial^2\tau_{ij}}{\partial x_k\partial x_k}\end{aligned} \tag{4.33}$$

$$\frac{\partial\epsilon}{\partial t} + \bar{u}_i\frac{\partial\epsilon}{\partial x_i} = -C_{\epsilon 1}\frac{\epsilon}{K}\tau_{ij}\frac{\partial\bar{u}_i}{\partial x_j} + C_\epsilon\frac{\partial}{\partial x_i}\left(\frac{K}{\epsilon}\tau_{ij}\frac{\partial\epsilon}{\partial x_j}\right) - C_{\epsilon 2}\frac{\epsilon^2}{K} + \nu\frac{\partial^2\epsilon}{\partial x_i\partial x_i} \tag{4.34}$$

where A_{ij} and M_{ijkl} are functions of the energy-spectrum sensor in time and wave-number space, $C_s \approx 0.11$, $C_{\epsilon 1} = 1.44$, and $C_{\epsilon 2} = 1.92$

5. **Algebraic-stress turbulence models**: Based on the fact that nonlinear anisotropic models for the Reynolds-stress tensor can be derived algebraically from an analysis of the Reynolds-stress equation, additional two-equation models of the K-ϵ type can be constructed. It comes with the additional assumption that the turbulence is locally

homogeneous and in equilibrium. The resulting system of PDEs then takes the form,

$$\frac{\partial \bar{u}_i}{\partial t} + \bar{u}_j \frac{\partial \bar{u}_i}{\partial x_j} = -\frac{\partial \bar{p}}{\partial x_i} + \nu \frac{\partial^2 \bar{u}_i}{\partial x_j \partial x_j} - \frac{\partial \tau_{ij}}{\partial x_j}, \tag{4.35}$$

$$\frac{\partial \bar{u}_i}{\partial x_i} = 0, \tag{4.36}$$

$$\frac{\partial K}{\partial t} = -\tau_{ij}\frac{\partial \bar{u}_i}{\partial x_j} - \epsilon \tag{4.37}$$

$$\frac{\partial \epsilon}{\partial t} = -C^*_{\epsilon 1}\frac{\epsilon}{K}\tau_{ij}\frac{\partial \bar{u}_i}{\partial x_j} - C_{\epsilon 2}\frac{\epsilon^2}{K} \tag{4.38}$$

$$\begin{aligned}\tau_{ij} = \frac{2}{3}K\delta_{ij} - \frac{3}{3-2\eta^2+6\xi^2}\Bigg[&\alpha_1\frac{K^2}{\epsilon}\bar{S}_{ij} \\ &+ \alpha_2\frac{K^3}{\epsilon^2}(\bar{S}_{ik}\bar{W}_{kj} + \bar{S}_{jk}\bar{W}_{ki}) \\ &-\alpha_3\frac{K^3}{\epsilon^2}\left(\bar{S}_{ik}\bar{S}_{kj} - \frac{1}{3}\bar{S}_{kl}\bar{S}_{kl}\delta_{ij}\right)\Bigg]\end{aligned} \tag{4.39}$$

A common choice [61] for modeling the Reynolds-stress tensor are the zero-equation models. They can be used rather easily as they only calculate the mean-velocity and pressure quantities. That ease of use comes at the cost of accuracy, though, as they do not take history effects into account and require a turbulent length scale specific to each problem. Increasing the complexity with the one- and two-equation models allows for a more accurate modeling at the expense of computational cost, although they are also not able to take all physical properties of the underlying problem into account. Stress-equation models solve many of the shortcomings of the other models, but are not applicable to all turbulent flows.

The right model to choose for modeling the Reynolds-stress tensor is depending entirely on the problem one needs to solve and the dominating physical properties.

4.3 SIMPLIFICATIONS

The goal of this chapter is to develop a scalable iterative solver for the RANS equations, with a particular focus on Restricted additive Schwarz (RAS) combined with ILU(0). Recently, additive Schwarz methods have received renewed interest (see section 4.8 for an overview and more details), with RAS/ILU showing a lot of promise for solving equations such as these [62, 63]. For our initial investigation we consider a simplified setup by employing a direct numerical simulation (DNS). This means that we are solving the Navier-Stokes equations without the use of any turbulence model, thus we effectively set the Reynolds-stress tensor to zero. This simplification allows us to work on our solver and preconditioner without the added difficulty introduced by using a turbulence model.

4.4 MOTIVATING APPLICATION

The model problem for this chapter consists of wind blowing across a field hitting a wind turbine located in the middle of that domain. The wind turbine exerts an opposing force causing turbulence and near-singularities in the underlying equations to arise. Figure 4.1 presents a schematic of this setup. A cross-section of a sample solution to this model problem

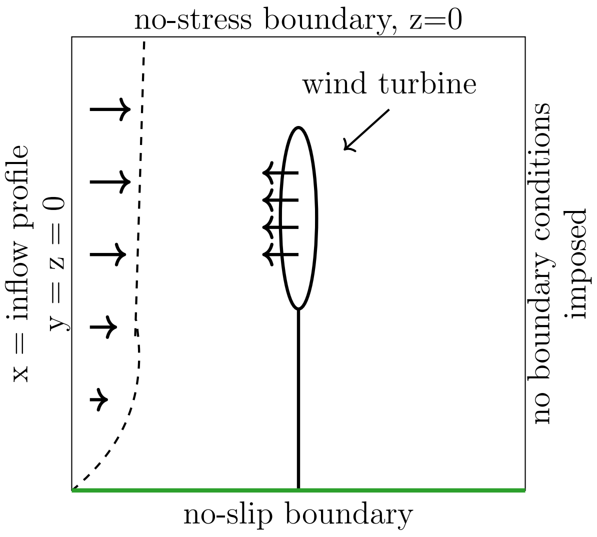


Figure 4.1: Schematic of motivating application

for a Reynolds number of $\mathscr{R} = 500$ is shown in fig. 4.2.

4.4.1 Turbine Force

The turbine force is composed of a few different parts: the location of the turbine (in the x/y/z planes), the diameter of the rotator, and the yaw of the turbines [64]. These are then composed in order to obtain the different required quantities, starting with the thickness of the turbine,

$$T = \exp\left(-\left(\frac{x_{rot}}{W}\right)^6\right) \tag{4.40}$$

where x_{rot} is the rotated x location of the turbine (based on the yaw of the turbines) and W is the thickness of the rotator plane. Next, the quantity representing the disk of the turbine is defined as

$$D = \exp\left(-\left(\frac{r}{R}\right)^6\right) \tag{4.41}$$

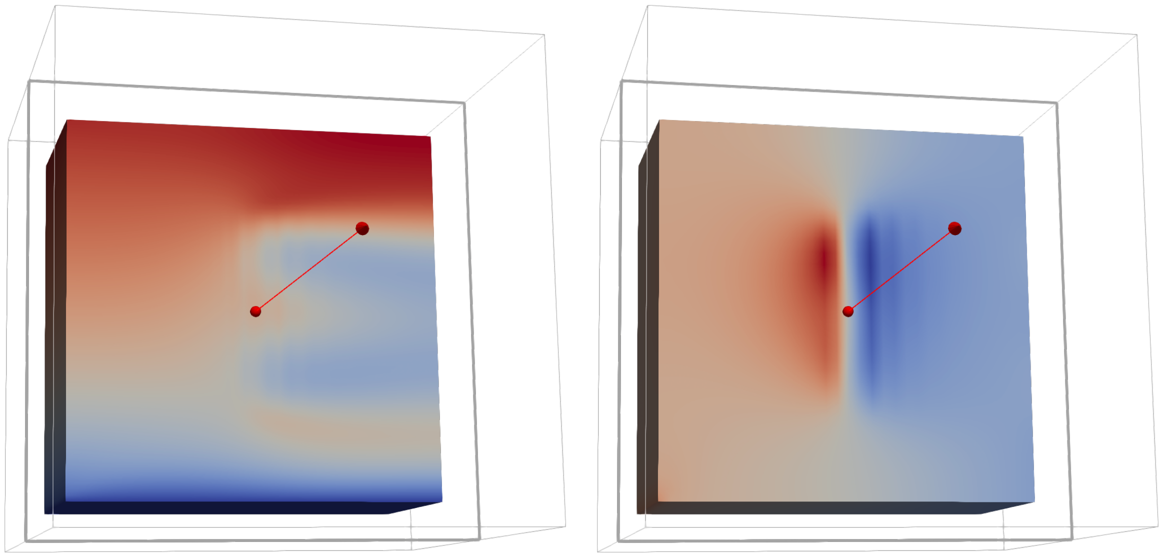

(a) (b)

Figure 4.2: (a) Cross-section of velocity component of sample solution, (b) cross-section of pressure component

where $r = \sqrt{y_{rot}^2 + z_{rot}^2}$, with y_{rot} and z_{rot} denoting the rotated y and z location, and R is the rotor radius. The force F is defined by

$$F = -0.5\pi R^2 \left(\frac{4ma}{1-ma}\right)\frac{r\sin(\pi r) + 0.5}{S_{norm}} \tag{4.42}$$

with $ma = 0.3$ being the modified thrust coefficient, and $S_{norm} = \frac{2+\pi}{2\pi}$. Then we compute the disk averaged velocity in the yawed case by

$$d_a = FTD\frac{y_{vec}}{vol_{norm}} \tag{4.43}$$

where $y_{vec} = \begin{bmatrix}\cos(\theta)\\ \sin(\theta)\\ 0\end{bmatrix}$ with θ being the yaw of the turbines, and $vol_{norm} = (2\Gamma(\frac{7}{6}))(\pi\Gamma(\frac{7}{6}))WR^2$ with $\Gamma(z) = \int_0^\infty t^{z-1}e^{-t}dt$. We now expand the dot product

$$t_{f_1} = d_a\cos(\theta)^2 \tag{4.44}$$

$$t_{f_2} = d_a\sin(\theta)^2 \tag{4.45}$$

$$t_{f_3} = 2d_a\cos(\theta)\sin(\theta) \tag{4.46}$$

and compose the full turbine force t_f by combining them with the two components of the velocity function u_0 and u_1 (i.e., along the horizontal plane parallel to the plane corresponding to the ground),

$$t_f = t_{f_1}u_0^2 + t_{f_2}u_1^2 + t_{f_3}u_0u_1 \tag{4.47}$$

In order to include this term in the final expression of the RANS equation it is subtracted from eq. (4.8).

The resulting turbine force is localized in a relatively small area in the middle of the domain. Figure 4.3 shows a visualization of the resulting force.

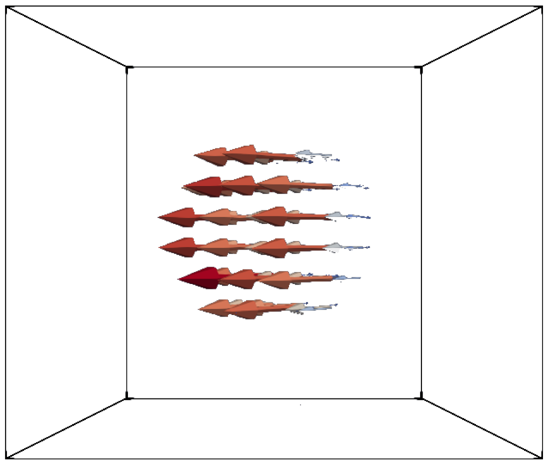


Figure 4.3: Visualization of turbine force.

4.5 DISCRETIZATION AND LINEARIZATION

The finite elements must satisfy the inf-sup condition (also known as Ladyženskaja-Babuška-Brezzi (LBB) condition) in order to guarantee stability, due to the absence of pressure in the continuity equation. We choose the Q2-Q1 Taylor-Hood elements that satisfy this condition. We express the system of linear equations in matrix-vector form by

$$\begin{cases} L\mathbf{u} + N(\mathbf{u}) + B^T p = \mathbf{f} \\ B\mathbf{u} = g \end{cases} \tag{4.48}$$

where $L\mathbf{u}$ is the discretization of the viscous term, $N(\mathbf{u})$ is the discretization of the nonlinear convective term, $B\mathbf{u}$ is the discretization of the negative divergence of $\mathbf{u}$, and $B^T p$ is the discretization of the gradient of p. All contributions to the source term, the boundary integral and contributions by the boundary conditions are contained in the right-hand side vectors $\mathbf{f}$ and g.

In order to apply a solver to this system, we need to linearize its nonlinearity. Two common choices to do so are Picard and Newton linearization. With Picard linearization u_i at the current iteration k is approximated by taking it from the previous iteration, u_i^{k-1}. This simple step turns the nonlinear PDE into a linear PDE. Picard linearization works well for nonlinear differential equations where the nonlinearity is due to convection-type terms. For more general nonlinearities, Newton linearization might be required. Newton linearization starts out in a similar way as Picard linearization by replacing u_i^k with the approximation $u_i^{k-1} + \delta u_i^k$. This leads to a more complicated expression that is typically simplified, with the assumption that δu_i^k is small and thus any power of δu_i^k greater than 1 is discarded. This results in a linearized system of equations. If the assumption of δu_i^k being small does not hold, then Newton linearization may diverge.

Once linearization has been done, the system of equations described in eq. (4.48) is described as in eq. (4.49),

$$A = \begin{bmatrix} F & B^T \\ B & 0 \end{bmatrix} \begin{bmatrix} \mathbf{u} \\ p \end{bmatrix} = \begin{bmatrix} \mathbf{f} \\ g \end{bmatrix}, \tag{4.49}$$

where F contains the action of L combined with the linearization of $N(\mathbf{u})$. The system of equations in eq. (4.49) is a linear saddle point problem with a zero block on the main diagonal (corresponding to the continuity equation).

4.6 EXISTING SOLVERS AND PRECONDITIONERS

Over the last few decades several different solvers and particularly preconditioners for the Navier-Stokes and, by extension, RANS equations have been developed.

Wasserman et al. [65] propose an extension to the multigrid method using the unconditionally positive-convergent (UPC) implicit time integration scheme. They successfully use this method to solve the RANS equations for two-equation turbulence models. They show its robustness and convergence improvements, in particular relative to an equivalent single-grid method based on the UPC scheme.

Another recent paper involving multigrid schemes was published by Baars et al. [66] They proposed a novel multigrid preconditioning method designed in particular for the 3D Navier-Stokes equations by using a skew partitioning scheme with rotated parallelepiped shaped overlapping subdomains that do not have any faces aligned with the actual grid. They show good scaling behavior of their new preconditioner and successfully used it to precondition

GMRES for solving the lid-driven cavity problem at high Reynolds numbers.

Farrell et al. [67] propose an augmented Lagrangian preconditioner for the 3D stationary Navier-Stokes equations that is particularly suited for high Reynolds numbers. They successfully solved the Navier-Stokes equations with Reynolds numbers up to 5000, however, their approach has some limitations. In particular, the discretization used does not exactly represent the divergence-free constraint and the multigrid algorithm used is tightly coupled to the use of piecewise constant elements for the pressure.

Discretizations other than standard finite elements have been proposed for solving the (RA)NS equations. In their recent book chapter on numerical methods in turbulence simulation, Fischer and Tomboulides [68] present the spectral element methods for turbulence. The spectral element method is a natural extension of spectral methods that retains the rapid convergence of spectral methods while accommodating more complex domains. They show the successful use of this discretization with the SEMFEM preconditioner (a hybrid preconditioner based on both spectral and finite elements) and various multigrid-based preconditioners.

Another possible discretization is a finite volume discretization. Klaij and Vuik [69] compared different SIMPLE and SIMPLE-type preconditioners (see section 4.6.1) using the finite volume discretization for cell-centered, collocated variables on unstructured grids. They find that in a real-world application, some variants of SIMPLE do not work as well (particularly, MSIMPLER) but others hold up quite well and appeared rather robust regarding the choice of parameters.

Most methods can be categorized in one of two categories: Methods that rely on a careful reordering of the degrees of freedom paired with some classical iterative method [70–72], and methods that are based on segregation. For the latter category, the system is split into its velocity and pressure component and then factorized into blocks to allow the use of iterative schemes [73–77].

Block preconditioners have two clear advantages over other approaches [78]:

1. Since the velocity and pressure components are split allows an easier application of existing preconditioning strategies to these individual blocks.

2. Any coupling between the physics is located in the Schur complement operator. This reduces the challenge in block preconditioning to finding an effective and appropriate approximation of the action of the inverse Schur complement.

In the following we give an overview of existing preconditioning schemes, both block preconditioners and some based on reordering of the degrees of freedom. Following this overview we will focus specifically on block-preconditioning strategies for designing our new preconditioner.

4.6.1 LDU Decomposition and Error Analysis

The block preconditioners for Krylov solvers are based on the LDU decomposition of the discretized system matrix,

$$A = \begin{bmatrix} F & B^T \\ B & 0 \end{bmatrix} = L_b D_b U_b = \begin{bmatrix} I & 0 \\ BF^{-1} & I \end{bmatrix} \begin{bmatrix} F & 0 \\ 0 & S \end{bmatrix} \begin{bmatrix} I & F^{-1}B^T \\ 0 & I \end{bmatrix} \tag{4.50}$$

with the Schur complement $S = -BF^{-1}B^T$. If we denote F^{-1} in the lower triangular block by the G_1, F^{-1} in the upper triangular block by G_2, and $\hat{S}$ as an approximation to S, we obtain the following decomposition,

$$\hat{A} = \begin{bmatrix} I & 0 \\ BG_1 & I \end{bmatrix} \begin{bmatrix} F & 0 \\ 0 & \hat{S} \end{bmatrix} \begin{bmatrix} I & G_2B^T \\ 0 & I \end{bmatrix} = \begin{bmatrix} F & FG_2B^T \\ BG_1F & BG_1FG_2B^T + \hat{S} \end{bmatrix} \tag{4.51}$$

Thus, we can denote the error matrix $E = A - \hat{A}$ as

$$E = \begin{bmatrix} 0 & B^T - FG_2B^T \\ B - BG_1F & -BG_1FG_2B^T - \hat{S} \end{bmatrix} \tag{4.52}$$

From the error matrix we are able to see how approximations to G_1, G_2, and $\hat{S}$ affect the momentum and/or continuity part of the original problem. For instance if $G_2 = F^{-1}$ then the resulting scheme is momentum preserving as the upper right block of the error matrix is the zero matrix. At the same time he quality of approximation for $\hat{S}$ only affects the continuity equation.

Most preconditioners based on this factorization either group the L and D or the D and U factors together, resulting in eq. (4.53) and eq. (4.54).

$$\hat{A}_{LD} = \begin{bmatrix} F & 0 \\ BG_1F & \hat{S} \end{bmatrix} \begin{bmatrix} I & G_2B^T \\ 0 & I \end{bmatrix} \tag{4.53}$$

$$\hat{A}_{DU} = \begin{bmatrix} I & 0 \\ BG_1 & I \end{bmatrix} \begin{bmatrix} F & FG_2B^T \\ 0 & \hat{S} \end{bmatrix} \tag{4.54}$$

A common practice of preconditioners is to drop the ungrouped part of the LDU factorization and use only one of the resulting block-triangular factors. The resulting error matrices for these two cases are given in eq. (4.55) and eq. (4.56).

$$E_{LD} = \begin{bmatrix} 0 & B^T \\ B(I - G_1F) & -\hat{S} \end{bmatrix} \tag{4.55}$$

$$E_{DU} = \begin{bmatrix} 0 & B^T(I - FG_2) \\ B & -\hat{S} \end{bmatrix} \tag{4.56}$$

Depending on which grouping is used, the resulting scheme incurs a fixed (though potentially very small) error either in the momentum or continuity equation. The rest of the error depends entirely either on the quality of the approximation of G_1 or G_2 respectively (but not both) and the quality of the approximation of $\hat{S}$. This approach is a careful balancing act between reducing the complexity (and potential sources of errors) in the scheme itself and the error incurred by being based on an approximate system to begin with. There are also some mathematical reasons why such an approach might be favorable as they still produce a correct solution. For instance, for SIMPLE, the resulting pressure field obtained is such that the velocity field still satisfies the continuity equation. For more details on this see [79, chapters 6.5 to 6.7].

