Performance analysis of POT3D with PETSc preconditioners and solvers

Jaffery Irudayasamy
Student Number: S2336648

August 19, 2023

MSc in High Performance Computing with Data Science

The University of Edinburgh

Year of Presentation: 2023

Abstract

This dissertation presents the research conducted as part of the annual ISC Student Cluster Competition (SCC) and an extended project focused on the performance analysis of POT3D with PETSc preconditioners and solvers. The SCC involved student teams designing and optimizing computer clusters to achieve peak performance with supplied programs. Our team from EPCC participated in the 2023 onsite competition, tuning two programs for the cluster: the GPU-based benchmark HPCG and the CPU and GPU-based distributed-memory program POT3D, which served as the potential field solver.

Expanding on the work done with POT3D for the SCC, we proceeded to implement two versions of the POT3D application with CPU-only and GPU-based PETSc Krylov subspace solvers. The primary objective was to analyze the performance of standard preconditioners/solvers from the PETSc library in comparison to different preconditioners, including the conjugate gradient solver implemented in POT3D. The resulting code was benchmarked on Cirrus, yielding valuable insights into the performance benefits and limitations of PETSc solvers and the POT3D solver concerning boundary conditions and matrix formation. Our findings shed light on the potential of PETSc solvers to enhance the POT3D application, contributing to the field of computational science and engineering.

Contents

List of Tables

List of Figures

List of Code Snippets

Acknowledgements

I am immensely grateful to Xu Guo and Spyro Nita for their exceptional leadership and guidance as part of TeamEPCC during the Student Cluster Competition. Their expertise and support were instrumental in our successful journey.

My sincere appreciation goes to my supervisor, Dr. William Lucas, whose excellent and frequent guidance proved invaluable throughout this project. His expertise in numerical algorithms and Fortran provided the necessary foundation and support, especially in areas where I had limited prior experience.

I extend heartfelt thanks to my parents for their unwavering support in various forms. Their encouragement and belief in my abilities have been a constant source of motivation.

Lastly, I would like to express my gratitude to Ben Morse and the entire EPCC staff for creating an enjoyable and enriching environment. Their contributions made this year's experience truly memorable.

The collective support and encouragement from these individuals have played a crucial role in the successful completion of this project, and I am deeply thankful for their contributions.

Chapter 1

Introduction

The fundamental representation of the coronal magnetic field is established through a photospheric boundary radial magnetic field, forming the basis of a potential (current-free) field model [1]. This model plays a pivotal role in comprehending diverse solar and heliospheric phenomena, encompassing the interplanetary magnetic field [3], coronal heating and X-ray emissions [4], and the topology of the coronal magnetic field [5].

One essential tool used to estimate the magnetic field of the solar corona is POT3D, a Fortran code that utilizes photospheric magnetic field data obtained from the National Solar Observatory's Global Oscillation Network Group [6] and the Helioseismic Magnetic Imager (HMI) aboard the Solar Dynamics Observatory [7] as a boundary condition. POT3D computes Potential Field (PF) solutions to approximate the magnetic field of the solar corona. It has been extensively employed in various studies to investigate the structure and dynamics of the corona. Notably, POT3D serves as the potential field solver for the Wang–Sheeley–Arge (WSA)[8] model within NASA's Community Coordinated Modeling Center (CCMC) CORHEL software suite [9].

POT3D utilizes a specialized iterative conjugate gradient solver for estimating potential field solutions. To accelerate solver convergence, preconditioners are utilized to reduce the number of iterations needed to reach a solution [2]. The code is designed for efficient parallelization using the Message Passing Interface (MPI) and adopts a three-dimensional decomposition method. Furthermore, it takes advantage of GPU acceleration through OpenACC and leverages an external cuSPARSE library, making it highly suitable for computationally intensive tasks [1].

This dissertation project extends from the work carried out for the ISC23 Student Cluster Competition [10], an annual event where student teams compete to design high-performance computing clusters. POT3D was one of the applications used in this year's competition, laying the groundwork for further exploration in the extended project. The focus of this extension involves investigating the performance and extensibility of standard solver libraries like PETSc [11] compared to the custom solver implemented in POT3D, specifically for both CPU-only and GPU systems. By conducting this analysis, we aim to gain insights into the potential of PETSc preconditioners and solvers [12] within the context of POT3D's computational requirements.

Chapter 2

Background and Literature review

2.1 POT3D: Model description

Potential Fields (PFs) are magnetic fields without electric currents, resulting in zero forces [1]. Setting the current to zero in Maxwell's equation yields magnetic field B, which is a gradient of scalar potential Φ. The divergence-free condition for magnetic field B is a manifestation of the absence of magnetic monopoles in nature. This condition ensures that magnetic field lines form closed loops, precluding the existence of isolated magnetic sources or sinks. This distinct behaviour leads to the Laplace equation (2.1) governing the magnetic scalar potential Φ, which can be solved under appropriate boundary conditions.

$$\nabla^2\Phi = 0 \tag{2.1}$$

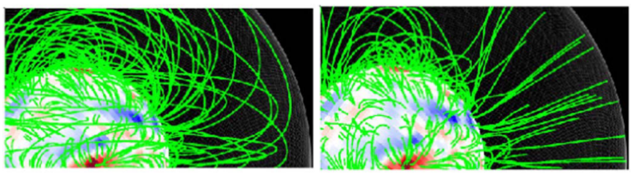


Figure 2.1: Boundary conditions: (Left) require the lines to close within the outer boundary. (Right) lines can extend beyond. reference [1].

The lower boundary condition at the solar surface is determined based on observational data of the radial magnetic field component B_r obtained from the National Solar Observatory's Global Oscillation Network Group [6] and the Helioseismic Magnetic Imager (HMI) aboard the Solar Dynamics Observatory [7]. The upper boundary condition varies depending on the specific application. For a 'closed wall' scenario, the magnetic field lines must close within the outer boundary. Alternatively, for a 'source surface' scenario, the magnetic field lines are allowed to extend beyond it (as depicted in Figure 2.1). Once the Laplace equation is solved, the magnetic field B can be obtained by computing its gradient, i.e., $B = \nabla\Phi$.

2.2 Numerical technique in POT3D

The finite difference method is a numerical technique to approximate solutions to differential equations. It works by discretizing the solution domain and approximating derivatives using the differences between neighbouring grid points. In the context of POT3D, this method approximates the solution to the Laplace equation (2.1). The equation is elliptical, which means it is not time-dependent, so the program can handle the geometric changes in the grid as it approaches the poles without too much difficulty. However, this does complicate the matrix system's solution, necessitating additional iterations. To address this challenge, POT3D employs a second-order finite-difference method as described in reference [13], wherein equation (2.1) takes the following form.

$$
\begin{aligned}
\nabla^2\Phi_{i,j,k} \approx\ & \frac{1}{\Delta r_i}\left[\frac{\Phi_{i+1,j,k}-\Phi_{i,j,k}}{\Delta r_{i+\frac{1}{2}}}-\frac{\Phi_{i,j,k}-\Phi_{i-1,j,k}}{\Delta r_{i-\frac{1}{2}}}\right] \\
&+\frac{1}{sin\theta_j\Delta\theta_j}\left[sin\theta_{i,j+\frac{1}{2}}\frac{\Phi_{i,j+1,k}-\Phi_{i,j,k}}{\Delta\theta_{j+\frac{1}{2}}}-sin\theta_{i,j-\frac{1}{2}}\frac{\Phi_{i,j,k}-\Phi_{i,j-1,k}}{\Delta\theta_{j-\frac{1}{2}}}\right] \\
&+\frac{1}{sin^2\theta_j\Delta\phi_k}\left[\frac{\Phi_{i,j,k+1}-\Phi_{i,j,k}}{\Delta\phi_{k+\frac{1}{2}}}-\frac{\Phi_{i,j,k}-\Phi_{i,j,k-1}}{\Delta\phi_{k-\frac{1}{2}}}\right]=0
\end{aligned}
\tag{2.2}
$$