4.6.2 Pressure-Convection-Diffusion (PCD)

The PCD preconditioner [73] is based on the D_bU_b factors and approximated the Schur complement S by $S \approx \hat{S} = -A_pF_p^{-1}Q_p$, with the full algorithm described in algorithm 4.1. Solving our mode problem as described in fig. 4.1 using the PCD preconditioner in Firedrake

Algorithm 4.1: PCD preconditioner

```
1 Compute $[r_u; r_p] = P_t[u; p]$
2 Solve $Sp = r_p$
3 Update $r_u = r_u - B^T p$
4 Solve $Fu = r_u$
```

for varying numbers of Reynolds numbers results in a breakdown of the solver for very small Reynolds number, as shown in fig. 4.4. For the results in fig. 4.4 we use LU as exact solver for the various solves necessary. The PCD preconditioner is known to exhibit good convergence behavior for enclosed flows [27]. However, it requires the assembly of a new matrix and thus is both inefficient and implementationally inconvenient. Applying the PCD preconditioner to our model problem shows that it is rather sensitive to the Reynolds number and fails to converge for rather small Reynolds numbers.

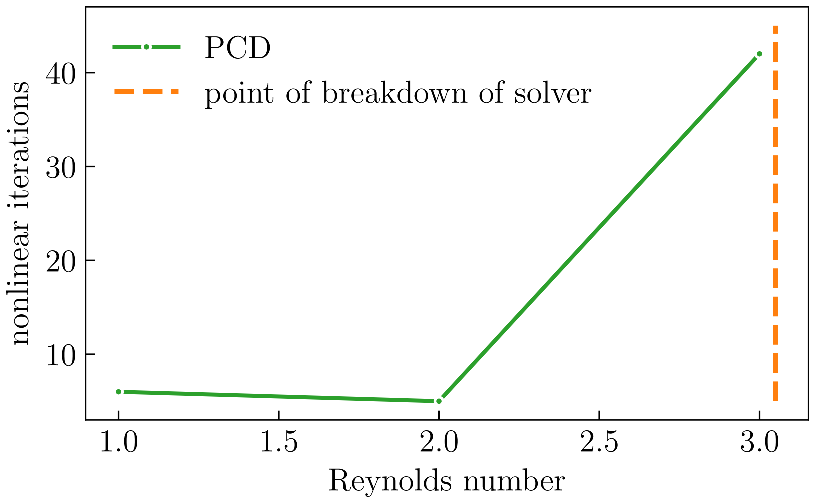


Figure 4.4: GMRES with PCD in 3D, implemented in Firedrake.

4.6.3 Least-Squares Commutator

The least squares commutator (LSC) preconditioner [27, 74] is based in the same principle as the PCD preconditioner. It approximates the discrete convection-diffusion operator, F_p, such that the commutator of the convection-diffusion operator,

$$\epsilon = L\nabla - \nabla L_p \tag{4.57}$$

where L_p is the convection-diffusion operator in the pressure space, becomes small by solving the least-squares problem

$$\min || \left[Q_v^{-1} F Q_v^{-1} B^T\right]_j - Q_v^{-1} B^T Q_p^{-1} \left[F_p\right]_j ||_{Q_v} \tag{4.58}$$

for column j of matrix F_p with $||x||_{Q_v} = \sqrt{x^T Q_v x}$. This allows the approximation of the Schur complement as

$$BF^{-1}B^T \approx (BQ_v^{-1}B^T)(BQ_v^{-1}FQ_v^{-1}B^T)^{-1}(BQ_v^{-1}B^T). \tag{4.59}$$

The dense inverse Q_v^{-1} is approximated by its diagonal, $D_v = diag(Q_v)$. The full algorithm is described in algorithm 4.2. The LSC preconditioner was proposed in response to the drawbacks of PCD, in particular the need of PCD to construct a new matrix, by only using the existing blocks of the system matrix. Elman et al. [27] find that the convergence behavior of LSC is not dependent on the grid resolution but does depend on the Reynolds number. They show that for solving a two-dimensional flow over a step increasing the Reynolds

Algorithm 4.2: LSC preconditioner

```
1 Solve $S_f z_2 = r_2$, where $S_f = BD_v^{-1}B^T$
2 Update $r_2 = BD_v^{-1}FD_v^{-1}B^T z_2$
3 Solve $S_f z_2 = -r_2$
4 Update $r_1 = r_1 - B^T z_2$
5 Solve $Fz_1 = r_1$
```

number from 10 to 200 increases the number of nonlinear Newton iteration by a factor of about 2 and the number of linear iterations by a factor of about 3. For the lid-driven cavity in two dimensions they observed a much more dramatic increase in the number of iterations, about 4× for the number of nonlinear iterations and about 20× for the number of linear iterations.

4.6.4 Augmented Lagrangian (AL)

The idea of the Augmented Lagrangian preconditioner [75] is to introduce an additional term in the equations that modifies the Schur complement but does not change the continuous solution. This approach is also referred to as grad-div stabilization, and leads to the following variation of eq. (4.49) ,

$$\begin{bmatrix} F + \gamma B^T M_p^{-1} B & B^T \\ B & 0 \end{bmatrix} \begin{pmatrix} \delta u \\ \delta p \end{pmatrix} = \begin{pmatrix} b + \gamma B^T M_p^{-1} c \\ c \end{pmatrix} \tag{4.60}$$

with M_p being the pressure mass matrix. For γ not too small, the Schur complement inverse is approximated by

$$S^{-1} \approx -(\nu + \gamma) M_p^{-1}. \tag{4.61}$$

This approach requires specialized multigrid algorithm, and are described as difficult to implement [80]. Practical approaches apply certain simplifications that allow for algebraic multigrid techniques to be applied, however, the convergence of the resulting scheme deteriorates as the Reynolds number increases [81]. Recently He et al. [82] propose a variation of the AL preconditioner that uses the Schur complement approximation from the SIMPLE preconditioner as part of the approximation of the inverse of the Schur complement matrix for the AL preconditioner. They were able to solve the RANS equations with large Reynolds number much more efficiently than the basic SIMPLE algorithm described below.

4.6.5 SIMPLE and Its Variations

The original SIMPLE preconditioner [76, 77] is based on the $L_b D_b$ factors with the Schur complement S approximated by $S \approx \hat{S} = -BD^{-1}B^T$ where $D = diag(F)$, with the full algorithm shown in algorithm 4.3. with p^* estimated from the prior iteration. There have

Algorithm 4.3: SIMPLE preconditioner

```
1 Solve $Fu^* = r_u - B^T p^*$
2 Solve $S\delta p = r_p - Bu^*$
3 Update $u = u^* - D^{-1}B^T\delta p$
4 Update $p = p^* + \delta p$
```

been many variations of SIMPLE developed over the years. SIMPLEC replaced the diagonal approximation of F with a diagonal matrix where each entry is the absolute row sum of the corresponding row in F, leading to a better approximation to the matrix F. SIMPLER is a variation that adds an additional step to the algorithm to provide a better value for p^* at the start of each iteration by solving the equation

$$Sp^* = r_p - BD^{-1}((D - F)u^k + r_u) \tag{4.62}$$

with u^k obtained from the prior iteration. These variations themselves have different sub-variations themselves, all aiming to make the error in the algorithm smaller or to reduce the dependence on parameters like the Reynolds number.

4.6.6 Reordering of Degrees of Freedom

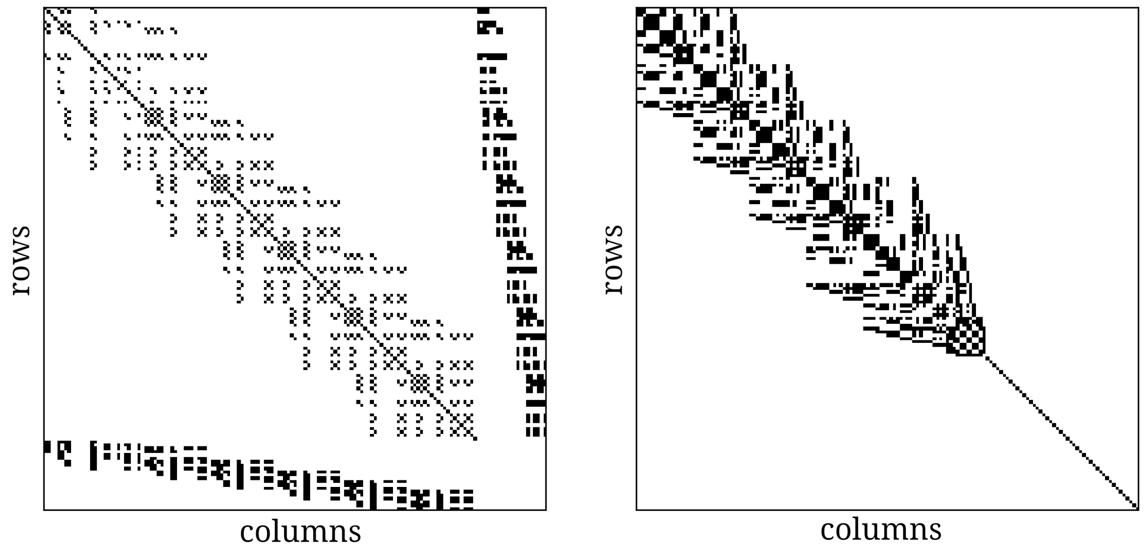
Applying standard finite element code to block preconditioners as the ones presented in section 4.6.1 has one major disadvantage: it requires an adaptation of the setup in order to split velocity and pressure degrees of freedom. An alternative approach that does not require such a splitting is the application of ILU with a specific reordering of the degrees of freedom.

A direct application of LU or ILU is not possible, as the continuity equations contain a zero pressure block resulting in a zero pivot. Applying a pivoting strategy, however, increases the memory requirements significantly [83]. There has been work done by Wille et al. [84–86] that shows that certain specific orderings of the degrees of freedom result in pivoting to not be necessary. In particular, a typical sorting of the unknowns so that the velocity degrees of freedom come first and the pressure degrees of freedom come second results in a so-called *p-last* ordering. The resulting system matrix not only has the troublesome zero block but also has a large profile.

A different approach is to combine a node renumbering scheme with a specific reordering of the degrees of freedom [70], in particular, the Cuthill-McKee (and related Reverse Cuthill-McKee (RCM)) or Sloan [87] ordering (for meshes with only nodes). Instead of a p-last ordering as proposed by Wille, these reorderings are based on the concept of levels.

For the Cuthill-McKee ordering, a level contains all degrees of freedom that are directly connected to the previous level and are not yet contained in any level themselves. For the Sloan ordering, a level is constructed by taking the node with the highest node number directly connected to the nodes of the previous levels. Then, the new level contains this particular node and all nodes with a smaller node number not yet contained in any previous level.

Once the levels have been established, the degrees of freedom are ordered by first taking the velocity degrees of freedom of level 1 followed by the pressure degrees of freedom of level 1. Then the same is done for level 2, etc. This ordering is also called *p-last per level*. The result is a system matrix with a similar profile but smaller bandwidth as obtained by a *p-last* type ordering while also avoiding zero pivots. fig. 4.5 shows the sparsity pattern of a system matrix for Navier-Stokes using both a p-last ordering and Reverse Cuthill-McKee. One thing we note right away is that ordering the degrees of freedom according to Reverse



(a) p-last ordering (b) Reverse Cuthill-McKee ordering

Figure 4.5: Sparsity pattern of Navier-Stokes example system matrix in two dimensions

Cuthill-McKee results in a system matrix with a dense diagonal, there is no more a zero block located in the bottom right corner. Additionally, the bandwidth of the matrix is greatly reduced (roughly by a factor 4 from 133 to 32 in the example shown in fig. 4.5), allowing for better memory access patterns. Both of these plots were obtained through Firedrake.

Using such an ordering allows the use of ILU to solve the system [70]. Our experiments show that it is possible to solve systems with Reynolds number as big as 8, as shown in fig. 4.6. Once the Reynolds number goes past 8, this solver stalls out and fails to converge

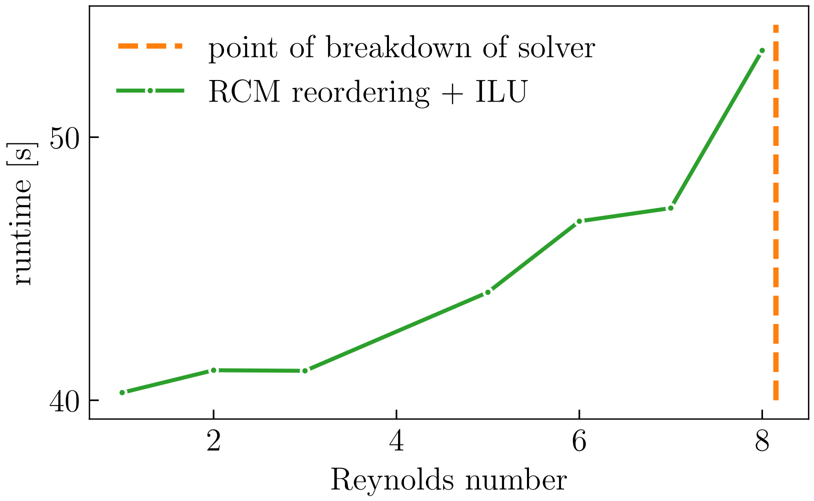


Figure 4.6: Time-to-convergence of RCM ordering + ILU with increasing Reynolds number

further for our model problem. This suggests that although ILU is indeed capable of solving the Navier-Stokes equations using an ordering of the degrees of freedom according to Cuthill-McKee, it is not sufficient for the added challenges introduced by placing a wind turbine in the domain.

An extension of ILU was developed by Chen et al. [71], proposing a preconditioner based on *HILUSCI* [72] (*Hierarchical Incomplete LU-Crout with Scalability-oriented and Inverse-based dropping*) called *HILUNG*. They were successfully able to apply this method to the 2d driven cavity problem and 3D laminar flow over cylinder problem for large Reynolds numbers of up to 5000. However, HILUSCI is currently only available as serial algorithm limiting the scalability of this solver.

4.7 RECENT WORK

In recent years there has been some renewed work done to develop better preconditioners for the Navier-Stokes equations and, by extension, the Reynolds-averaged Navier-Stokes equations. Voronin et al. [31] propose using monolithic geometric multigrid based on the $\mathbb{Q}_1\text{iso}\mathbb{Q}_2/\mathbb{Q}_1$ discretization to develop new preconditioners. Both Vanka and Braess-Sarazin smoothing are leveraged, and provide an extensive local Fourier analysis for predicting the algorithmic performance. On conclusion in [31] is that monolithic multigrid using both Braess-Sarazin and Vanka are indeed effective preconditioners; Braess-Sarazin performs better than Vanka, although this is highly dependent on the underlying implementation.

Monolithic multigrid is known to demonstrate robust convergence for the Stokes equations [30]. So far the most successful multigrid approaches for saddle-point systems, like the ones considered in this chapter, are based on geometric multigrid, as shown by Adler et al. [88]. In subsequent works, Farrell et al. [89] present an extensive Fourier analysis of additive Vanka as part of monolithic multigrid including an implementation, PCPATCH, of the algorithm in PETSc [90].

The setup, as depicted in fig. 4.1, is a problem that is currently solved as part of the WindSE software developed at the National Renewable Energy Laboratory (NREL). There is a need to find an efficient and reliable solution to the RANS equations in the context of the FEnICs/Firedrake software framework.

Many of the approaches described in section 4.6 are generally capable of solving the Navier-Stokes and also the RANS equations, some even for relatively large Reynolds numbers. However, each method therein struggles to varying degrees with our model problem. This is in large part due to the sudden changes in velocity and pressure that occur around the turbine. These sudden changes and the resulting turbulences contribute to the difficulty in the problem.

4.8 RESTRICTED ADDITIVE SCHWARZ (RAS) + INCOMPLETE LU (ILU)

At the end of the 20th century, Cai and Sarkis [91] present restricted additive Schwarz (RAS) as a variation to the classical additive Schwarz algorithms that reduces communication and leads to faster convergence both in terms of iteration count and in CPU time. In 2006, Zhongze and Saad [92] propose a preconditioner, called SchurRAS, that is based on applying a RAS algorithm to the LDU decomposition of the original system matrix, noting that the only difficult operation is the Schur complement solve as the other operations of this factorization are akin to restriction and prolongation operators. This preconditioner is found to be up to twice as fast as simple RAS for a 2D model problem. Alcin et al. [62] investigate the efficiency and scalability of a two-level Schwarz algorithm for compressible and incompressible flows, showing that the use of RAS often results in good scalability and performance. Liu et al. develop an efficient implementation in [93] of a RAS preconditioner for sparse linear systems on the GPU showing that it is capable of reaching speedups of up to 10 times compared to the CPU. Similarly, Yang et al. [94] present a study on using

GMRES on the GPU preconditioned by a parallel ILU(0) and ILUT algorithms based on RAS, finding that they are able to achieve a speedup of up to 8 times relative to the CPU, with ILU(0) offering better speedup, but ILUT offering better convergence for their model problems.

Saberi et al. propose a restricted additive Vanka relaxation scheme in [95] that is used as part of a geometric Multigrid cycle. This relaxation is based on RAS with the prolongation operator replaced by a modified version that restricts prolongation to degrees of freedom in the current set. When using restricted additive Vanka they are able to achieve reduction rates comparable to multiplicative Vanka while being an additive algorithm. Ram et al. [96] propose a hybrid parallel ILU preconditioner that is capable of taking full advantage of shared memory parallelism in the ILU factorization step by using a multilevel nested dissection reordering of the degrees of freedom. RAS preconditioned GMRES with their algorithms achieve a parallel efficiency of more than 80%. Riva et al. [63] perform a local Fourier analysis of additive Schwarz relaxation schemes in the context of multigrid methods. This analysis reveals that RAS achieves comparable convergence rates to Additive Schwarz (AS), having various favorable properties in terms of scalability and communication cost. They conclude that RAS is generally a good alternative to consider.