where i, j, and k correspond to the Cartesian coordinates x, y, and z, respectively, of a point in a three-dimensional grid. The parameters Δr_i, $\Delta\theta_j$, and $\Delta\phi_k$ represent the spacings between adjacent grid points along the x, y, and z directions, respectively. It's important to note that although the grid is indexed by Cartesian coordinates, it can be visualized as a 3D cube grid of points, and when these points are converted to spherical polar coordinates, they form a sphere. Equation (2.2) can be represented in matrix form as

$$A\Phi = 0 \tag{2.3}$$

where A is a sparse matrix that represents the coefficients of the finite difference equation (2.2) in matrix form (2.3). It contains the finite difference coefficients corresponding to the Laplace operator applied to the scalar field Φ in three-dimensional spherical coordinates. The elements of matrix A are determined by the coefficients of the finite difference terms involving neighbouring grid points in the original equation. Matrix A is stored in a custom diagonal

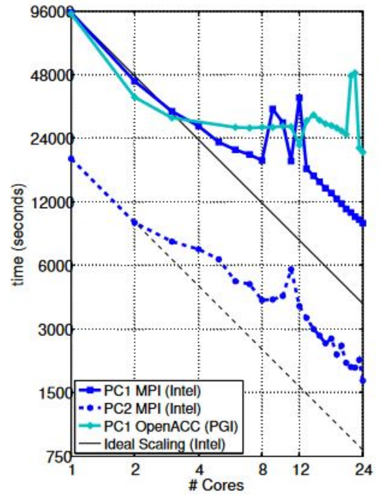
(DIA) sparse format [14] for the inner grid points, while the boundary conditions are implemented in a matrix-free manner as defined by equation (2.2).

The Preconditioned Conjugate Gradient (PCG) method serves as an iterative numerical technique for solving linear equations [15]. In the specific context of the POT3D application, it addresses an equation represented by (2.3). This method strategically employs preconditioners to enhance convergence during solving. To maximize efficiency, two distinct preconditioners (PCs) are applied. The first preconditioner (PC1) involves diagonal scaling or point-Jacobi preconditioning [16]. While computationally efficient, it may offer comparatively moderate effectiveness. PC1 exploits the matrix's diagonal elements to transform the linear system, thus aiming to improve convergence in the iterative solving process. On the other hand, the second preconditioner (PC2) employs ILU0, a technique enhancing convergence by approximating the matrix with lower and upper triangular factors [17]. PC2 is harnessed within a nonoverlapping domain decomposition method, designed to expedite convergence. However, it comes with heightened computational costs. Generally, PC2 is more potent, although there are scenarios where PC1 might yield better results.

Convergence of the solver is gauged by comparing the norm of the preconditioned residual to the norm of the preconditioned right-hand side, utilizing a predefined tolerance. In this instance, the specified tolerance is set at 10^{-9}. This tolerance value is an adjustable parameter within the POT3D application, offering control over the solver's behaviour.

2.3 Preconditioners in POT3D

The implementation details of the preconditioners used in POT3D are discussed in reference [2]. In the POT3D application, PC1 (diagonal scaling / point-Jacobi) is implemented with the support for OpenACC acceleration. The PC2 (ILU0) preconditioner presents a challenge to vectorization, which in turn impacts the efficient computation on GPUs using OpenACC pragmas. This challenge arises from the requirement to solve a set of sequential sparse matrix equations with a triangular structure, as part of applying the PC2 preconditioning. Achieving vectorized solutions for these triangular matrix equations is notably complex. To address this challenge, NVIDIA's cuSPARSE library [18] was utilized, which offers two distinct algorithms for solving these equations. For practical implementation, utilizing the first algorithm was the focus. This choice was driven by the fact that the performance difference between the two algorithms was found to be negligible. The cuSPARSE library is written in C, but to allow Fortran users to use the library directly, Fortran wrappers are provided. However, to minimize the amount of added code to POT3D, these wrappers were not used. Instead, a small C function call to the cuSPARSE library was implemented, which is then called from the function from the Fortran code.



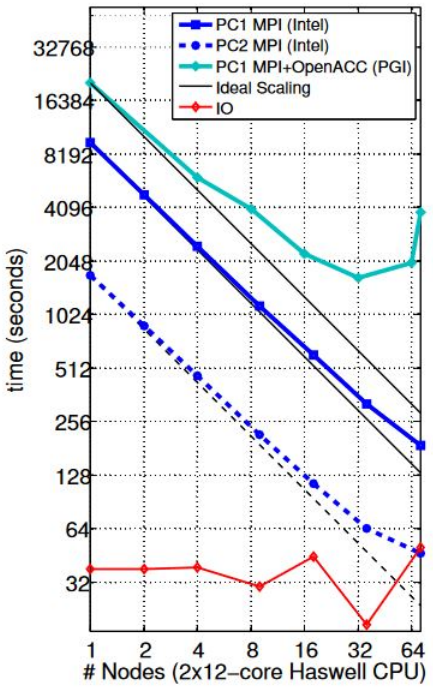


Figure 2.2: CPU timing results for POT3D with MPI and MPI+OpenACC. Intel 2015.2.164 compiler with the MVAPICH 2.1 MPI library was used for the MPI runs. For the MPI+OpenACC x86 multi-core runs, the PGI 17.5 compiler with the same version of the MVAPICH library was used. Left: results within a single compute node. Right: results for multiple nodes. The graphs were obtained from reference [2].

In reference [2], the performance of both preconditioners was analyzed using the Comet supercomputer at the San Diego Supercomputing Center. The tests involved CPU MPI scaling of POT3D and OpenACC compiled to x86 multi-core, as well as MPI+OpenACC runs on NVIDIA GPUs using the PSG cluster at NVIDIA. The tests were conducted using the PGI compiler (version 16.10 + CUDA 8.0) and the OpenMPI 1.10.2 MPI library.

From Figure 2.2 we see that, for the CPU runs, PC2 is much more efficient than PC1, running up to four times faster. The MPI-only code scales very well using multiple nodes up to about 64 nodes where it starts to lose efficiency, and the MPI+OpenACC on x86 multi-core is not running efficiently. Although the code shows decent scaling at first, as the number of nodes increases, the quality of the scaling decreases, and at a high number of nodes, the performance stops improving and starts getting worse. The GPU performance results of POT3D using MPI+OpenACC (Figure 2.3)show reasonable scaling, though efficiency is lost when run on many GPU nodes.

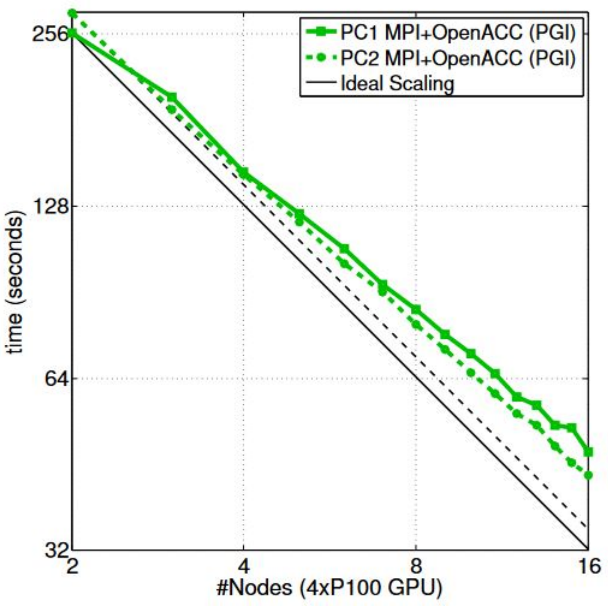


Figure 2.3: GPU timing results for POT3D with MPI+OpenACC for multiple nodes. The graph was obtained from reference [2].

2.4 Execution stages in POT3D solver

In order to integrate PETSc preconditioners and solvers into the POT3D application, it was essential to identify the various execution stages within the POT3D application. This analysis was conducted to discern the specific stages that required modifications and determine where PETSc function calls should be appropriately incorporated. The source code of POT3D was meticulously examined for this purpose, and the identified stages are outlined in this section.