4.8.1 Restricted Additive Schwarz (RAS)

We denote the adjacency graph of A to be the graph $G = (V, E)$, where V is the set of all vertices representing the binary relation that a_{ij} is nonzero between vertices i and j. Now assume that the domain Ω is decomposed into n subdomains Ω_i^δ where δ denotes the extent to which the subdomains overlap. This decomposition is obtained from a partitioning of V into p subgraphs V_i^δ. The superscript δ indicates the overlap between the different partitions, and the subscript i is the index of the respective subdomain. The overlap δ is defined as the number of immediate neighbors of V_i^0 (i.e., without overlap) that are included in V_i^δ. This allows us to define the restriction operator

$$R_{i,\delta} : \Omega \to \Omega_i^\delta \text{ with } R_{i,\delta}(x) = x_{i,\delta} \tag{4.63}$$

where $x_{i,\delta}$ contains only components of z that belong to Ω_i^δ. The prolongation operator is correspondingly defined as,

$$P_{i,\delta} : \Omega_i^\delta \to \Omega \text{ with } P_{i,\delta}(x_{i,\delta}) = x \tag{4.64}$$

where $x = P_{i,\delta}(x_{i,\delta})$ is the zero-extension of a vector $x_{i,\delta}$ from Ω_i^δ to Ω. The thus defined restriction and prolongation operators allow us to express the local matrix $A_{i,\delta}$ corresponding to the subdomain Ω_i^δ as

$$A_{i,\delta} = R_{i,\delta} A P_{i,\delta} \tag{4.65}$$

With this notation we then denote the additive Schwarz preconditioner as

$$M = \sum_{i=1}^{n} P_{i,\delta} A_{i,\delta}^{-1} R_{i,\delta} \tag{4.66}$$

where each subdomain is solved independently. The special case of the block Jacobi preconditioner is obtained by setting the overlap to 0,

$$M = \sum_{i=1}^{n} P_{i,0} A_{i,0}^{-1} R_{i,0}. \tag{4.67}$$

A combination of the additive Schwarz and block-Jacobi preconditioning schemes results in the restricted additive Schwarz preconditioner (RAS),

$$M = \sum_{i=1}^{n} P_{i,0} A_{i,\delta}^{-1} R_{i,0} \tag{4.68}$$

$$= \sum_{i=1}^{n} P_{i,0} (R_{i,\delta} A P_{i,\delta})^{-1} R_{i,0} \tag{4.69}$$

where the subdomains overlap as measured by δ. The solution found on $\Omega_{i,\delta}$ is then restricted to the non-overlapping part of the subdomain, $\Omega_{i,0}$, when prolonged to the global domain Ω.

This leaves us with the task of solving the inverse of the local matrix $A_{i,\delta}$. This is often performed approximately, especially for larger δ. A common way to solve this inverse problem is by means of an incomplete LU factorization.

4.8.2 ILU(0)

ILU(0) takes the sparse matrix A and replaces all existing non-zero entries in A with new values, such that the lower triangular matrix L is stored in the lower triangular part of A (with the diagonal implicitly assumed to be a unit diagonal) and the upper triangular matrix U is stored in the upper triangular part of A (including the diagonal). A pointwise factorization of ILU(0) is given in algorithm 4.4

Algorithm 4.4: Point-wise ILU(0) factorization

```
for $i \leftarrow 2$ to $n$ do
    for $k \leftarrow 1$ to $i-1$ do
        if $(i,k) \notin K$ then
            continue
        end
        $a_{ik} = a_{ik}/a_{kk}$
        for $j \leftarrow k+1$ to $n$ do
            if $(i,j) \notin K$ then
                continue
            end
            $a_{ij} = a_{ij} - a_{ik}a_{kj}$
        end
    end
end
```

4.8.3 ILU(k)

ILU(0) has the advantage that the resulting factorization has the same sparsity pattern as the original matrix and is stored in the same memory. However, this comes at the cost of accuracy. ILU(k) is similar to ILU(0) except that it allows for fill-in to occur. The integer k defines the amount of fill-in to occur, with a larger k resulting in more fill-in. The initial level l_{ij} of each entry of A is defined by

$$l_{ij} = \begin{cases} 0, & (i,j) \in K \\ \infty, & (i,j) \notin K \end{cases} \tag{4.70}$$

This means that the nonzero entries of A have an initial level of 0, and all other entries have a level of infinity. The levels are then updated by

$$l_{ij} = \min(l_{ij}, l_{ip} + l_{pj} + 1) \tag{4.71}$$

The level of a nonzero entry always remains unchanged, but certain zero entries are updated to a smaller positive level. If the new level is greater than the cutoff k, this entry is eliminated. Only entries whose levels remain below the cutoff k throughout the iteration serve as extension to the original sparsity pattern K, resulting in a more accurate but less sparse factorization. algorithm 4.5 gives an algorithmic overview of the ILU(k) factorization.

Algorithm 4.5: ILU(k) factorization

```
For all nonzero entries in the nonzero pattern $K$ define $l_{ij} = 0$
for $i \leftarrow 2...n$ do
    for $l \leftarrow 1...i-1$ do
        if $l_{il} > k$ then
            continue
        end
        $a_{il} = a_{il}/a_{ll}$
        for $j \leftarrow l+1...n$ do
            $a_{ij} = a_{ij} - a_{il}a_{lj}$
            $l_{ij} = \min(l_{ij}, l_{il} + l_{lj} + 1)$
        end
    end
    if $l_{ij} > k$ then
        $a_{ij} = 0$
    end
end
```

4.9 SOFTWARE

We make use of various existing solver libraries and frameworks:

hypre [97, 98] software library providing implementations of a wide range of high performance preconditioners and solvers for large and sparse linear systems of equations, with a particular emphasis on massively parallel computers.

PETSc [99–101] software library that includes a large suite of scalable solvers for linear and nonlinear equations and various other algorithms. It also is capable of interacting with other software libraries - like hypre - able to also take advantage of their capabilities.

Firedrake [102] overarching framework we use for implementing our solvers and preconditioners. Firedrake is "an automated system for the solution of partial differential equations using the finite element method (FEM)" [103]. Firedrake brings native support for a range of numerical solvers and provides a convenient interface for incorporating PETSc with all of its options and features.

4.10 PROPOSED SOLVER

In order to solve our model problem we revisit the LDU factorization presented in section 4.6.1 and design a new solver by combining Newton with GMRES, AMG, and RAS/ILU by creating a new PETSc interface. The preconditioning is based on the LDU factorization of the system matrix in eq. (4.49).

4.10.1 Revisiting LDU Decomposition

For this attempt we revisit the approach described in section 4.6.1, starting with the LDU block-decomposition of the system matrix,

$$A = \begin{bmatrix} F & B^T \\ B & 0 \end{bmatrix} = L_b D_b U_b = \begin{bmatrix} I & 0 \\ BF^{-1} & I \end{bmatrix} \begin{bmatrix} F & 0 \\ 0 & S \end{bmatrix} \begin{bmatrix} I & F^{-1}B^T \\ 0 & I \end{bmatrix} \tag{4.72}$$

with the Schur complement $S = -BF^{-1}B^T$. The clear advantage of this approach is the lack of need to invert the full system matrix. Instead, the only two inverses required for an exact solve are the inverse of the velocity block matrix, F, and the inverse of the Schur complement matrix, S, which is the source of error in this part of the algorithm. However, there is another source of potential error that needs to be considered, namely the error caused by the nonlinear iteration. As described in section 4.5, the two most common choices for linearizing a nonlinear system are Picard and Newton linearization. Due to the nonlinearity arising from our model problem, it is necessary to use the more general Newton linearization over the simpler Picard linearization. In addition, Newton linearization exhibits better convergence behavior [104].

4.10.2 Solver Design

Taking all of these considerations into account we design a solver algorithm that attempts to solve our model problem as described in algorithm 4.6. A visualization of the solver giving a high-level overview is given in fig. 4.7. Since we are working with a nonlinear system we use a Newton iteration with a linesearch to linearize the system at each iteration. Once linearized we apply a GMRES solver with its preconditioner based on the LDU factorization of the system matrix. Once the LDU factorization has been obtained, we use GMRES to solve for both the action of the velocity block and the action of the Schur-complement block. These two GMRES solve are both preconditioned with algebraic multigrid (AMG) that uses RAS/ILU as relaxation scheme on each level. An algorithmic description of the full algorithm is given in algorithm 4.6. Accessing the RAS/ILU algorithm in hypre from Firedrake through PETSc is currently not possible with the latest stable release of PETSc. To enable this a new interface in PETSc first had to be implemented [105] in order to expose

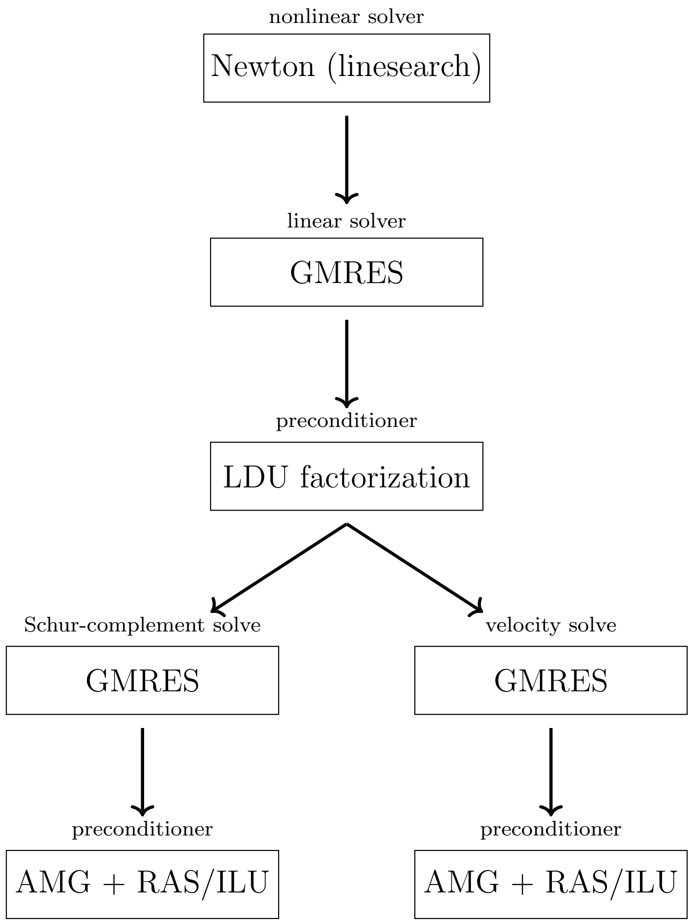


Figure 4.7: High-level overview of the solver.

the solver in such a way that we are able to interact with it through the normal solver interface in Firedrake.

Using this solver we are able to solve systems with Reynolds numbers up to around $\mathscr{R} = 500$. Figure 4.8 shows such a solve for a test run using 4 MPI ranks and a $30 \times 30 \times 30$ mesh resulting in about 710 000 degrees of freedom. We observe the solver to be consistent in reducing the error until it achieves a residual convergence of 10^{-6}. However, the solver requires a long time to complete with a total runtime of more than 12 hours. This is to be expected, however, as we only used 4 MPI ranks, thus dividing our full domain into 4 subdomains on which we run an ILU(0) solve. Each subsolve is done with around 150 000 degrees of freedom and thus is very slow. Increasing the MPI rank and thus reducing the subdomain size will improve the time required per ILU solve, however, it comes at a cost of increased iteration count as we reduce the accuracy of our solver by breaking it down into an increasing number of non-overlapping domains.

To asses the performance of our solver we compare it to a direct solver that uses Newton

Algorithm 4.6: High-level overview of (RA)NS solver

```
1  Set up system matrix $A$
2  Set up right hand side $b_u$ and $b_p$
3  Choose initial guess $u$ and $p$. Compute initial residual $res_{\text{init}} = \| [b_u; b_p]^T - A\,[u; p]^T \|$
4  Set current residual $res = res_{\text{init}}$
5  Set tolerance $tol$
6  while $\frac{res}{res_{init}} > tol$ do
7      Newton linearization to obtain $A_{lin}$
8      LDU factorization of $A_{lin}$
9      for $i = 1..15$ do
10         Precondition velocity system with RAS/ILU
11         GMRES iteration for velocity solve
12     end
13     for $i = 1..15$ do
14         Precondition Schur-complement with RAS/ILU
15         GMRES iteration for Schur-complement solve
16     end
17     Update current residual $res = \| [b_u; b_p]^T - A\,[u; p]^T \|$
18 end
```

for linearization and GMRES preconditioned by LU using the `mumps` package [106] for the linear solve. Since a full LU factorization is memory intensive and slow, we reduce the mesh size to a size of $25 \times 25 \times 25$ resulting in a little over 415 000 degrees of freedom, using a total of 50 MPI ranks on Delta. Running both our solver and a solver based on `mumps` to a convergence tolerance of 10^{-5} for a Reynolds number of $\mathscr{R} = 100$ results in the solver with `mumps` requiring a total of 96 iterations and 3495 seconds (about 1 hour). If we run the same problem with our solver we reach convergence in 6 iterations and 860 seconds (about 14 minutes). This suggests that the added complexity yields better convergence and lower runtime.

4.10.3 Profiling the Solver

For profiling the solver, we first profile PETSc to see which parts of the solver are responsible for how much of the overall runtime. A flamegraph visualization of that profile is shown in fig. 4.9. fig. 4.10 shows a breakdown of the largest contributors of the overall runtime of the solver As can be seen from fig. 4.10 close to 80% of the overall runtime is spent in the AMG+RAS/ILU preconditioner. In turn timing the various parts of the AMG cycle, we observe that about 80% of that time is spent in relaxation, with the remaining

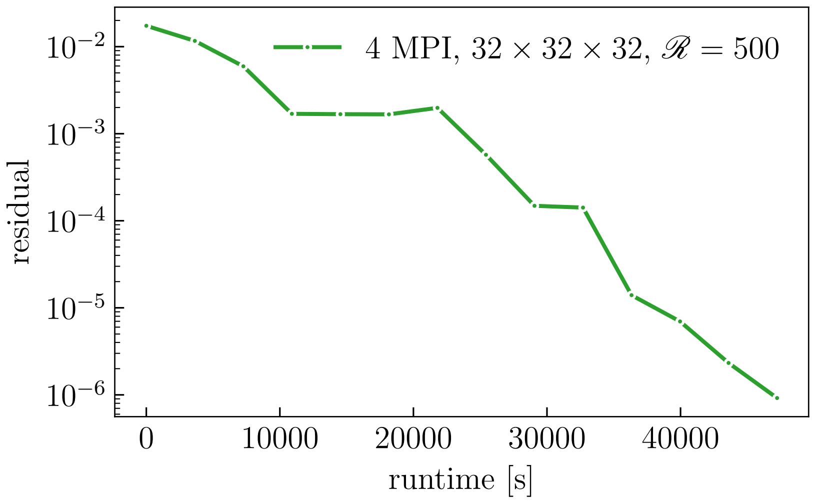


Figure 4.8: RANS solver applied to model problem with $\mathscr{R} = 500$.

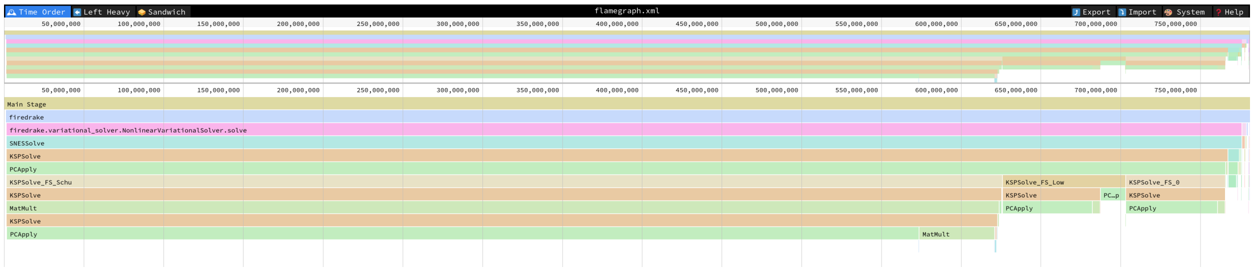


Figure 4.9: Flamegraph visualized with speedscope [107].

time mostly spent moving between the different meshes. Thus, a more detailed profile of the RAS/ILU algorithm is needed to obtain a detailed and complete understanding of where all this time is spent. Timing the various parts that make up the RAS/ILU algorithm, we obtain the timings as shown in table 4.1. The timings are taken for running one iteration of RAS/ILU for the finest level in the AMG algorithm. As observed in table 4.1 there are three parts that are responsible for the largest amount of time. The forward solve and backward substitution combined make up 58.5% of the overall runtime. These two parts of the algorithm are made up of a couple `for` loops facilitating this solve. They are already implemented in an optimal way given the underlying data structures. Most of the remaining runtime, 39%, is spent in computing the residual. This is an expensive operation as it involves communicating data between the different subdomains. A further breakdown of the various parts of this algorithm is shown in table 4.2. As can be seen in table 4.2 almost all of the time, 96.29%, is spent performing the computations, which is overlapped with any necessary communication. This, however, is rather unsurprising as these runs are done on a small test machine with only 4 MPI ranks. This results in the local RAS domains to have

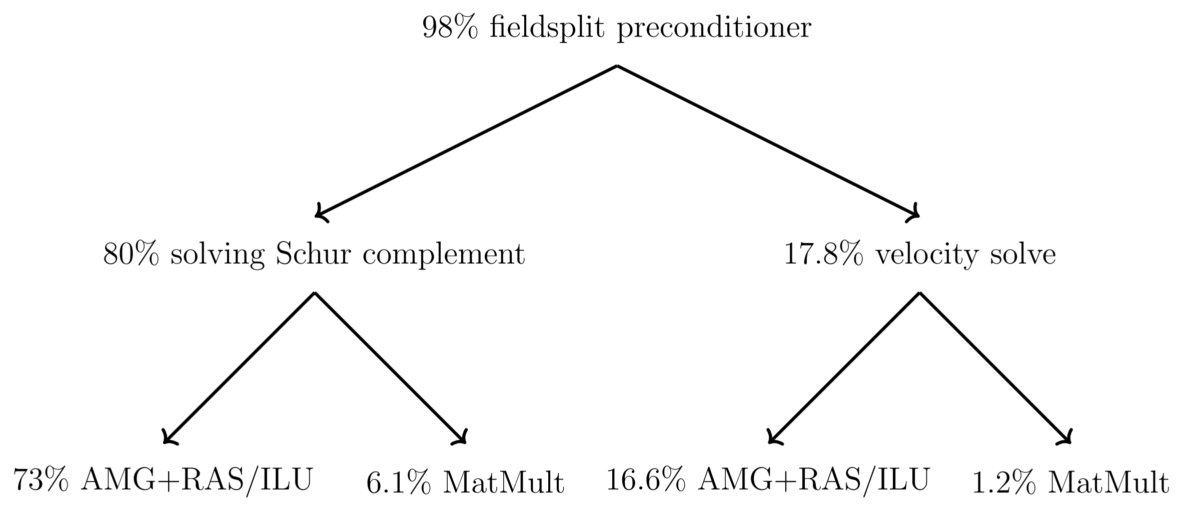


Figure 4.10: Breakdown of PETSc timings listing the largest contributor to the overall runtime, given in percentage points of the overall runtime.

	time [ms]	percentage
setup	0.002	0.1
computing residual	36.218	39.0
communicate	3.062	3.3
forward solve	19.706	21.2
backward substitution	34.625	37.3
clean up	0.205	0.2

Table 4.1: Breakdown of RAS/ILU algorithm by timing various parts, $30 \times 30 \times 30$ elements spread across 4 MPI ranks.

around 170 000 degrees of freedom, which are very large systems to be solved with ILU. The obvious solution is to increase the number of MPI ranks in order to reduce the local problem size. This, however, comes at the expense of convergence as AMG+RAS/ILU with an increasing number of Schwarz-domains has a reduced convergence rate for solving such a system as a single global ILU solve. Figure 4.11 shows how our solver scales with varying number of MPI ranks. The numerical values behind fig. 4.11 are found in table 4.3. As can be seen from both fig. 4.11 and table 4.3, increasing the number of MPI ranks does indeed decrease the required runtime. However, the runtime does not scale at the same rate as the number of MPI ranks is scales. Up to 30 MPI ranks the runtime scales noticeably worse than the number of MPI ranks, due to both an increase in required communication and worse convergence behavior when more RAS subdomains are used. Both of these, however, are to be expected. Once the number of MPI ranks becomes high enough, the runtime scaling matches much more closely to the scaling of the number of MPI ranks.

	time [ms]	percentage
setup	0.016144	0.04
pack send data	0.028518	0.08
start communication	0.006551	0.02
local matvec	34.872804	96.29
end communication	0.066910	0.18
remote matvec	1.224446	3.38

Table 4.2: Breakdown of computing residual as part of RAS/ILU algorithm.

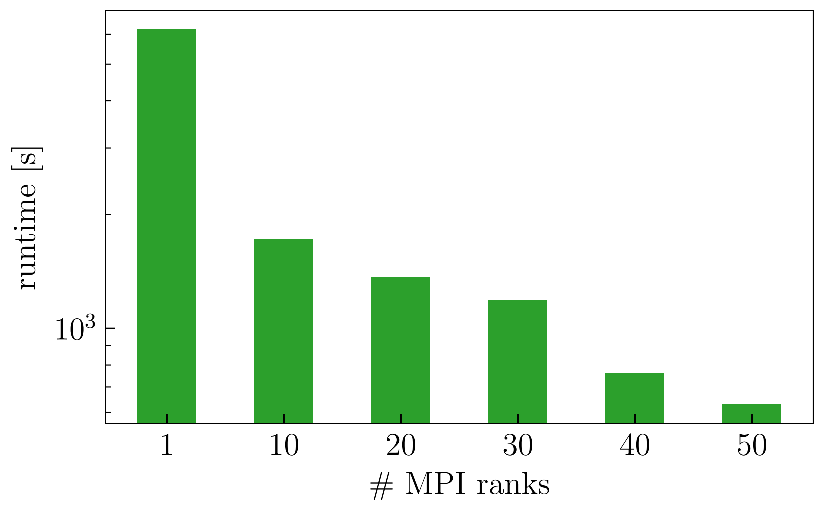


Figure 4.11: Scaling of RANS solve for $\mathscr{R} = 10$ for increasing number of MPI ranks.

Knowing that the majority of the runtime of the solver is spent in the preconditioning scheme, AMG with RAS/ILU, we use a cost analysis based on this algorithm to learn more about the behavior of our solver overall.

4.10.4 Performance Model

To better understand the behavior of the AMG+RAS/ILU preconditioner scheme and how it behaves with varying problem size and varying number of MPI ranks we establish a performance model of the algorithm. AMG consists of multiple steps as it consists of multiple levels. The total solve time, thus, is a combination of the time spent on the different levels in the cycle. Let l be the total number of levels in the algorithm, then we decompose the overall time by

$$T = \sum_{i=1}^{l-1} (T^i_{\text{solve}} + T^i_{\text{restrict}}) + T^l_{\text{solve}} + \sum_{i=1}^{l-1} (T^i_{\text{solve}} + T^i_{\text{interpolate}}), \tag{4.73}$$

# MPI ranks	scaling	runtime [s]	scaling
1	-	6186.288	-
10	10.00	1725.585	3.59
20	2.00	1367.729	1.26
30	1.50	990.315	1.38
40	1.33	759.262	1.30
50	1.25	628.377	1.21

Table 4.3: Numerical values for solving RANS problem with $\mathscr{R} = 10$ for increasing number of MPI ranks.

with T^i_{solve} the application of the relaxation scheme, T^i_{restrict} the restriction from level i to level $i-1$, and $T^i_{\text{interpolate}}$ the interpolation from level $i-1$ to i. We work with a symmetric setup with the same number of pre- and post-relaxation steps. With his knowledge we simplify this equation slightly,

$$T = \sum_{i=1}^{l-1} (2T^i_{\text{solve}} + T^i_{\text{restrict}} + T^i_{\text{interpolate}}) + T^l_{\text{solve}}. \tag{4.74}$$

In order to be able to set up the model as shown in eq. (4.74) we need to describe the three parts, T^i_{solve}, T^i_{restrict}, and $T^i_{\text{interpolate}}$. T^i_{solve} is an application of RAS/ILU at that level. Let N be the average number of degrees of freedom per Schwarz domain, N_Ω the average number of degrees of freedom sitting along the boundary between neighboring domains and thus need to be communicated, and we know that most rows of the system matrix contain 97 nonzeros, stored in a sparse matrix format, then we define the performance model for RAS/ILU to consist of:

1. Compute the current local residual: $res = \beta b + \alpha A x$.
    - flops: $97N + 2N$
    - reads: $101N$ doubles
    - writes: N doubles
2. Pack the send data.
    - flops: 0
    - reads: N_Ω doubles
    - writes: N_Ω doubles

3a. Local ILU solve: forward solve followed by backward substitution.

- flops: $97N/2$
- reads: $97N/2 + N$ doubles
- writes: N doubles

3b. Communicate the boundary data overlapped with local computation. The communication is modeled using the max-rate performance model [17].

$$T = t_c + n_r\alpha_l + \frac{kn_b}{\min(R_N, kR_C)} \tag{4.75}$$

with t_c the time for copying the data, n_r the number of messages a rank is sending, α_l the latency introduced by MPI per message, k the number of processes, n_b the number of bytes sent per process, R_N the injection bandwidth, and R_C the rate that can be achieved by each process in sending or receiving a message.

4. ILU solve with received boundary data: forward solve followed by backward substitution.

   - flops: $97N_\Omega/2$
   - reads: $97N_\Omega/2 + N_\Omega$ doubles
   - writes: N_Ω doubles

5. Update global solution: $u = \beta u_{\text{upd}}$

   - flops: N
   - reads: $2N$ doubles
   - writes: N doubles

The restriction operator, denoted in eq. (4.74) by T^i_{restrict}, consists of two parts:

1. Computing the current residual.

   - flops: $97N$
   - reads: $97N + 2N$ doubles
   - writes: N doubles

2. Restrict the problem to the next coarser level.

- flops: N
- reads: N doubles
- writes: N_c doubles, where N_c the number of degrees of freedom on the next coarser level

The interpolation operator, denoted in eq. (4.74) by $T^i_{\text{interpolate}}$, consists mostly out of the actual interpolation of the data from the coarser level and correcting the solution on the current level. This is summarized as:

- flops: N
- reads: N_c doubles, where N_c the number of degrees of freedom on the coarser level
- writes: N doubles

With all of these parts in place we only need some metrics for our test machine, Delta [108]. These values have been obtained using the BabelStream benchmark [109]:

- Peak flop rate: 573 GFLOP/s for CPU, 9472 GFLOP/s for GPU
- Peak read speed: 27.78 GB/s for CPU and 1361.50 GB/s for GPU
- Peak write speed: 25.96 GB/s for CPU and 1330.26 GB/s for GPU

Having obtained these values we are now able to calculate a theoretical runtime for any given problem size and number of MPI ranks. For instance, fig. 4.12 shows the predicted timings for a problem of size $30 \times 30 \times 30$ with measured timings for various numbers of MPI ranks added in. As expected the runtime of the algorithm scales rather well with the number of MPI ranks. We also see how communication becomes more noticeable relative to the overlapped local computations as we add in more and more MPI ranks. Even though our measured timings lie above the line with the predicted timings, they are generally within a factor of 2 of the predicted timings and exhibit similar behavior and scaling. Given the complexity of the algorithm we consider this to be overall a good fit.

What the model does not tell us is the expected convergence, and consequently, time to converge. However, the model does provide a prediction of the effect on the overall solver with a change in either problem size of number of MPI ranks.

Even though our timings scale well with the number of MPI ranks, adding more MPI ranks degrades the convergence of this solver. We were able to solve a problem with a Reynolds number of $\mathscr{R} = 500$ with 4 MPI ranks (albeit very slow), with 50 MPI ranks we are not able to go far beyond $\mathscr{R} = 400$. We attempt to improve this situation with homotopy.

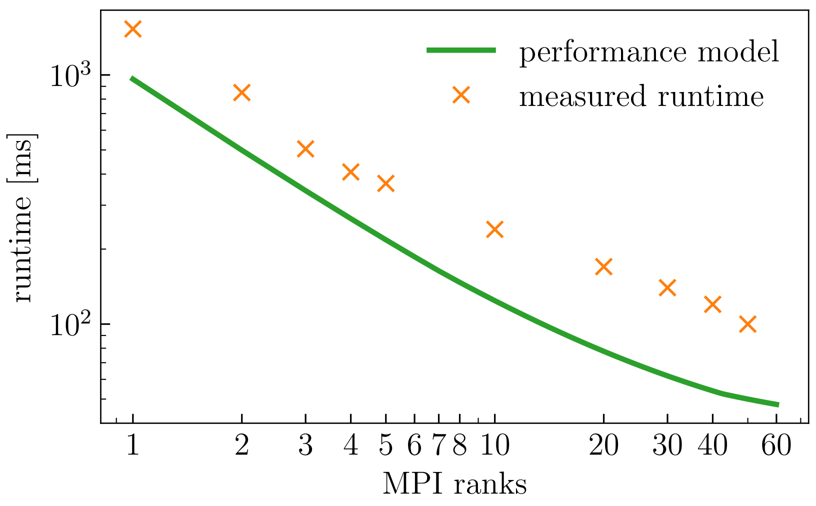


Figure 4.12: Performance model for AMG+RAS/ILU for problem of size $30 \times 30 \times 30$ with MPI ranks ranging from 1 to 60.

4.10.5 Homotopy or Continuation Methods

Homotopy (also known as continuation methods) [110–114] is an attempt to combine the capabilities of a solver to solve an "easy" version of some problem with a continuous transitioning from the easy problem to the target ("hard") problem by using information from previous runs to jump-start the next runs.

Given our model problem with a Reynolds number that is easy to solve with, $\overline{\mathscr{R}}$. Call the target Reynolds number that we want to solve with $\widehat{\mathscr{R}}$. Expressing our problem setup as the function G, we then create a new problem as

$$G(\mathscr{R}) = \lambda G(\overline{\mathscr{R}}) + (1 - \lambda) G(\widehat{\mathscr{R}}) \tag{4.76}$$

with $\lambda \in [0, 1]$. Setting $\lambda = 1$ equates to solving our model problem for the "easy" Reynolds number, $\mathscr{R} = \overline{\mathscr{R}}$. Setting $\lambda = 0$ equates to solving our model problem for the "hard" Reynolds number, $\mathscr{R} = \widehat{\mathscr{R}}$. Thus, changing λ step-by-step from 1 to 0 allows us to transition from the "easy" to the "hard" problem in a stepwise manner. For each new λ we make use of the solution to the problem with the old λ to give us a good initial guess for the harder problem.

The idea behind homotopy is based on the fact that Newton algorithms require an initial guess close to the solution in order to converge. Performing only a small change to our model problem should not alter our solution space too much, leaving the old solution as a good initial guess that is "close" to our new solution.

Adding homotopy to our solver algorithm as described in algorithm 4.6 results in the algorithm as given in algorithm 4.7. The main challenge with using such an algorithm is

Algorithm 4.7: Homotopy algorithm

```
Set starting Reynolds number $\mathscr{R}_{\text{init}}$ and target Reynolds number $\mathscr{R}_{\text{target}}$
Choose $\omega \in (0, 1] :=$ step size damping
Set $\tau_{\text{target}}$ = convergence tolerance
Set $\tau_{\text{step}}$ = intermediate convergence tolerance
Set $\mathbf{x} = x_0$ = initial guess
Set $\mathscr{R} = \mathscr{R}_{\text{init}}$
while $\mathscr{R} < \mathscr{R}_{target}$ do
    if not first iteration then
        Compute $\mathscr{R} = \mathscr{R} + \Delta\mathscr{R}$
    end
    if $\mathscr{R} < \mathscr{R}_{target}$ then
        Set $\tau = \tau_{\text{step}}$
    else
        Set $\tau = \tau_{\text{target}}$
    end
    Construct nonlinear form for current $\mathscr{R}$
    Compute $\mathbf{res}_0$ := initial residual
    while $\frac{res_k}{res_0} > \tau$ do
        Perform Newton linearization with linesearch
        for steps $\leftarrow$ 1..10 do
            Preconditioner: LDU factorization
                AMG+RAS/ILU for action of $F^{-1}$
                AMG+RAS/ILU for action of $S^{-1}$
            Perform fGMRES step
        end
    end
end
```

to figure out the optimal value for τ_{step} [113]. To derive a theoretical bound we start by constructing a differential problem,

$$\mathbf{u}_{\text{new}} = \mathbf{u}(\mathscr{R}) + \mathbf{u}^{(1)}(\mathscr{R})\tau_{\text{step}}, \tag{4.77}$$

which defines how a new initial guess can be constructed by using the previous solution. We then denote the typical weak formulation of the Navier-Stokes equations,

$$\mu a(\mathbf{u}, \mathbf{v}) + c(\mathbf{u}, \mathbf{u}, \mathbf{v}) + b(\mathbf{v}, p) + b(\mathbf{u}, q) = \langle \mathbf{f}, \mathbf{v} \rangle \quad \forall \mathbf{v} \in V, q \in P. \tag{4.78}$$

Formally differentiating this with $\mathbf{u}^{(k)} = \frac{d^k\mathbf{u}}{d\mathscr{R}^k}$ and $p^{(k)} = \frac{d^k p}{d\mathscr{R}^k}$,

$$\begin{aligned}\mu a(\mathbf{u}^{(k)},\mathbf{v}) + & c(\mathbf{u}^{(k)},\mathbf{u},\mathbf{v}) + c(\mathbf{u},\mathbf{u}^{(k)},\mathbf{v}) + b(\mathbf{v},p^{(k)}) + b(\mathbf{u}^{(k)},q) \\ &= \mathbb{L}_k(\mathbf{u}^{(k-1)},\ldots,\mathbf{u}^{(1)},\mathbf{u};\mathbf{v};Re) \quad \forall \mathbf{v}\in V, q\in P, k=1,2,\ldots \end{aligned} \tag{4.79}$$

which is rearranged to obtain

$$\begin{aligned}\mathbb{L}_k(\mathbf{u}^{(k-1)},\ldots,\mathbf{u}^{(1)},\mathbf{u};\mathbf{v};\mathscr{R}) = & \sum_{j=0}^{k-1} \frac{\alpha_{kj}}{(-\mathscr{R})^{k+1-j}} a(\mathbf{u}^{(j)},\mathbf{v}) \\ & - \sum_{j=1}^{k-1} \beta_{kj} c(\mathbf{u}^{(j)},\mathbf{u}^{(k-j)},\mathbf{v}) \quad \forall \mathbf{v}\in H_0^1(\Omega) \end{aligned} \tag{4.80}$$

with $\mathbf{u}^{(0)} = \mathbf{u}$ and both $\alpha_{kj}$and β_{kj} not depending on the Reynolds number $\mathscr{R}$. If we, then, choose $\mathbf{v} = \mathbf{u}^{(k)}$ and $q = -p^{(k)}$ and apply the properties of the bilinear norms, we obtain

$$\frac{\sigma}{\mathscr{R}}\|\mathbf{u}^{(k)}\| \leq \sum_{j=0}^{k-1} \frac{\alpha_{kj}}{\mathscr{R}^{k+1-j}}\|\mathbf{u}^{(j)}\| + \gamma \sum_{j=1}^{k-1} \beta_{kj} \|^{(j)}\| \cdot \|\mathbf{u}^{(k-j)}\| \tag{4.81}$$

with $\sigma = 1 - \gamma\|\mathbf{u}\|\mathscr{R}$. If we assume we have a solution $\mathbf{u}(\overline{\mathscr{R}})$ with $\mathscr{R}_{\text{target}} > \overline{\mathscr{R}}$, then

$$\mathbf{u}_{\text{new}} = \mathbf{u}(\overline{\mathscr{R}}) + \mathbf{u}^{(1)}(\overline{\mathscr{R}})\tau_{\text{step}} \tag{4.82}$$

where $\tau_{\text{step}} = \mathscr{R} - \overline{\mathscr{R}}$ and $\mathbf{u}^{(1)}$ the solution to the differentiated problem given in eq. (4.80) with $k = 1$ and $\mathscr{R} = \overline{\mathscr{R}}$. The standard mean value theorem is now applied to get the relation

$$\begin{aligned}\|\mathbf{u}(\mathscr{R}) - \mathbf{u}_{\text{new}}\| &= \|\mathbf{u}(\mathscr{R}) - \mathbf{u}(\overline{\mathscr{R}}) - \mathbf{u}^{(1)}(\overline{\mathscr{R}})\tau_{\text{step}}\| \\ &\leq \sup_{\overline{\mathscr{R}}\leq\xi\leq\mathscr{R}} \frac{1}{2}\left\|\frac{d^2\mathbf{u}}{d\mathscr{R}^2}(\xi)\right\|(\Delta\mathscr{R})^2. \end{aligned} \tag{4.83}$$

where $\xi \in [\overline{\mathscr{R}},\mathscr{R}]$ is such that the tangent at ξ is parallel to the secant line through the two endpoints. If $0 < \hat{\sigma} \leq \sigma(\xi) < 1$ for $\xi \in [\overline{\mathscr{R}},\mathscr{R}]$,then with $k = 2$ we have that

$$\left\|\frac{d^2\mathbf{u}}{d\mathscr{R}^2}(\xi)\right\| \leq \frac{4}{\overline{\mathscr{R}}^2\hat{\sigma}^3}\|\mathbf{u}(\xi)\| \tag{4.84}$$

which is then plugged into eq. (4.83) to obtain

$$\|\mathbf{u}(\mathscr{R}) - \mathbf{u}_{\text{new}}\| \leq \frac{2}{\overline{\mathscr{R}}^2\hat{\sigma}^3}\|\mathbf{u}(\mathscr{R})\|(\Delta\mathscr{R})^2. \tag{4.85}$$

This relation is assured if we have that

$$\|\mathbf{u}(\mathscr{R}) - \mathbf{u}_{\text{new}}\| \leq \frac{\hat{\sigma}}{2}\|\mathbf{u}(\mathscr{R})\|. \tag{4.86}$$

Combining both equation eq. (4.85) and eq. (4.86) we obtain that $\mathbf{u}_{\text{new}}$ lies in the attraction ball at $\mathscr{R}$ whenever

$$\tau_{\text{step}} < \overline{\mathscr{R}}\frac{\hat{\sigma}^2}{2}. \tag{4.87}$$

Actually determining the value of this bound, however, is difficult. For instance, near bifurcation points we need $\hat{\sigma} \to 0$ resulting in a very small step size. One way to reliably determine the optimal step size is by numerical experiment.