2.4.1 Loading input and domain decomposition

The POT3D application has the capability for multiprocess execution using MPI. In the initial stage, each process reads the configurations from the pot3d.dat file. Additionally, based on the MPI process decomposition and the configurations in pot3d.dat, specific portions of the magnetic field data in the input `.h5` file are loaded into each process. The data forms the matrix A of the linear equation 2.3 that POT3D aims to solve, analogous to $Ax = b$. The loading process is distributed and utilizes a sparse Diagonal Storage Format (DIA) [19] to store the matrix A. Simultaneously, the right-hand side vector b and the solution vector x are initialized in a distributed manner as well.

For the purpose of this dissertation, we focus on the modifications made to the data after this initial stage, which are necessary for implementing PETSc preconditioners and solvers. Consequently, the specifics of domain decomposition and the intricacies of data loading are not discussed in this paper. However, further information about these aspects can be found in the documentation provided with the source code.

2.4.2 Preconditioning

POT3D offers a choice between two preconditioners: a straightforward Point-Jacobi preconditioner and a relatively sophisticated Incomplete LU (ILU0) preconditioner, as discussed in Section 2.2. After loading the data for matrix A (Equation 2.3), it undergoes preconditioning to enhance the convergence rate of the conjugate gradient solver.

2.4.2.1 Point-Jacobi/diagonal-scaling (PC1)

The Point Jacobi preconditioner rescales the matrix to improve the overall system's conditioning. This rescaling involves calculating the reciprocals of the diagonal elements (diagonal scaling factors) of the matrix [16]. Then, each element of the matrix is multiplied by its corresponding diagonal scaling factor, which effectively adjusts the magnitudes of the elements.

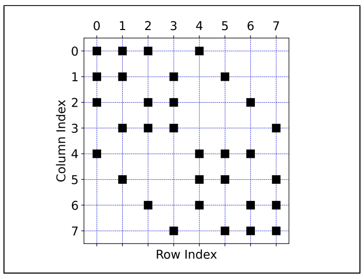
In POT3D, the diagonal scaling factors are computed and stored in an array before the iterative solver stage. During each iteration of the CG solver, the matrix elements are rescaled by performing element-wise scalar multiplication with their corresponding diagonal scaling factor.

2.4.2.2 Incomplete LU (ILU0) factorization (PC2)

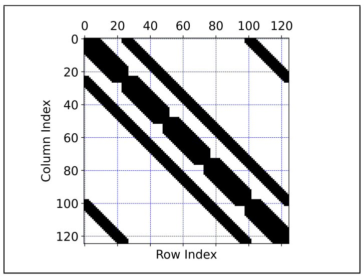
The ILU0 preconditioning technique enhances solver convergence by approximating a matrix using lower and upper triangular factors [17]. To implement ILU0 preconditioning in POT3D, the A matrix is transformed from a sparse Diagonal Storage Format (DIA) [19] to a compressed sparse row (CSR) format [20]. This transformation is essential as the CSR format enables efficient matrix-vector multiplications and is compatible with the external cuSPARSE library—an option for preconditioning in POT3D.

During this conversion process, it's important to note that the coefficients that multiply the boundary values are not included in the resultant CSR matrix. Each boundary has a subset of coefficients in the matrix row that is not included during the conversion, leading to the loss of boundary condition information in the CSR matrix. However, these missing coefficients are incorporated during the iterative solver stage based on the boundary conditions specified in the input configuration file (pot3d.dat), as discussed in Section 2.4.3.

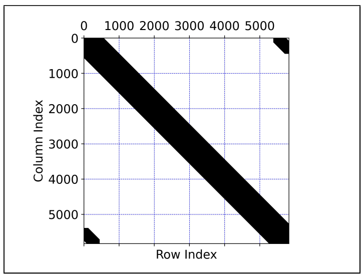
Alongside this, the conversion includes zero approximations for values using a tolerance threshold. The sparsity pattern of the resulting CSR matrix is illustrated in Figure 2.4.



(a) Dimension: 8×8
Sparsity: 50%



(b) Dimension: 125×125
Sparsity: 95.04%



(c) Dimension: 5832×5832
Sparsity: 99.88%

Figure 2.4: Sparsity pattern of matrix A in CSR format preconditioned with the ILU0 preconditioner for various problem sizes. The sparsity percentage indicates the proportion of zero elements in the matrix.