The potential of homotopy can be best illustrated by looking at an example given in table 4.4. Solving our model problem for $\mathscr{R} = 400$ takes a total of 10 nonlinear iterations

	continuation		without continuation	
Reynolds number	# nonlinear iteration	time [s]	# nonlinear iteration	time [s]
100	3	664	3	664
400	2	481	10	2214
500	4	949	-	-
600	11	2649	-	-

Table 4.4: Comparison of continuation approach to direct solves.

with a total runtime of 2214 seconds (about 37 minutes). However, first solving for $\mathscr{R} = 100$ and then using the obtained solution to solve for $\mathscr{R} = 400$ takes 3 nonlinear iterations and 664 seconds (about 11 minutes) for the first solve and then 2 iterations and 481 seconds (about 8 minutes) for the seconds solve, for an overall total of 1145 seconds (about 19 minutes). Thus, adding the intermediate solve at $\mathscr{R} = 100$ cuts the required time to convergence for the problem with $\mathscr{R} = 400$ roughly in half.

Using homotopy we are also able to push our solver to higher Reynolds numbers. Continuing on with the above problem it is then possible to solve the problem with $\mathscr{R} = 500$ with another 4 iterations and 949 seconds (about 16 minutes), and the problem with $\mathscr{R} = 600$ with another 11 iterations and 2649 seconds (about 44 minutes). Once $\mathscr{R} = 700$ is reached, however, our solver hits a wall almost instantly and struggles to converge any further.

Since our solver works well for small to moderate Reynolds numbers, the source of instability in iteration is often associated with errors arising from aliasing [115]. These errors arise from the fact that a discrete grid is used to represent our nonlinear term and are typically due to an insufficient quadrature degree of the convective term. Figure 4.13 and table 4.5

show the effect of the quadrature rule for $\mathscr{R} = 100$ when run up to a convergence tolerance of 10^{-6}. As can be seen from both fig. 4.13 and table 4.5 the quadrature degree does indeed

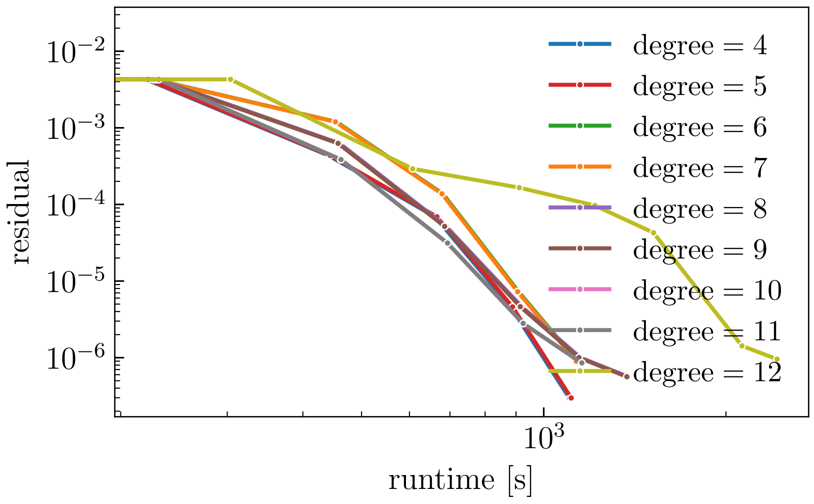


Figure 4.13: Effect of quadrature degree on convergence, $30 \times 30 \times 30$ elements, $\mathscr{R} = 100$, tol $= 10^{-6}$.

quadrature degree	iteration count	runtime [s]	final residual norm
4	5	1102	2.96×10^{-7}
5	5	1110	2.98×10^{-7}
6	5	1135	8.97×10^{-7}
7	5	1131	8.97×10^{-7}
8	6	1379	5.64×10^{-7}
9	6	1371	5.64×10^{-7}
10	5	1154	8.55×10^{-7}
11	5	1156	8.55×10^{-7}
12	8	2429	9.56×10^{-7}

Table 4.5: Effect of quadrature degree on convergence, $30 \times 30 \times 30$ elements, $\mathscr{R} = 100$, tol $= 10^{-6}$.

have an effect on the time-to-convergence. Choosing the wrong quadrature degree can lead to a doubling in time-to-convergence, as a careful balance between a high enough degree to be able to represent the underlying polynomials exactly and a low enough degree to not introduce unnecessary computational work. As we are working with Q2/Q1 Taylor-Hood finite elements in three dimensions, we expect a quadrature degree of 6 to be sufficient to avoid additional error terms. As can be seen from table 4.5, there is very little different in the required time-to-convergence for quadrature degrees 4 to 7. The choice of quadrature degree only has a minor, if any, impact on the overall convergence behavior, requiring the same number of non-linear iterations and resulting in almost the same final residual norm.

Based on this experience we set up our solver to solve our model problem for a Reynolds number of 1000. We combined various pieces of our solver that we know already by setting the first two intermediate Reynolds numbers to be 100 and 400. Next we try $\mathscr{R} = 800$, followed by a value of 900 and 1000. The convergence tolerance for the intermediate Reynolds numbers was set to 10^{-4} with the final tolerance at $\mathscr{R} = 1000$ set to 10^{-6}. The result is shown in fig. 4.14. As can be seen from fig. 4.14, the solver initially works very well for $\mathscr{R} = 100$ and

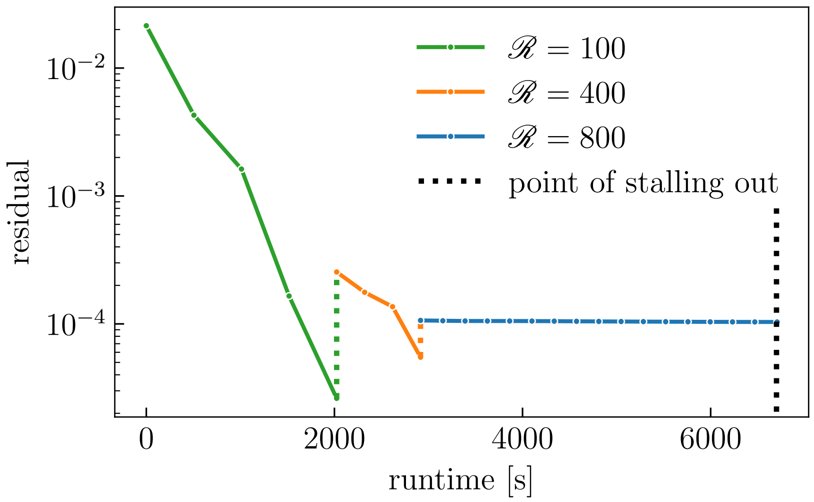


Figure 4.14: Running our solver from $\mathscr{R} = 100$ to $\mathscr{R} = 1000$ with intermediate Reynolds number of 400 and 800.

subsequently $\mathscr{R} = 400$, confirming the findings presented in table 4.4. However, the next solve for $\mathscr{R} = 800$ starts out with an initial guess very close to the convergence tolerance, though it struggles to make any further progress and essentially stalls out at this point.

4.10.6 Dynamic Solver

Finding a way to resolve the issue that the solver gets stuck when the "wrong" next Reynolds number is chosen required making the solver dynamic. After solving an initial system, we first increase the Reynolds number by a fixed amount and then run the solver for a new problem. There are three possible scenarios that arise:

1. the old solution is still a good enough (i.e., within the given tolerance) solution for the new problem;

2. the solver might stall out, which we define to be the case when consecutive nonlinear steps differ by less than 10^{-4};
3. the solver converges.

If the solver does not converge we apply increasing simplifications to the solver in order to get it to converge. This process is repeated until we reach the target Reynolds number. A flow chart of how exactly the solve works in practice is given in fig. 4.15, with an algorithmic description of the solver provided in algorithm 4.8.

After setting up some initial variables, we solve our initial model problem for a Reynolds number of $\mathscr{R} = 100$. We know from the previous analysis that this is a problem our solver is well capable of handling. After each completion of the solver we check whether the solver converged or stalled. If it converged, we check if any iterations were performed, as it is possible that the solution to the previous problem is a sufficient solution for our current problem. In that case we increase the Reynolds by 50 and restart the solver. If one or more iterations were performed we increment the Reynolds number by 150 and restart the solver using the current solution as new initial guess.

If the solver did not converge then at first we reduce the Reynolds number increment from 150 to 100, and, if the solver still does not converge, we further reduce it to 50. If the solver continues to stall, we reduce the Newton damping factor slightly and/or further reduce the Reynolds number increment to 25. If despite all our efforts the solver still does not converge we consider the solve to have failed.

As long as we are not solving the problem with our target Reynolds number it is sufficient to run our solver to a looser convergence tolerance. In practice we use 10^{-4} as convergence tolerance for the solves with the intermediate Reynolds numbers, and 10^{-6} for the solves with the final target Reynolds number.

To test this dynamic solver we ran it on Delta with a total of 768 MPI ranks across 12 nodes (64 ranks per node) for a problem of size $40 \times 40 \times 40$, resulting in almost 2 million degrees of freedom in the global mesh. The results of this run is shown in fig. 4.16 with a summary presented in table 4.6. We see that the solver starts out strong, but stalls out first at $\mathscr{R} = 600$. After reducing the Reynolds number the solve succeeds for $\mathscr{R} = 550$. The solver struggles to move past $\mathscr{R} = 750$ but with increasing simplifications and modifications to the problem it eventually succeeds. The final solve for $\mathscr{R} = 1000$ at first starts out slow, before it abruptly converges rapidly to the final tolerance of 10^{-6}. From this data we can see that our solver indeed works as intended, it is capable of tuning its own parameters to overcome stalls.

It is worth noting that such an autonomous run of the solver does not need to be done every time. Once a successful continuation path is known, the solver can be instructed as to what path to choose. This allows the solver to skip all intermediate failing solves, and reduces the required runtime in this case by about 30%.

This solver, even though we have shown it to be working for our model problem, is still not a fast solver. With a prescribed continuation path the solver still requires about 8 hours to complete. There are a many tweaks and optimizations possible to further improve on this solver. See section 4.10.8 for a list of several possible future avenues to pursue.

4.10.7 Conclusions

In this chapter we looked at the RAS/ILU algorithm in the context of starting the development of a solver for the RANS equations arising in the context of wind turbine modeling. We observed the challenges posed by such a problem and how the commonly used solvers and preconditioners struggle to reach convergence for small to moderately high Reynolds numbers. We then presented an overview of both the RAS and the ILU algorithms and showed how in recent years RAS/ILU has increasingly become an intriguing and competitive choice of solver.

Working in the context of Firedrake, PETSc, and hypre, we explored the necessary steps and additions to the respective libraries that were necessary in order to implement these algorithms and use them as part of a larger solver. The solver we designed is comprised of several different parts, including Newton linearization with linesearch, GMRES for the linear solves, and AMG with RAS/ILU as a relaxation schemes to precondition some of the GMRES solves. We presented results to show that this solver is indeed well capable of solving the model problem for moderately high Reynolds numbers. Profiling the solver showed that the majority of time is spent performing AMG with RAS/ILU relaxation. A performance model for this algorithm showed that our measured timings for this preconditioner are indeed not far off from our predicted timings. This model gave us a better understanding of how the algorithm behaves for different problem sizes and number of MPI ranks.

Next we looked at homotopy and showed that instead of directly solving our target problem it is advantageous to solve one or more easier intermediate problems. This can not only cut down the required time-to solution, but also enables the solve of more difficult problems than we could solve without homotopy.

At the moment, Firedrake is not ready to run on the GPU, restricting our experiments to the CPU. However, work on adding GPU capabilities to Firedrake is in progress and, once completed, will allow the use of our interface in PETSc to access the GPU implementation of AMG and RAS/ILU in hypre. The parallel nature of these algorithms suggests that they are likely well suited for GPUs.

4.10.8 Future Directions

In this chapter we explored various techniques for solving our model problem. However, there are various approaches and techniques that we did not explore but that might provide valuable and interesting improvements to the solver. Some of the possible avenues to pursue in the future further include:

- pseudo time-stepping [116]: Even though we are considering steady-state problems, pseudo time-stepping is an approach that solves for the steady-state solution by using a time-stepper to evolve an initial guess to the steady-state problem. One of the likely difficulties with this approach is that small time steps make the pressure term very stiff.

- arc-length continuation model [110]: There are variations to the homotopy approach as described here. In particular, when progressing along the path for the Reynolds numbers, if the solver stalls then a different way to progress other than simply increasing the Reynolds number is to progress along the arc-length S of the solution. It is possible to show that a critical point of the problem

$$F(\mathbf{u}, \mathscr{R}) = 0 \tag{4.88}$$

  is a regular point of the following problem: Find $\mathbf{u}(S)$, $\mathscr{R}(S)$ such that

$$F(\mathbf{u}, \mathscr{R}) = 0 \text{ and } \left\| \frac{d\mathbf{u}}{dS} \right\|^2 + \left| \frac{d\mathscr{R}}{dS} \right|^2 = 1 \text{ in } \mathbb{R} \tag{4.89}$$

- projection with 2 steps [117, 118]: Another possible approach is to solve the problem in 2 steps. First, we solve $\mathbf{u} = F^{-1} rhs$ where $\mathbf{u}$ is the velocity part of the solution. Next, we project the thus-found solution onto the divergence-free space, and iterate back to the first step.

- overset meshes [119]: Looking at the problem setup and the solution, it is obvious that most of what we are interested in is happening around and behind the turbine. Thus, we do not necessarily need the same mesh resolution everywhere in our domain. It is possible to cover the full domain in a mesh with low resolution and overlay a grid with higher resolution around the areas of interest. This allows us to focus our efforts on what we are most interested in modeling.

- Newton linearization is a well understood algorithm that is currently used for similar problems at NREL. It was chosen in part for that reason so that we can focus our attention on the preconditioning algorithms. However, there exists a wide range of other nonlinear solvers that could possibly be combined with a preconditioning scheme based on RAS/ILU for such a problem.

4.11 RESOURCES

The sample Firedrake code used for the analysis in this chapter is listed in appendix A. The modified PETSc interface to include access to the RAS/ILU algorithm in hypre is found in the git repository at `https://github.com/luspi/petsc`.

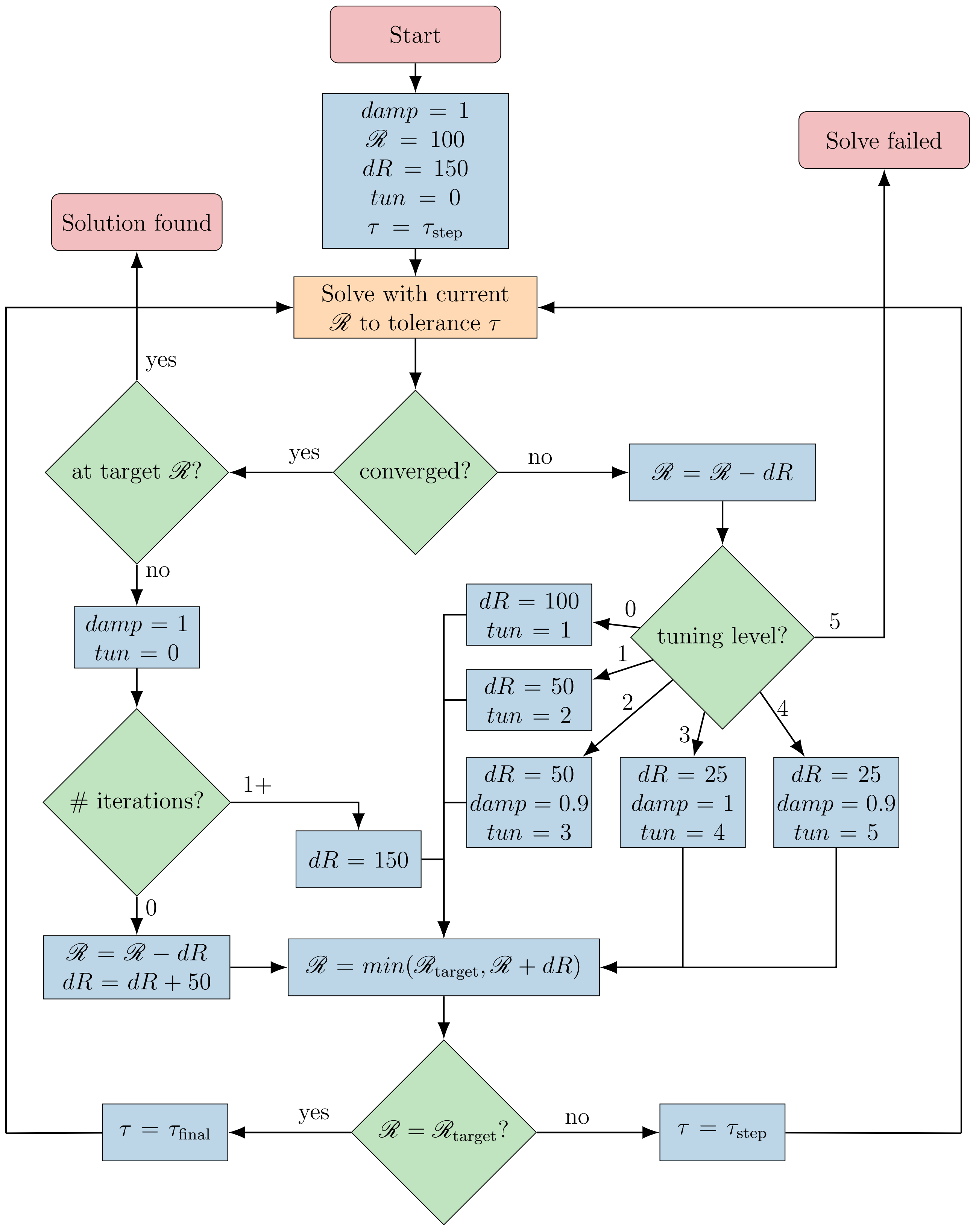

Start
$damp = 1$
$\mathscr{R} = 100$
$dR = 150$
$tun = 0$
$\tau = \tau_{\text{step}}$
Solve failed
Solution found
Solve with current $\mathscr{R}$ to tolerance τ
yes
at target $\mathscr{R}$?
yes
converged?
no
$\mathscr{R} = \mathscr{R} - dR$
tuning level?
no
$damp = 1$ $tun = 0$
$dR = 100$ $tun = 1$
0
5
1
$dR = 50$ $tun = 2$
2
4
3
# iterations?
1+
$dR = 50$ $damp = 0.9$ $tun = 3$
$dR = 25$ $damp = 1$ $tun = 4$
$dR = 25$ $damp = 0.9$ $tun = 5$
$dR = 150$
0
$\mathscr{R} = \mathscr{R} - dR$ $dR = dR + 50$
$\mathscr{R} = min(\mathscr{R}_{\text{target}}, \mathscr{R} + dR)$
yes
no
$\tau = \tau_{\text{final}}$
$\mathscr{R} = \mathscr{R}_{\text{target}}$?
$\tau = \tau_{\text{step}}$

Figure 4.15: Flow chart of dynamic solver, with *damp* the damping factor, dR the next update to the Reynolds number, *tun* the level of tuning after failed solves, and τ the convergence tolerance.