After the conversion, the CSR matrix is factorized using ILU0 [17]. The resulting LU factor matrix is utilized by the conjugate gradient solver to achieve faster convergence of the solution. The solver employs a forward and backward solve for the sparse system $(LU)x = y$ [21].

2.4.3 Iterative solver

POT3D utilizes an iterative conjugate gradient solver [22] to provide potential field solutions for three different models: potential field source surface (PFSS), potential field current sheet (PFCS), and open field (OF).

In the implementation, these models are distinguished based on how the boundary coefficients are treated within the solver space. For the OF and PFSS models, the radial boundary coefficients are excluded from the solver space, allowing field lines to extend beyond the outer boundary (Figure 2.1). On the other hand, for the PFCS model, the radial boundaries are included, causing field lines to close within the outer boundary (Figure 2.1). This differentiation occurs while preparing the direction vector for the iterative solver.

In the CG solver of POT3D, the matrix-direction vector product is achieved by unpacking the direction vector into an array of the same size as the local coefficient matrix A. This enables element-wise multiplication and addition for each row. During the unpacking process, coefficients are added to or removed from the direction vector elements that are associated with the radial boundaries, as determined by the model specified in the input configuration file pot3d.dat.

As the application involves distributed data loading (as discussed in Section 2.4.1), some direction vector elements required for the local matrix-vector product may reside in neighbouring processes. To address this, explicit MPI communication is performed during each iteration to obtain the missing elements in each process.

This customization in the CG solver enables POT3D to handle its specific application requirements while the remaining operations adhere to the standard iterative CG solver procedures.

2.4.4 Compute magnetic field solutions

The magnetic scalar potential, denoted as Φ, is treated as a vector in our calculations, akin to an array. This vector Φ is derived through the iterative solving process as described in equation (2.3). Subsequently, this derived vector Φ is employed to calculate magnetic fields along the radial (r), azimuthal (θ), and polar (ϕ) directions. Moreover, it aids in determining the overall magnitude of the magnetic field. To achieve this, the distributed vector Φ is gathered in the root process (rank = 0), where the calculations for these values are performed. Additionally, data logging is carried out in the root process.

2.5 PETSc: A scalable toolkit for parallel linear solvers

PETSc (Portable, Extensible Toolkit for Scientific Computation) is a widely used open-source software library designed to facilitate the implementation of scalable parallel numerical solvers for a variety of scientific and engineering applications. It offers a comprehensive set of tools and data structures to address the challenges of large-scale parallel computations, making it a valuable resource for researchers and practitioners in the field of computational science and engineering [23].

At its core, PETSc provides an array of functionalities, including linear and nonlinear solvers, time integration, and optimization routines. One of its primary strengths lies in its ability to perform parallel computations efficiently on distributed memory systems, enabling users to harness the full potential of high-performance computing (HPC) clusters and supercomputers [24]. The toolkit's development began in the 1990s at the Mathematics and Computer Science

Division of Argonne National Laboratory, and it has since grown to be a collaborative effort involving contributions from numerous researchers worldwide. This collaborative nature has resulted in a robust, feature-rich software package that continues to evolve with new algorithms and optimization techniques [25]].

Parallel linear solvers are a cornerstone of many scientific simulations, and PETSc offers a wide range of scalable algorithms for solving linear systems arising from partial differential equations (PDEs) and other numerical problems. These solvers are designed to take advantage of the inherent parallelism in modern computing architectures, allowing researchers to efficiently tackle complex problems on massively parallel systems [26].

The integration of PETSc has yielded significant results across diverse scientific applications. In cardiac simulations [27], PETSc facilitated accurate modelling of blood flow in the heart, uncovering crucial physiological insights. In topology optimization [28], PETSc enabled parallel solving of complex structural design problems, revealing novel features. Moreover, in cardiac modelling [29], PETSc's modular solver efficiently simulated intricate electrophysiological processes, allowing adjustments for specific electrocardiogram patterns. PETSc has also been used in conjunction with other software packages to solve complex scientific computing problems. For example, PETSc has been used with the finite element method software package FEniCS to solve a wide range of partial differential equations [30]. In the context of Fortran applications like POT3D, it presents an excellent opportunity to enhance computational efficiency, reduce memory consumption, and enable simulations on larger problem sizes that are potentially beyond the capabilities of serial solvers [11].

2.6 PETSc integration in Fortran applications

While PETSc primarily supports C and C++, it provides Fortran interfaces, enabling the integration of PETSc routines into Fortran codes [31]. This feature allows researchers to leverage PETSc's advanced functionalities without rewriting their entire Fortran application, thus preserving existing investments in code development. In the context of Fortran applications like POT3D, PETSc integration offers seamless access to a diverse collection of linear and nonlinear solvers [12]. By leveraging PETSc's built-in solvers, researchers can avoid the time-consuming and error-prone process of implementing custom solvers from scratch. This feature facilitates the adoption of state-of-the-art algorithms and methodologies, enhancing the solver's robustness and accuracy. Moreover, PETSc simplifies the management of parallelism, allowing Fortran developers to focus on the scientific aspects of their applications. The library abstracts away complex communication and data distribution details, enabling researchers to easily parallelize their Fortran codes without delving into the intricacies of parallel programming.

However, integrating PETSc into Fortran applications does pose certain challenges. One significant issue arises due to potential mismatches between the operation of the Fortran language and the C-based interfaces provided by PETSc. These differences stem from variations in memory management and the indexing methods. Proper care must be taken to correctly handle data structures and memory allocation in order to ensure effective communication between the Fortran and PETSc components [32]. Furthermore, developers may need to invest time in comprehending and navigating PETSc's extensive documentation and APIs to make optimal use of its features.

2.7 Preconditioners and solvers in PETSc

In the PETSc framework (Portable, Extensible Toolkit for Scientific Computing), achieving efficient and accurate solutions relies on two essential components: Preconditioners and Solvers [33]. PETSc provides a comprehensive range of preconditioners [34], each designed for specific scenarios to enhance the convergence of iterative solvers. When dealing with matrices where the main diagonal elements dominate, the Jacobi preconditioner (JAC) [16] is beneficial. JAC effectively exploits these dominant diagonal elements to accelerate convergence for well-structured matrices. For matrices with intricate patterns of empty entries (sparsity), the Incomplete LU factorization (ILU) preconditioner [17] is effective. ILU approximates the matrix factorization while ignoring less significant entries, making it suitable for matrices with irregular patterns of sparsity. It often yields improved convergence rates for various problems. In cases involving complex geometries and varying scales, the Algebraic Multigrid (AMG) preconditioner [35] excels. AMG employs a hierarchy of grids and interpolation to efficiently tackle systems with diverse scales and intricate geometries.

Transitioning to solvers in PETSc [12], the Generalized Minimal Residual (GMRES) solver [36] is a valuable choice for systems lacking symmetry or dominant diagonal characteristics. GMRES, when paired with suitable preconditioners, demonstrates efficiency in converging for challenging and less-well-behaved systems. When dealing with symmetric positive definite matrices, the Conjugate Gradient (CG) solver [15] stands out. CG is particularly efficient for self-adjoint systems, benefiting from symmetric preconditioners and exhibiting swift convergence. In situations encompassing nonsymmetric matrices without specific structural properties, the BiConjugate Gradient Stabilized (BCGS) solver [37] is useful. Its fusion of conjugate gradient and stabilized methods provides versatility for a broad spectrum of linear systems.

For problems rooted in partial differential equations (PDEs) and sparse systems, these iterative solvers offer robust solutions. Alternatively, in the case of potentially dense systems rather than sparse ones, PETSc's direct solvers such as Lower-Upper Factorization (LU) [38] provide accurate solutions. This array of preconditioners and solvers equips PETSc with a versatile toolkit, enabling researchers to address diverse scientific computing challenges with tailored efficiency and accuracy.

2.8 PETSc's GPU capabilities

Traditionally, PETSc has been utilized for parallel computations on CPU architectures, but recent developments have introduced support for leveraging the computational power of GPUs to accelerate linear solvers and preconditioners [39].