Algorithm 4.8: Dynamic solver

```
Set initial variables: $damp = 1.0$, $\mathscr{R} = 100$, $dR = 200$, $tun = 0$, $\tau = \tau_{\text{step}}$
while solution not yet found do
    Run solver with current Reynolds number, $\mathscr{R}$, to tolerance, $\tau$.
    if Solve has converged then
        if $\mathscr{R} == \mathscr{R}_{target}$ then
            Mark solution as found.
            break;
        else
            Reset variables $damp = 1$, $tun = 0$.
            if iteration count $< 3$ then
                Revert Reynolds number, $\mathscr{R} = \mathscr{R} - dR$.
                Increase step size, $dR = \min(\mathscr{R}_{\text{target}} - \mathscr{R}, dR + 50)$.
            else
                Reset step size, $dR = 200$.
            end
        end
    else
        Revert Reynolds number, $\mathscr{R} = \mathscr{R} - dR$.
        if $tun == 0$ then
            Reduce Reynolds step size, $dR = 100$, and increase tuning level, $tun = 1$.
        else if $tun == 1$ then
            Reduce Reynolds step size, $dR = 50$, and increase tuning level, $tun = 2$.
        else if $tun == 2$ then
            Reduce damping factor, $damp = 0.9$, and increase tuning level, $tun = 3$.
        else if $tun == 3$ then
            Reduce Reynolds step size, $dR = 25$, reset damping factor, $damp = 1.0$,
              and increase tuning level, $tun = 4$.
        else if $tun == 4$ then
            Reduce damping factor, $damp = 0.9$, and increase tuning level, $tun = 5$.
        else
            Mark solve as failed.
            break
        end
    end
    Calculate new Reynolds number, $\mathscr{R} = \min(\mathscr{R}_{\text{target}}, \mathscr{R} + dR)$
    if $\mathscr{R} == \mathscr{R}_{target}$ then
        Set tolerance, $\tau = \tau_{\text{final}}$
    else
        Set tolerance, $\tau = \tau_{\text{step}}$
    end
end
```

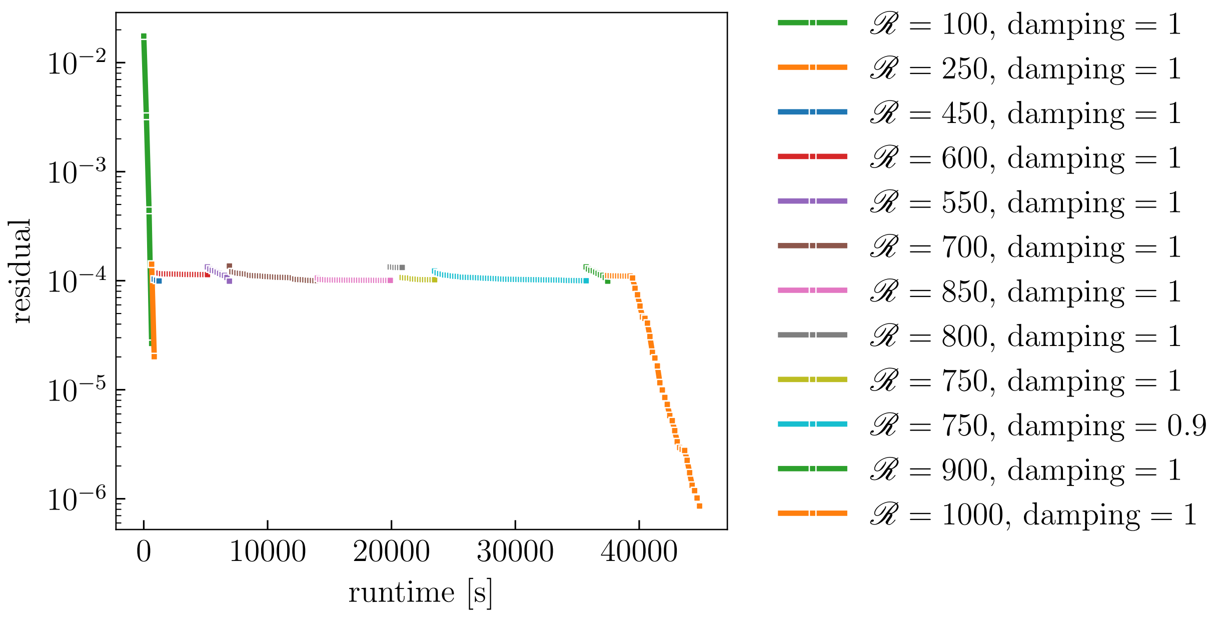


Figure 4.16: Convergence behavior of dynamic solver from $\mathscr{R} = 100$ to $\mathscr{R} = 1000$.

$\mathscr{R}$	damping factor	tolerance τ	dR	# iterations	total time [s]
100	1	10^{-4}	-	3	638.0
250	1	10^{-4}	150	1	208.0
400	1	10^{-4}	150	0	0.6
450	1	10^{-4}	200	2	389.0
600	1	10^{-4}	150	19	3906.0
550	1	10^{-4}	100	9	1773.0
700	1	10^{-4}	150	35	7109.0
850	1	10^{-4}	150	30	5881.0
800	1	10^{-4}	100	5	960.0
750	1	10^{-4}	50	13	2618.0
750	0.9	10^{-4}	50	60	12 233.0
900	1	10^{-4}	150	9	1743.0
1000	1	10^{-6}	150	28	7414.0
Total				214	44 872.6

Table 4.6: Continuation path of dynamic solver from $\mathscr{R} = 100$ to $\mathscr{R} = 1000$ with failing solves highlighted.

CHAPTER 5: CONCLUSION

In the preceding chapter an overview of various research topic has been given that together compose my work on improving the performance of iterative algorithms in various ways. In particular we considered three parts.

1. **Halo exchanges across large heterogeneous supercomputers.** We gave an overview of a new halo exchange library we designed that implements an efficient and flexible way to move halo data across large heterogeneous machines using MPI either alone or in combination with OpenCL, HIP, and CUDA. A paper on this work has been published in the December 2022 issue of the journal *Parallel Computing*, `https://doi.org/10.1016/j.parco.2022.102973`.

2. **Smoothers for the Stokes equations.** In this chapter we explored different relaxation schemes that are used within a multigrid preconditioner for the GMRES iterative solver for solving the Stokes equations. Taking advantage of a highly structured discretization, we performed an exhaustive performance analysis of two popular relaxation schemes Braess-Sarazin and Vanka, and compared their performance with an emphasis on the GPU to each other and also to Schur-Uzawa and a Block-Triangular preconditioner. We learned that Vanka is performing better than Braess-Sarazin for this setup on the GPU while on the CPU Braess-Sarazin has the clear advantage. Both the Block-Triangular preconditioner and Multigrid paired with Schur-Uzawa are not able to achieve comparable performance on either the CPU or the GPU. A paper on this work has been accepted for publication in May 2024 in the *International Journal of High Performance Computing Applications*.

3. **RAS+ILU for the Reynolds-Averaged Navier-Stokes equations.** In the third chapter we presented the RANS equations and showed how they are derived. After detailing the particulars of our sample application - modeling the wind flow around a turbine located in an open field - we presented some common strategies for modeling the Reynolds-stress tensor. Presenting a range of preconditioners that are used for solving the RANS equations, we illustrated how they struggle for this model problem and break down rather quickly. We then presented the RAS/ILU algorithm and designed a solver that takes advantage of this algorithm and showed how we are able to push this solver to much higher Reynolds number than what we were able to do before, including the use of homotopy or continuation methods. We also performed a detailed analysis including a performance model for this algorithm, and showed that the performance of

the implementation is not far off from the predicted performance. Lastly we proposed a dynamic solver that is capable of self-tuning its homotopy parameters. This work is being prepared for publication at the moment.

REFERENCES

[1] H. C. Edwards, C. R. Trott, and D. Sunderland, "Kokkos: Enabling manycore performance portability through polymorphic memory access patterns," *journal of Parallel and Distributed Computing*, vol. 74, no. 12, pp. 3202 – 3216, 2014, domain-Specific Languages and High-Level Frameworks for High-Performance Computing. https://doi.org/10.1016/j.jpdc.2014.07.003

[2] S. R. Slattery, P. P. H. Wilson, and R. P. Pawlowski, *The Data Transfer Kit: A geometric rendezvous-based tool for multiphysics data transfer.* American Nuclear Society - ANS, 7 2013. https://www.osti.gov/biblio/22212795

[3] M. Bianco, "An Interface for Halo Exchange Pattern," 2013. https://doi.org/10.5281/zenodo.831983

[4] "Generic Communication Layer," 2020, accessed: 2020-04-27. https://github.com/eth-cscs/gcl

[5] "RAJA performance portability layer (C++)," 2020, accessed: 2021-05-29. https://github.com/LLNL/RAJA

[6] D. A. Beckingsale, J. Burmark, R. Hornung, H. Jones, W. Killian, A. J. Kunen, O. Pearce, P. Robinson, B. S. Ryujin, and T. R. Scogland, "RAJA: Portable performance for large-scale scientific applications," in *2019 IEEE/ACM International Workshop on Performance, Portability and Productivity in HPC (P3HPC)*, 2019. https://doi.org/10.1109/P3HPC49587.2019.00012 pp. 71–81.

[7] C. Pearson, K. Wu, I.-H. Chung, J. Xiong, and W.-M. Hwu, "TEMPI: An interposed MPI library with a canonical representation of CUDA-aware datatypes," 2021. https://doi.org/10.1145/3431379.3460645

[8] C. G. Baker and M. A. Heroux, "Tpetra, and the use of generic programming in scientific computing," *Sci. Program.*, vol. 20, no. 2, pp. 115–128, Apr. 2012. https://doi.org/10.1155/2012/693861

[9] M. A. Heroux, R. A. Bartlett, V. E. Howle, R. J. Hoekstra, J. J. Hu, T. G. Kolda et al., "An overview of the Trilinos project," *ACM Transactions on Mathematical Software*, vol. 31, no. 3, pp. 397–423, September 2005. https://doi.org/10.1145/1089014.1089021

[10] "MiniGhost halo exchange mini-application," 2020, accessed: 2020-04-27. https://github.com/Mantevo/miniGhost

[11] R. Barrett, M. Heroux, and C. Vaughan, "MiniGhost : a miniapp for exploring boundary exchange strategies using stencil computations in scientific parallel computing," 01 2012. https://doi.org/10.2172/1039405

[12] "Mantevo Project," 2020, accessed: 2020-04-27. https://mantevo.github.io

[13] M. P. I. Forum, "MPI: A message-passing interface standard," 2015, version 3.0. https://www.mpi-forum.org/docs/mpi-3.0/mpi30-report.pdf

[14] "CUDA C++ programming guide." https://docs.nvidia.com/cuda/

[15] "HIP documentation." https://rocm.docs.amd.com/projects/HIP/en/latest/index.html

[16] Khronos OpenCL Working Group, "The OpenCL specification," 2018, version 1.2. https://www.khronos.org/registry/OpenCL/specs/opencl-2.1.pdf

[17] W. Gropp, L. N. Olson, and P. Samfass, "Modeling MPI communication performance on SMP nodes: Is it time to retire the ping pong test," in *Proceedings of the 23rd European MPI Users' Group Meeting*, ser. EuroMPI 2016. Association for Computing Machinery, 2016. https://doi.org/10.1145/2966884.2966919 pp. 41–50.

[18] "High performance computing conjugate gradients (HPCCG)," 2021, accessed: 2021-01-08. https://github.com/Mantevo/HPCCG

[19] M. Project, "miniFE finite element mini-application," 2022, accessed: 2022-02-22. https://github.com/Mantevo/miniFE

[20] D. Moulton, L. N. Olson, and A. Reisner, "Cedar framework," 2017, version 0.1. https://github.com/cedar-framework/cedar

[21] A. Malevsky, "Message-passing tools for structured grid communications user's guide version 2.0," centre de Recherche en Calcul Appliqué, 5160 boul. Décarie, bureau 400, Montréal, Québec.

[22] A. R. Reisner, J. D. Moulton, M. Berndt, and L. N. Olson, "Scalable line and plane relaxation in a parallel structured multigrid solver," *Parallel Computing*, vol. 100, 10 2020. https://doi.org/10.1016/j.parco.2020.102705

[23] A. Reisner, L. N. Olson, and J. D. Moulton, "Scaling structured multigrid to 500k+ cores through coarse-grid redistribution," *SIAM Journal on Scientific Computing*, vol. 40, no. 4, pp. C581–C604, 2018. https://doi.org/10.1137/17M1146440

[24] Y. Dou and Z.-Z. Liang, "A class of block alternating splitting implicit iteration methods for double saddle point linear systems," *Numerical Linear Algebra with Applications*, vol. 30, no. 1, p. e2455, 2023. https://doi.org/10.1002/nla.2455

[25] F. Nataf and P.-H. Tournier, "Recent advances in domain decomposition methods for large-scale saddle point problems," *Comptes Rendus. Mécanique*, 2022, online first. https://doi.org/10.5802/crmeca.130

[26] S. V. Ershkov, E. Y. Prosviryakov, N. V. Burmasheva, and V. Christianto, “Towards understanding the algorithms for solving the Navier-Stokes equations,” *Fluid Dynamics Research*, vol. 53, no. 4, p. 044501, 07 2021. https://doi.org/10.1088/1873-7005/ac10f0

[27] H. Elman, D. Silvester, and A. Wathen, *Finite elements and fast iterative solvers: with applications in incompressible fluid dynamics*, ser. Numerical Mathematics and Scientific Computation. New York: Oxford University Press, 2005. https://doi.org/10.1093/acprof:oso/9780199678792.001.0001

[28] M. Benzi, G. Golub, and J. Liesen, “Numerical solution of saddle point problems,” *Acta Numer.*, vol. 14, pp. 1–137, 2005. https://doi.org/10.1017/s0962492904000212

[29] Y. Notay, “Convergence of some iterative methods for symmetric saddle point linear systems,” *SIAM Journal on Matrix Analysis and Applications*, vol. 40, no. 1, pp. 122–146, 2019. https://doi.org/10.1137/18M1208836

[30] A. Brandt and N. Dinar, “Multigrid solutions to elliptic flow problems,” in *Numerical Methods for Partial Differential Equations*, S. V. PARTER, Ed. Academic Press, 1979, pp. 53–147. https://doi.org/10.1016/B978-0-12-546050-7.50008-3

[31] A. Voronin, Y. He, S. MacLachlan, L. N. Olson, and R. Tuminaro, “Low-order preconditioning of the Stokes equations,” *Numerical Linear Algebra with Applications*, vol. 29, no. 3, p. e2426, 2022. https://doi.org/10.1002/nla.2426

[32] D. Braess and R. Sarazin, “An efficient smoother for the Stokes problem,” *Appl. Numer. Math.*, vol. 23, no. 1, pp. 3–19, 1997, multilevel methods (Oberwolfach, 1995). http://doi.org/10.1016/S0168-9274(96)00059-1

[33] W. Zulehner, “A class of smoothers for saddle point problems,” *Computing*, vol. 65, no. 3, pp. 227–246, 2000. https://doi.org/10.1007/s006070070008

[34] S. R. Franco, C. Rodrigo, F. J. Gaspar, and M. A. V. Pinto, “A multigrid waveform relaxation method for solving the poroelasticity equations,” *Computational and Applied Mathematics*, vol. 37, pp. 4805–4820, 2018. https://doi.org/10.1007/s40314-018-0603-9

[35] J. F. Maitre, F. Musy, and P. Nignon, “A fast solver for the Stokes equations using multigrid with a UZAWA smoother,” in *Advances in Multi–Grid Methods*, ser. Notes on Numerical Fluid Mechanics, D. Braess, W. Hackbusch, and U. Trottenberg, Eds., vol. 11. Braunschweig: Vieweg, 1984. https://doi.org/10.1007/978-3-663-14245-4_8 pp. 77–83.

[36] J. H. Adler, T. R. Benson, E. C. Cyr, S. P. MacLachlan, and R. S. Tuminaro, “Monolithic multigrid methods for two-dimensional resistive magnetohydrodynamics,” *SIAM J. Sci. Comput.*, vol. 38, no. 1, pp. B1–B24, 2016. http://doi.org/10.1137/151006135

[37] P. E. Farrell, Y. He, and S. P. MacLachlan, “A local Fourier analysis of additive Vanka relaxation for the Stokes equations,” *Numerical Linear Algebra with Applications*, vol. 28, no. 3, p. e2306, 2021. https://doi.org/10.1002/nla.2306

[38] J. Adler, Y. He, X. Hu, S. MacLachlan, and P. Ohm, "Monolithic multigrid for a reduced-quadrature discretization of poroelasticity," *SIAM J. Sci. Comput.*, vol. 45, no. 3, pp. S54–S81, 2023. https://doi.org/10.1137/21m1429072

[39] V. John and L. Tobiska, "Numerical performance of smoothers in coupled multigrid methods for the parallel solution of the incompressible Navier-Stokes equations," *International Journal For Numerical Methods In Fluids*, vol. 33, no. 4, pp. 453–473, Jan 2000. https://doi.org/10.1002/1097-0363(20000630)33:4%3C453::aid-fld15%3E3.0.co;2-0

[40] M. Paisley and N. Bhatti, "Comparison of multigrid methods for neutral and stably stratified flows over two-dimensional obstacles," *J. Comput. Phys.*, vol. 142, no. 2, pp. 581–610, 1998. https://doi.org/10.1006/jcph.1998.5915

[41] M. Larin and A. Reusken, "A comparative study of efficient iterative solvers for generalized Stokes equations," *Numer. Linear Algebra Appl.*, vol. 15, no. 1, pp. 13–34, 2008. http://doi.org/10.1002/nla.561

[42] J. H. Adler, T. R. Benson, and S. P. MacLachlan, "Preconditioning a mass-conserving discontinuous Galerkin discretization of the Stokes equations," *Numerical Linear Algebra with Applications*, vol. 24, no. 3, p. e2047, 2017. https://doi.org/10.1002/nla.2047

[43] C. Greif and Y. He, "A closed-form multigrid smoothing factor for an additive Vanka-type smoother applied to the Poisson equation," *Numerical Linear Algebra with Applications*, vol. 30, no. 5, p. e2500, 2023. https://doi.org/10.1002/nla.2500

[44] P. Munch and M. Kronbichler, "Cache-optimized and low-overhead implementations of additive Schwarz methods for high-order FEM multigrid computations," *The International Journal of High Performance Computing Applications*, p. 10943420231217221, 2023. https://doi.org/10.1177/10943420231217221

[45] A. Bienz, R. D. Falgout, W. Gropp, L. N. Olson, and J. B. Schroder, "Reducing parallel communication in algebraic multigrid through sparsification," *SIAM Journal on Scientific Computing*, vol. 38, no. 5, pp. S332–S357, 2016. https://doi.org/10.1137/15m1026341

[46] A. Bienz, W. D. Gropp, and L. N. Olson, "Reducing communication in algebraic multigrid with multi-step node aware communication," *The International Journal of High Performance Computing Applications*, vol. 34, no. 5, pp. 547–561, 2020. https://doi.org/10.1177/1094342020925535

[47] J. Dendy, "Black box multigrid," *Journal of Computational Physics*, vol. 48, no. 3, pp. 366–386, 1982. https://doi.org/10.1016/0021-9991(82)90057-2

[48] A. Reisner, M. Berndt, J. D. Moulton, and L. N. Olson, "Scalable line and plane relaxation in a parallel structured multigrid solver," *Parallel Computing*, vol. 100, p. 102705, 2020. https://doi.org/10.1016/j.parco.2020.102705

[49] M. ur Rehman, T. Geenen, C. Vuik, G. Segal, and S. P. MacLachlan, “On iterative methods for the incompressible Stokes problem,” *International journal for Numerical Methods in Fluids*, vol. 65, no. 10, pp. 1180–1200, 2011. https://doi.org/10.1002/fld.2235

[50] B. Ayuso de Dios, F. Brezzi, L. D. Marini, J. Xu, and L. Zikatanov, “A simple preconditioner for a discontinuous Galerkin method for the Stokes problem,” *J. Sci. Comput.*, vol. 58, no. 3, pp. 517–547, 2014. http://doi.org/10.1007/s10915-013-9758-0

[51] J. R. Rice and R. F. Boisvert, “Solving elliptic problems using ELLPACK,” *Mathematics and Computers in Simulation*, vol. 28, no. 4, pp. 339–340, 1986. https://doi.org/10.1016/0378-4754(86)90066-2

[52] H. Anzt, S. Tomov, and J. J. Dongarra, “Implementing a sparse matrix vector product for the SELL-C/SELL-C-σ formats on NVIDIA GPUs,” 2014. https://api.semanticscholar.org/CorpusID:61878276

[53] W. L. Briggs, V. E. Henson, and S. F. McCormick, *A Multigrid Tutorial.* Philadelphia: SIAM Books, 2000, second edition. https://doi.org/10.1137/1.9780898719505

[54] U. Trottenberg, C. W. Oosterlee, and A. Schüller, *Multigrid.* London: Academic Press, 2001. https://shop.elsevier.com/books/multigrid/trottenberg/978-0-08-047956-9

[55] Y. He and S. MacLachlan, “Two-level Fourier analysis of multigrid for higher-order finite-element discretizations of the Laplacian,” *Numerical Linear Algebra with Applications*, vol. 27, no. 3, p. e2285, 2020. https://doi.org/10.1002/nla.2285

[56] Y. He and S. MacLachlan, “Local Fourier analysis for mixed finite-element methods for the Stokes equations,” *Journal of Computational and Applied Mathematics*, vol. 357, pp. 161–183, 2019. https://doi.org/10.1016/j.cam.2019.01.029

[57] V. John and L. Tobiska, “A coupled multigrid method for nonconforming finite element discretizations of the 2d-Stokes equation,” *Computing*, vol. 64, no. 4, pp. 307–321, 2000, international GAMM-Workshop on Multigrid Methods (Bonn, 1998). http://doi.org/10.1007/s006070070027

[58] M. Larin and A. Reusken, “A comparative study of efficient iterative solvers for generalized Stokes equations,” *Numer. Linear Algebra Appl.*, vol. 15, no. 1, pp. 13–34, 2008. http://doi.org/10.1002/nla.561

[59] G. Alfonsi, “Reynolds-averaged Navier-Stokes equations for turbulence modeling,” *Applied Mechanics Reviews - APPL MECH REV*, vol. 62, 07 2009. https://doi.org/10.1115/1.3124648

[60] H. Tennekes and J. L. Lumley, *A First Course in Turbulence.* The MIT Press, 03 1972. https://doi.org/10.7551/mitpress/3014.001.0001

[61] D. C. Wilcox et al., *Turbulence Modeling for CFD.* DCW industries La Canada, CA, 1998, vol. 2.

[62] H. Alcin, B. Koobus, O. Allain, and A. Dervieux, “Efficiency and scalability of a two-level Schwarz algorithm for incompressible and compressible flows,” *International Journal for Numerical Methods in Fluids*, vol. 72, no. 1, pp. 69–89, 2013. https://doi.org/10.1002/fld.3733

[63] Álvaro Pé de la Riva, C. Rodrigo, F. J. Gaspar, J. H. Adler, X. Hu, and L. Zikatanov, “A local Fourier analysis for additive Schwarz smoothers,” *Computers & Mathematics with Applications*, vol. 158, pp. 13–20, 2024. https://doi.org/10.1016/j.camwa.2023.12.039

[64] R. N. King, K. Dykes, P. Graf, and P. E. Hamlington, “Optimization of wind plant layouts using an adjoint approach,” *Wind Energy Science*, vol. 2, no. 1, pp. 115–131, 2017. https://doi.org/10.5194/wes-2-115-2017

[65] M. Wasserman, Y. Mor-Yossef, I. Yavneh, and J. B. Greenberg, “Robust multigrid solution of RANS equations with two-equation turbulence models,” 2010. https://api.semanticscholar.org/CorpusID:14511099

[66] S. Baars, M. van der Klok, J. Thies, and F. W. Wubs, “A staggered-grid multilevel incomplete LU for steady incompressible flows,” *International journal for Numerical Methods in Fluids*, vol. 93, no. 4, pp. 909–926, 2021. https://doi.org/10.1002/fld.4913

[67] P. E. Farrell, L. Mitchell, and F. Wechsung, “An augmented Lagrangian preconditioner for the 3d stationary incompressible Navier-Stokes equations at high Reynolds number,” *SIAM journal on Scientific Computing*, vol. 41, no. 5, pp. A3073–A3096, 2019. https://doi.org/10.1137/18m1219370

[68] P. F. Fischer and A. G. Tomboulides, “Chapter 3 - spectral element methods for turbulence,” in *Numerical Methods in Turbulence Simulation*, ser. Numerical Methods in Turbulence, R. D. Moser, Ed. Academic Press, 2023, pp. 95–137. https://doi.org/10.1016/B978-0-32-391144-3.00009-7

[69] C. M. Klaij and C. Vuik, “SIMPLE-type preconditioners for cell-centered, colocated finite volume discretization of incompressible Reynolds-averaged Navier-Stokes equations,” *International journal for Numerical Methods in Fluids*, vol. 71, no. 7, pp. 830–849, 2013. https://doi.org/10.1002/fld.3686

[70] Segal, Rehman, and Vuik, “Preconditioners for incompressible Navier-Stokes solvers,” *Numerical Mathematics: Theory, Methods and Applications*, vol. 3, pp. 245–275, 2010. https://doi.org/10.4208/nmtma.2010.33.1

[71] Q. Chen, X. Jiao, and O. Yang, “Robust and efficient multilevel-ILU preconditioning of hybrid Newton-GMRES for incompressible Navier-Stokes equations,” *International journal for Numerical Methods in Fluids*, vol. 93, no. 12, pp. 3405–3423, Aug. 2021. https://doi.org/10.1002/fld.5039

[72] Q. Chen, A. Ghai, and X. Jiao, "HILUCSI: Simple, robust, and fast multilevel ILU for large-scale saddle-point problems from PDEs," *Numerical Linear Algebra with Applications*, vol. 28, no. 6, p. e2400, 2021. https://doi.org/10.1002/nla.2400

[73] D. Kay, D. Loghin, and A. Wathen, "A preconditioner for the steady-state Navier-Stokes equations," *SIAM journal on Scientific Computing*, vol. 24, no. 1, pp. 237–256, 2002. https://doi.org/10.1137/S106482759935808X

[74] H. Elman, V. E. Howle, J. Shadid, R. Shuttleworth, and R. Tuminaro, "Block preconditioners based on approximate commutators," *SIAM journal on Scientific Computing*, vol. 27, no. 5, pp. 1651–1668, 2006. https://doi.org/10.1137/040608817

[75] M. Benzi and M. A. Olshanskii, "An augmented Lagrangian-based approach to the Oseen problem," *SIAM journal on Scientific Computing*, vol. 28, no. 6, pp. 2095–2113, 2006. https://doi.org/10.1137/050646421

[76] S. V. Patankar and D. B. Spalding, "A calculation procedure for heat, mass and momentum transfer in three-dimensional parabolic flows," *International journal of Heat and Mass Transfer*, vol. 15, pp. 1787–1806, 1972. https://doi.org/10.1016/0017-9310(72)90054-3

[77] D. A. Knoll and W. J. Rider, "A multigrid preconditioned Newton-Krylov method," *SIAM journal on Scientific Computing*, vol. 21, no. 2, pp. 691–710, 1999. https://doi.org/10.1137/S1064827598332709

[78] E. C. Cyr, J. N. Shadid, and R. S. Tuminaro, "Stabilization and scalable block preconditioning for the Navier-Stokes equations," *Journal of Computational Physics*, vol. 231, no. 2, pp. 345–363, 2012. https://doi.org/10.1016/j.jcp.2011.09.001

[79] S. Patankar, *Numerical Heat Transfer and Fluid Flow.* CRC Press, 1980. https://doi.org/10.1201/9781482234213

[80] S. Hamilton, M. Benzi, and E. Haber, "New multigrid smoothers for the Oseen problem," *Numerical Linear Algebra with Applications*, vol. 17, no. 2-3, pp. 557–576, 2010. https://doi.org/10.1002/nla.707

[81] M. Benzi, M. A. Olshanskii, and Z. Wang, "Modified augmented Lagrangian preconditioners for the incompressible Navier-Stokes equations," *International journal for Numerical Methods in Fluids*, vol. 66, no. 4, pp. 486–508, 2011. https://doi.org/10.1002/fld.2267

[82] X. He, C. Vuik, and C. M. Klaij, "Combining the augmented Lagrangian preconditioner with the simple Schur complement approximation," *SIAM journal on Scientific Computing*, vol. 40, no. 3, pp. A1362–A1385, 2018. https://doi.org/10.1137/17M1144775

[83] M. Benzi, H. Choi, and D. B. Szyld, "Threshold ordering for preconditioning nonsymmetric problems," in *Scientific Computing, Proceedings of the Workshop, 10–12 March 1997, Hong Kong*, G. Golub, S.-H. Lui, F. Luk, and R. Plemmons, Eds. Singapore: Springer, 1997. https://www.researchgate.net/publication/2669955_Threshold_Ordering_for_Preconditioning_Nonsymmetric_Problems pp. 159–165.

[84] O. Dahl and S. O. Wille, "An ILU preconditioner with coupled node fill-in for iterative solution of the mixed finite element formulation of the 2d and 3d Navier-Stokes equations," *International journal for Numerical Methods in Fluids*, vol. 15, no. 5, pp. 525–544, 09 1992. https://doi.org/10.1002/fld.1650150503

[85] S. O. Wille and A. F. D. Loula, "A priori pivoting in solving the Navier-Stokes equations," *Communications in Numerical Methods in Engineering*, vol. 18, pp. 691–698, 2002. https://doi.org/10.1002/cnm.528

[86] S. O. Wille, Ø. Staff, and A. F. D. Loula, "Efficient a priori pivoting schemes for a sparse direct Gaussian equation solver for the mixed finite element formulation of the Navier-Stokes equations," *Applied Mathematical Modelling*, vol. 28, pp. 607–616, 2004. https://doi.org/10.1016/j.apm.2003.11.001

[87] S. W. Sloan, "An algorithm for profile and wavefront reduction of sparse matrices," *International journal for Numerical Methods in Engineering*, vol. 23, no. 2, pp. 239–251, 1986. https://doi.org/10.1002/nme.1620230208

[88] J. H. Adler, T. R. Benson, and S. P. MacLachlan, "Preconditioning a mass-conserving discontinuous Galerkin discretization of the Stokes equations," *Numerical Linear Algebra with Applications*, vol. 24, 2017. https://doi.org/10.1002/nla.2047

[89] P. E. Farrell, Y. He, and S. P. MacLachlan, "A local Fourier analysis of additive Vanka relaxation for the Stokes equations," *Numerical Linear Algebra with Applications*, vol. 28, no. 3, p. e2306, 2021. https://doi.org/10.1002/nla.2306

[90] P. E. Farrell, M. G. Knepley, F. Wechsung, and L. Mitchell, "PCPATCH: software for the topological construction of multigrid relaxation methods," *ACM Trans. Math. Softw.*, vol. 47, pp. 25:1–25:22, 2019. https://doi.org/10.1145/3445791

[91] X.-C. Cai and M. Sarkis, "A Restricted additive Schwarz preconditioner for general sparse linear systems," *SIAM Journal on Scientific Computing*, vol. 21, no. 2, pp. 792–797, 1999. https://doi.org/10.1137/S106482759732678X

[92] Z. Li and Y. Saad, "SchurRAS: A restricted version of the overlapping Schur complement preconditioner," *SIAM Journal on Scientific Computing*, vol. 27, no. 5, pp. 1787–1801, 2006. https://doi.org/10.1137/040608350

[93] H. Liu, Z. Chen, S. Yu, B. Hsieh, and L. Shao, "Development of a Restricted additive Schwarz preconditioner for sparse linear systems on NVIDIA GPU," 2014. https://api.semanticscholar.org/CorpusID:59388878

[94] B. Yang, H. Liu, Z. Chen, and X. Tian, "GPU-accelerated preconditioned GMRES solver," in *2016 IEEE 2nd International Conference on Big Data Security on Cloud (BigDataSecurity), IEEE International Conference on High Performance and Smart Computing (HPSC), and IEEE International Conference on Intelligent Data and Security (IDS)*, 2016. https://doi.org/10.1109/BigDataSecurity-HPSC-IDS.2016.28 pp. 280–285.

[95] S. Saberi, G. Meschke, and A. Vogel, "A restricted additive Vanka smoother for geometric multigrid," *Journal of Computational Physics*, vol. 459, p. 111123, 2022. https://doi.org/10.1016/j.jcp.2022.111123

[96] R. Ram, D. Grünewald, and N. R. Gauger, "Scalable hybrid parallel ILU preconditioner to solve sparse linear systems," in *Euro-Par 2021: Parallel Processing Workshops*, R. Chaves, D. B. Heras, A. Ilic, D. Unat, R. M. Badia, A. Bracciali, P. Diehl, A. Dubey, O. Sangyoon, S. L. Scott, and L. Ricci, Eds. Cham: Springer International Publishing, 2022. https://doi.org/10.1007/978-3-031-06156-1_46 pp. 540–544.

[97] R. D. Falgout and U. M. Yang, "hypre: A library of high performance preconditioners," in *Computational Science — ICCS 2002*, P. M. A. Sloot, A. G. Hoekstra, C. J. K. Tan, and J. J. Dongarra, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2002. https://doi.org/10.1007/3-540-47789-6_66 pp. 632–641.

[98] "*hypre*: High performance preconditioners," https://llnl.gov/casc/hypre, https://github.com/hypre-space/hypre.

[99] S. Balay, S. Abhyankar, M. F. Adams, S. Benson, J. Brown, P. Brune, K. Buschelman, E. M. Constantinescu, L. Dalcin, A. Dener, V. Eijkhout, J. Faibussowitsch, W. D. Gropp, V. Hapla, T. Isaac, P. Jolivet, D. Karpeev, D. Kaushik, M. G. Knepley, F. Kong, S. Kruger, D. A. May, L. C. McInnes, R. T. Mills, L. Mitchell, T. Munson, J. E. Roman, K. Rupp, P. Sanan, J. Sarich, B. F. Smith, S. Zampini, H. Zhang, H. Zhang, and J. Zhang, "PETSc web page," https://petsc.org/, 2023. https://petsc.org/

[100] S. Balay, S. Abhyankar, M. F. Adams, S. Benson, J. Brown, P. Brune, K. Buschelman, E. Constantinescu, L. Dalcin, A. Dener, V. Eijkhout, J. Faibussowitsch, W. D. Gropp, V. Hapla, T. Isaac, P. Jolivet, D. Karpeev, D. Kaushik, M. G. Knepley, F. Kong, S. Kruger, D. A. May, L. C. McInnes, R. T. Mills, L. Mitchell, T. Munson, J. E. Roman, K. Rupp, P. Sanan, J. Sarich, B. F. Smith, S. Zampini, H. Zhang, H. Zhang, and J. Zhang, "PETSc/TAO users manual," Argonne National Laboratory, Tech. Rep. ANL-21/39 - Revision 3.19, 2023.

[101] S. Balay, W. D. Gropp, L. C. McInnes, and B. F. Smith, "Efficient management of parallelism in object oriented numerical software libraries," in *Modern Software Tools in Scientific Computing*, E. Arge, A. M. Bruaset, and H. P. Langtangen, Eds. Birkhäuser Press, 1997. https://doi.org/10.1007/978-1-4612-1986-6_8 pp. 163–202.

[102] D. A. Ham, P. H. J. Kelly, L. Mitchell, C. J. Cotter, R. C. Kirby, K. Sagiyama, N. Bouziani, S. Vorderwuelbecke, T. J. Gregory, J. Betteridge, D. R. Shapero, R. W. Nixon-Hill, C. J. Ward, P. E. Farrell, P. D. Brubeck, I. Marsden, T. H. Gibson, M. Homolya, T. Sun, A. T. T. McRae, F. Luporini, A. Gregory, M. Lange, S. W. Funke, F. Rathgeber, G.-T. Bercea, and G. R. Markall, *Firedrake User Manual*, 1st ed., Imperial College London and University of Oxford and Baylor University and University of Washington, 5 2023. https://doi.org/10.25561/104839

[103] "The Firedrake project," https://www.firedrakeproject.org, accessed: 2024-01-16.

[104] M. D. Gunzburger and J. S. Peterson, "On conforming finite element methods for the inhomogeneous stationary Navier-Stokes equations," *Numer. Math.*, vol. 42, no. 2, pp. 173–194, jun 1983. https://doi.org/10.1007/BF01395310

[105] "Pull request implementing interface to hypre's ILU," https://gitlab.com/petsc/petsc/-/merge_requests/7458, accessed: 2024-05-13.

[106] "Multifrontal massively parallel soarse direct solver (MUMPS)," https://mumps-solver.org/, accessed: 2024-02-06.

[107] "Speedscope, an interactive flamegraph visualizer," https://www.speedscope.app/, accessed: 2024-05-30.

[108] "DELTA supercomputer homepage," https://delta.ncsa.illinois.edu/, accessed: 2023-03-21.

[109] T. Deakin, J. Price, M. Martineau, and S. McIntosh-Smith, "Evaluating attainable memory bandwidth of parallel programming models via BabelStream," *Int. J. Comput. Sci. Eng.*, vol. 17, pp. 247–262, 2018. https://api.semanticscholar.org/CorpusID:67046090

[110] H. B. Keller, "Continuation methods in computational fluid dynamics," in *Numerical and Physical Aspects of Aerodynamic Flows*, T. Cebeci, Ed. Berlin, Heidelberg: Springer Berlin Heidelberg, 1982. https://doi.org/10.1007/978-3-662-12610-3_1 pp. 3–13.

[111] M. D. Gunzburger, "8 - solution methods for large Reynolds numbers," in *Finite Element Methods for Viscous Incompressible Flows*, ser. Computer Science and Scientific Computing, M. D. Gunzburger, Ed. San Diego: Academic Press, 1989, pp. 105–114. https://doi.org/10.1016/B978-0-12-307350-1.50015-6

[112] G. Carey and R. Krishnan, "Continuation techniques for a penalty approximation of the Navier-Stokes equations," *Computer methods in applied mechanics and engineering*, vol. 48, pp. 265–282, 1985. https://doi.org/10.1016/s0045-7825(85)80002-5

[113] M. D. Gunzburger and J. S. Peterson, "Predictor and steplength selection in continuation methods for the Navier-Stokes equations," *Computers & Mathematics with Applications*, vol. 22, no. 8, pp. 73–81, 1991. https://doi.org/10.1016/0898-1221(91)90015-v

[114] D. A. Brown and D. W. Zingg, "A monolithic homotopy continuation algorithm with application to computational fluid dynamics," *Journal of Computational Physics*, vol. 321, pp. 55–75, 2016. https://doi.org/10.1016/j.jcp.2016.05.031

[115] J. Malm, P. Schlatter, P. F. Fischer, and D. S. Henningson, "Stabilization of the spectral element method in convection dominated flows by recovery of skew-symmetry," *J. Sci. Comput.*, vol. 57, no. 2, pp. 254–277, nov 2013. https://doi.org/10.1007/s10915-013-9704-1

[116] R. Swanson and E. Turkel, "Pseudo-time algorithms for the Navier-Stokes equations," *Applied Numerical Mathematics*, vol. 2, no. 3, pp. 321–333, 1986, special Issue in Honor of Milt Rose's Sixtieth Birthday. https://doi.org/10.1016/0168-9274(86)90037-1

[117] A. Chorin, "Numerical solution of the Navier-Stokes equations," *Mathematics of Computation*, vol. 22, 10 1968. https://doi.org/10.2307/2004575

[118] R. Temam and A. Chorin, "Navier-Stokes equations: Theory and numerical analysis," *Journal of Applied Mechanics*, vol. 45, no. 2, pp. 456–456, 06 1978. https://doi.org/10.1115/1.3424338

[119] A. Sharma, S. Ananthan, J. Sitaraman, S. Thomas, and M. A. Sprague, "Overset meshes for incompressible flows: On preserving accuracy of underlying discretizations," *Journal of Computational Physics*, vol. 428, p. 109987, 2021. https://doi.org/10.1016/j.jcp.2020.109987

APPENDIX A: DYNAMIC SOLVER IMPLEMENTATION IN FIREDRAKE

The test code used in chapter 4 to evaluate the proposed solver design is given in two files below. The solver is controlled by the content of `main.py` given in appendix A with the problem set up done in the file `problems.py` given in appendix A

```python
from firedrake import *
from petsc4py import PETSc
import argparse, sys, os, time, pathlib
from problemsetup import *
PETSc.Sys.popErrorHandler()

####################################################
####################################################

# parse command line arguments

parser = argparse.ArgumentParser(description="Solver", formatter_class=argparse.
    ArgumentDefaultsHelpFormatter)

parser.add_argument("-Re", type=int, dest="reynolds", default=10,
                    help="Target Reynolds number")
parser.add_argument("-file", type=str, dest="file", default="temp.log",
                    help="File where outout of script is piped to")
parser.add_argument("-logfile", type=str, dest="logfile", default="output.log",
                    help="File where to log the progress")

parser.add_argument("-nx", type=int, dest="nx", default=30,
                    help="dof's - x direction")
parser.add_argument("-ny", type=int, dest="ny", default=30,
                    help="dof's - y direction")
parser.add_argument("-nz", type=int, dest="nz", default=30,
                    help="dof's - z direction")

args, unknown = parser.parse_known_args(sys.argv[1:])

####################################################
####################################################

# limit the quadrature degree
parameters['form_compiler']['quadrature_degree'] = 6

####################################################
####################################################

if not os.path.exists(args.file):
    PETSc.Sys.Print("The file flag is not set to the current piped output. Please fix
    that first")
    sys.exit(1)

```

Listing A.1: File for controlling solver - main.py.

```
# print configuration to screen

PETSc.Sys.Print("")
PETSc.Sys.Print("************************************************")
PETSc.Sys.Print("")
PETSc.Sys.Print("** CONFIGURATION ** ")
PETSc.Sys.Print("")
PETSc.Sys.Print(f"Reynold's number: {args.reynolds}")
PETSc.Sys.Print(f"            Mesh: {args.nx}x{args.ny}x{args.nz}")
PETSc.Sys.Print(f"     # MPI ranks: {COMM_WORLD.Get_size()}")

PETSc.Sys.Print("")


logf = open(args.logfile, 'w')
logf.write("\n")
logf.write("----------------------------------------------\n")
logf.write("\n")
logf.write(f"** target Reynolds number = {args.reynolds}\n")
logf.write(f"** mesh = {args.nx}x{args.ny}x{args.nz}\n")
logf.write(f"** # MPI ranks = {COMM_WORLD.Get_size()}\n")
logf.write("\n")
logf.write("----------------------------------------------\n")
logf.write("\n")
logf.flush()

###################################################
###################################################

par_damp = 1
par_Re = min(args.reynolds, 100)
par_dR = 150
par_tun = 0

par_tau_step = 1e-4
par_tau_final = 1e-5
par_tau = par_tau_step if par_Re < args.reynolds else par_tau_final

_liniter = 10
_maxnonliniter = 100
_stol = 1e-4

###################################################
###################################################

M, Z, w, v, q = CreateRANSProblem(args.nx, args.ny, args.nz)

PETSc.Sys.Print(f"dim = {Z.dim()}")
PETSc.Sys.Print("")

solutionFound = False
while not solutionFound:
```

Listing A.1 (cont.)

```

    PETSc.Sys.Print("************************************************")
    PETSc.Sys.Print("")
    PETSc.Sys.Print(f" >> RUNNING SOLVER: Re = {par_Re}, par_damp = {par_damp}, par_tau
    = {par_tau}, current dR = {par_dR}")
    PETSc.Sys.Print("")

    logf.write("************************************************\n")
    logf.write(f">> Re = {par_Re}, par_damp = {par_damp}, par_tau = {par_tau}, current
    dR = {par_dR}\n")
    logf.flush()

    F, bcs, nsp = RANS3D(par_Re, Z, M, w, v, q)

    # some information about system
    appctx = {"Re": par_Re, "velocity_space": 0}

    parameters = {

            # aij allows to extract linearized matrices
            "mat_type": "aij",

            # some SNES options
            "snes_monitor": None,
            "snes_rtol": par_tau,
            "snes_atol": par_tau,
            "snes_stol": _stol,
            "snes_max_it": _maxnonliniter,
            "snes_converged_reason": None,
            "snes_view": None,

            # linearize with damped Newton
            "snes_type": "newtonls",
            "snes_linesearch_damping": par_damp,

            # fGMRES solver around fieldsplit
            "ksp_type": "fgmres",
            "ksp_convergence_test": "skip",
            "ksp_gmres_modifiedgramschmidt": None,
            "ksp_monitor_true_residual": None,
            "ksp_converged_reason": None,
            "ksp_rtol": 1e-10,
            "ksp_atol": 1e-10,
            "ksp_max_it": _liniter,

            # fieldsplit as preconditioner
            "pc_type": "fieldsplit",
            "pc_fieldsplit_type": "schur",
            "pc_fieldsplit_schur_fact_type": "full",
            # this ensures non-singular Schur-complement
            "pc_fieldsplit_schur_precondition": "selfp",

```

Listing A.1 (cont.)

```
        # velocity solve (AMG+RAS/ILU)
        "fieldsplit_0_ksp_type": "gmres",
        "fieldsplit_0_ksp_convergence_test": "skip",
        "fieldsplit_0_ksp_gmres_modifiedgramschmidt": None,
        #"fieldsplit_0_ksp_monitor_true_residual": None,
        "fieldsplit_0_ksp_rtol": 1e-2,
        "fieldsplit_0_ksp_atol": 1e-10,
        "fieldsplit_0_ksp_max_it": _liniter,
        #"fieldsplit_0_ksp_view": None,

        "fieldsplit_0_pc_type": "hypre",
        "fieldsplit_0_pc_hypre_type": "boomeramg",
        "fieldsplit_0_pc_hypre_boomeramg_tol": 1e-2,
        #"fieldsplit_0_pc_hypre_boomeramg_print_statistics": None,
        "fieldsplit_0_pc_hypre_boomeramg_ilu_max_nnz_per_row": 100,
        "fieldsplit_0_pc_hypre_boomeramg_strong_threshold": 0.5,
        "fieldsplit_0_pc_hypre_boomeramg_truncfactor": 0.3,
        "fieldsplit_0_pc_hypre_boomeramg_max_iter": 1,
        "fieldsplit_0_pc_hypre_boomeramg_smooth_type": "ILU",
        "fieldsplit_0_pc_hypre_boomeramg_ilu_type": "RAS-ILUk",
        "fieldsplit_0_pc_hypre_boomeramg_ilu_iterative_setup": "async-in-place",
        "fieldsplit_0_pc_hypre_boomeramg_ilu_level": 0,
        "fieldsplit_0_pc_hypre_boomeramg_ilu_tri_solve": 0,
        "fieldsplit_0_pc_hypre_boomeramg_ilu_local_reordering": 0,
        "fieldsplit_0_pc_hypre_boomeramg_smooth_num_sweeps": 1,
        "fieldsplit_0_pc_hypre_boomeramg_smooth_num_levels": 25,
        "fieldsplit_0_pc_hypre_boomeramg_max_levels": 5,
        "fieldsplit_0_pc_hypre_boomeramg_cycle_type": "v",
        "fieldsplit_0_pc_hypre_boomeramg_coarsen_type": "PMIS",
        "fieldsplit_0_pc_hypre_boomeramg_interp_type": "ext+i",

        # Schur-complement solve (AMG+RAS/ILU)
        "fieldsplit_1_ksp_type": "gmres",
        "fieldsplit_1_ksp_convergence_test": "skip",
        "fieldsplit_1_ksp_gmres_modifiedgramschmidt": None,
        #"fieldsplit_1_ksp_monitor_true_residual": None,
        "fieldsplit_1_ksp_rtol": 1e-2,
        "fieldsplit_1_ksp_atol": 1e-10,
        "fieldsplit_1_ksp_max_it": _liniter,

        # "fieldsplit_1_ksp_type": "preonly",
        "fieldsplit_1_pc_type": "hypre",
        "fieldsplit_1_pc_hypre_type": "boomeramg",
        "fieldsplit_1_pc_hypre_boomeramg_tol": 1e-2,
        #"fieldsplit_1_pc_hypre_boomeramg_print_statistics": None,
        "fieldsplit_1_pc_hypre_boomeramg_ilu_max_nnz_per_row": 100,
        "fieldsplit_1_pc_hypre_boomeramg_strong_threshold": 0.5,
        "fieldsplit_1_pc_hypre_boomeramg_truncfactor": 0.3,
        "fieldsplit_1_pc_hypre_boomeramg_max_iter": 1,
        "fieldsplit_1_pc_hypre_boomeramg_smooth_type": "ILU",
        "fieldsplit_1_pc_hypre_boomeramg_ilu_type": "RAS-ILUk",
        "fieldsplit_1_pc_hypre_boomeramg_ilu_iterative_setup": "async-in-place",
```

Listing A.1 (cont.)

```
            "fieldsplit_1_pc_hypre_boomeramg_ilu_level": 0,
            "fieldsplit_1_pc_hypre_boomeramg_ilu_tri_solve": 0,
            "fieldsplit_1_pc_hypre_boomeramg_ilu_local_reordering": 0,
            "fieldsplit_1_pc_hypre_boomeramg_smooth_num_sweeps": 1,
            "fieldsplit_1_pc_hypre_boomeramg_smooth_num_levels": 25,
            "fieldsplit_1_pc_hypre_boomeramg_max_levels": 5,
            "fieldsplit_1_pc_hypre_boomeramg_cycle_type": "v",
            "fieldsplit_1_pc_hypre_boomeramg_coarsen_type": "PMIS",
            "fieldsplit_1_pc_hypre_boomeramg_interp_type": "ext+i",

        }

    #w_new = w.copy()

    # set up nonlinear variational problem and solver
    #nvproblem = NonlinearVariationalProblem(F, w_new, bcs=bcs)
    nvproblem = NonlinearVariationalProblem(F, w, bcs=bcs)
    solver = NonlinearVariationalSolver(nvproblem, solver_parameters=parameters,
    nullspace=nsp, appctx=appctx)

    t1 = time.time_ns()

    solver.solve()

    t2 = time.time_ns()

    PETSc.Sys.Print("")
    PETSc.Sys.Print("************************************************")
    PETSc.Sys.Print("")
    PETSc.Sys.Print(f" >> SOLVE TIME: {(t2-t1)//1e6} ms")
    PETSc.Sys.Print("")
    PETSc.Sys.Print("************************************************")
    PETSc.Sys.Print("")

    f = open(args.file, "r")
    txt = f.read()

    lines = txt.split(" 0 SNES Function norm ")
    initres    = float(lines[-1].split("\n")[0])

    history = [initres]
    allparts = lines[-1].split(" SNES Function norm ")
    for p in allparts:
        history.append(float(p.split("\n")[0]))

    finalres   = history[-1]
    itercount = len(history)-1
    itercount_str = str(itercount)

    logf.write(">> Convergence history:\n")
    for h in history:
        logf.write(f">> {h}\n")
```

Listing A.1 (cont.)

```
    logf.write(">> End convergence history\n")
    logf.write(f">> # iterations: {itercount_str}\n\n\n")
    logf.flush()

    improvement = finalres/initres

    # CONVERGED
    if improvement <= par_tau or finalres <= par_tau or abs(improvement-1) < 1e-12:

        if par_Re == args.reynolds:

            solutionFound = True
            logf.write("\n>> SOLUTION FOUND!\n\n")
            break

        else:

            #w = w_new.copy()
            par_damp = 1.0
            par_tun = 0

            # too few iteration
            if itercount < 1:

                par_Re -= par_dR
                par_dR = min(args.reynolds-par_dR, par_dR+50)

            # solid solve
            else:

                par_dR = 150

    # NOT CONVERGED
    else:

        # reset Re
        if par_dR < par_Re:
            par_Re = par_Re-par_dR

        if par_tun == 0:

            par_dR = max(25, par_dR-50)
            par_damp = 1
            par_tun = 1

        elif par_tun == 1:

            par_dR = max(25, par_dR-50)
            par_damp = 1
            par_tun = 2

        elif par_tun == 2:
```

Listing A.1 (cont.)

```

            par_damp = 0.9
            par_tun = 3

        elif par_tun == 3:

            par_dR = max(25, par_dR-25)
            par_damp = 1
            par_tun = 4

        elif par_tun == 4:

            par_damp = 0.9
            par_tun = 5

        elif par_tun == 5:

            PETSc.Sys.Print(f"NO SOLUTION FOUND")
            logf.write("\n>> SOLVE FAILED!!\n\n")
            solutionFound = False
            break

    par_Re = min(args.reynolds, par_Re + par_dR)

    if par_Re == args.reynolds:
        par_tau = par_tau_final
    else:
        par_tau = par_tau_step


logf.close()

####################################################
####################################################

# Output solution

u, p = w.subfunctions
u.rename("Velocity")
p.rename("Pressure")
File("solution.pvd").write(u, p)

```

Listing A.1 (cont.)

```python
from firedrake import *
from scipy.special import gamma
from firedrake.__future__ import interpolate

def CreateRANSProblem(nx, ny, nz):

    # box mesh
    # M = BoxMesh(nx, ny, nz, 6, 2, 2,  hexahedral=True, reorder=False)
    M = UnitCubeMesh(nx, ny, nz,  hexahedral=True, reorder=False)

    # Q2/Q1 elements
    Q1 = VectorFunctionSpace(M, "Q", 2)
    Q2 = FunctionSpace(M, "Q", 1)
    Z = Q1 * Q2

    # create variables
    up = Function(Z)
    v, q = TestFunctions(Z)

    up.assign(0)

    return M, Z, up, v, q

def RANS3D(Re, Z, M, up, v, q):

    def build_turbine_force(x0, u, rotor_diam=130, yaw=0):

        # x0[0]: x-location of turbine
        # x0[1]: y-location of turbine
        # x0[2]: hub height of turbine (distance above ground level)
        assert isinstance(x0, list)
        assert len(x0) == 3

        # Define useful turbine values
        W = 0.1*rotor_diam # thickness across rotor plane
        R = 0.5*rotor_diam # rotor radius
        ma = 0.3 # axial induction factor
        C_tprime = 4.0*ma/(1.0-ma) # modified thrust coefficient

        x = SpatialCoordinate(M)

        # Set up some dim dependent values
        S_norm = (2.0+pi)/(2.0*pi)
        T_norm = 2.0*gamma(7.0/6.0)

        WTGbase = as_vector((cos(yaw), sin(yaw), 0.0))
        A = pi*R**2.0
        D_norm = pi*gamma(4.0/3.0)

        # Rotate and Shift the Turbine
        xrot =  cos(yaw)*(x[0]-x0[0]) + sin(yaw)*(x[1]-x0[1])
        yrot = -sin(yaw)*(x[0]-x0[0]) + cos(yaw)*(x[1]-x0[1])
```

Listing A.2: File for creating model problem setup - problemsetup.py

```
        zrot = x[2]-x0[2]
        xs = [xrot,yrot,zrot]

        # Create the function that represents the Thickness of the turbine
        T = exp(-pow((xs[0]/W), 6.0))

        # Create the function that represents the Disk of the turbine
        r = sqrt(xs[1]**2.0+xs[2]**2.0)/R
        D = exp(-pow(r, 6.0))

        # Create the function that represents the force
        force_profile = "sine"

        if force_profile == "constant":
            force = 1.0
        elif force_profile == "sine":
            force = (r*sin(pi*r)+0.5)/S_norm
        else:
            raise ValueError(f"Force profile {force_profile} not recognized.")

        F = -0.5*A*C_tprime*force

        # Calculate normalization constant
        volNormalization = T_norm*D_norm*W*R**2.0

        # compute disk averaged velocity in yawed case and don't project
        actuator_disk = F*T*D*WTGbase/volNormalization

        # Expand the dot product
        tf1 = actuator_disk * cos(yaw)**2
        tf2 = actuator_disk * sin(yaw)**2
        tf3 = actuator_disk * 2.0 * cos(yaw) * sin(yaw)

        # Compose full turbine force
        tf = tf1*u[0]**2+tf2*u[1]**2+tf3*u[0]*u[1]

        return tf

    # create variables
    u, p = split(up)

    # Calculate the turbine force
    tf = build_turbine_force([0.25, 0.5, 0.5], u, rotor_diam=0.5, yaw=0)
    turb_force = inner(tf, v)*dx

    F = (

        # Viscosity: nu*grad^2(u)
        (1.0 / Re) * inner(grad(u), grad(v)) * dx +

        # Convection: u*grad(u)
        inner(dot(u, nabla_grad(u)), v) * dx +
```

Listing A.2 (cont.)

```

        # Pressure: -grad(P)
        inner(grad(p), v)*dx -

        # Divergence: div(u)
        inner(div(u), q)*dx -

        # wind turbine
        turb_force
    )

    x = SpatialCoordinate(M)
    inflow = assemble(interpolate(as_vector([1.3*pow(x[2]/2, 0.5), 0.0, 0.0]), Z.sub(0))
    )
    bcs = [DirichletBC(Z.sub(0), inflow, (1,)),                              # left
           DirichletBC(Z.sub(0).sub(1), Constant((0)), (3,)),               # front
           DirichletBC(Z.sub(0).sub(1), Constant((0)), (4,)),               # back
           DirichletBC(Z.sub(0), Constant((0,0,0)),    (5,)),               # bottom
           DirichletBC(Z.sub(0).sub(2), Constant((0)), (6,)),               # top
           # DirichletBC(Z.sub(1), Constant((0)), (1,2,3,4,5,6))
        ]

    # create nullspace on velocity space
    nullspace = MixedVectorSpaceBasis(Z, [Z.sub(0), VectorSpaceBasis(constant=True, comm
    =COMM_WORLD)])

    return (F, bcs, nullspace)

```

Listing A.2 (cont.)