In 2013, Minden et al. introduced a preliminary version of PETSc that leveraged GPUs for improved performance [40]. They introduced a novel type of vectors and sparse matrices within this version, designed to execute operations directly on NVIDIA GPU processors. This enhanced implementation facilitates the use of Krylov methods, nonlinear solvers, and integrators within PETSc. Notably, these capabilities can be utilized without significant alterations to the existing API calls. This version of PETSc also seamlessly integrates with existing application codes written in languages like C, C++, Fortran, or Python.

Cuomo et al. (2015) [41] delved deeper into the potential of integrating PETSc with GPUs. They constructed a multi-level parallel framework for calculating Optical Flow on GPU clusters. This framework effectively employed the integration of PETSc and CUDA, yielding noteworthy enhancements in performance. The achieved improvement was substantial, approximately 95%, when compared to the conventional sequential implementation. Through the synergistic utilization of GPU clusters and the scientific computing middleware facilitated by PETSc, the authors aptly demonstrated the prowess of GPU acceleration in solving parabolic partial differential equations (PDEs) within heterogeneous computing environments.

To evaluate the performance of the GPU implementation of PETSc, Kumbhar (2011) [42] conducted a study using different sparse matrix storage schemes like Compressed Sparse Row (CSR), ELL, Diagonal, and Hybrid. The results indicated that for structured matrices, the GPU implementation showed up to a 4x speedup compared to an Intel Xeon quad-core CPU. However, the speedup diminished for multi-GPU applications due to high communication costs on the GPU cluster. Nevertheless, by implementing new storage schemes, Kumbhar achieved a 50% improvement in sparse matrix-vector operations and a 7x speedup for structured matrices, significantly enhancing the performance of vector operations on the GPU.

Chapter 3

ISC Student Cluster Competition

The ISC Student Cluster Competition (SCC) is an annual event that assembles teams of students from around the world to participate in a challenge focused on designing, constructing, and operating their own high-performance cluster. One of the key aspects of the competition is the benchmarking of the teams' clusters using a range of performance benchmarks.

3.1 Rules

In the competition, teams must run their programs on their cluster within a limited time frame of a few hours, while ensuring that the power consumption of the cluster does not exceed 6 kW as measured by a connected device. Each program may be run multiple times, but only one set of results per program can be submitted. The goal of each program varies, with some requiring the program to be executed in the shortest time possible, while others require the program to execute the highest number of FLOPs within a fixed period.

This year, the competition will test six programs on the clusters: POT3D [43], High-Performance Conjugate Gradient (HPCG) [44], High-Performance Linpack (HPL) [45], HPC Challenge (HPCC) [46], FluTAS [47], and Quantum Espresso [48]. I was responsible for testing and running POT3D and HPCG. HPCG is a common benchmark in high-performance computing, which executes a conjugate gradient algorithm to solve a 3-dimensional elliptic partial differential equation, primarily testing the machine's memory bandwidth. The goal of the SCC for HPCG is to achieve the highest average FLOP rate during a run of at least 180 seconds. For POT3D, the goal is simply to run the program with the provided input as quickly as possible.

3.2 Setup and hardware

The SCC cluster comprises two nodes. Each node features a single 64-core AMD EPYC 7713 processor. The storage system is composed of five 3TB NVMEs arranged within an XFS filesystem. Effective air-cooling is employed for the entire cluster. Additionally, each node is equipped with 8 NVIDIA A100 GPUs, each boasting a capacity of 40G. For the execution of HPCG on the cluster, we utilize a binary supplied by NVIDIA, integrated into the 21.4-hpcg container image [49]. This binary harnesses the computational power of the GPUs via the Docker container framework. In the case of compiling and executing POT3D, we rely on the nvhpc/22.11 SDK [50], which is seamlessly operational on the GPUs of the cluster.

3.3 Benchmarking and pre-competition results

3.3.1 HPCG benchmarking and results

The HPCG benchmark employs different methods of distributing computing tasks among processors and accelerators. As a result, average memory latency and bandwidth varied depending on the method used. The benchmark is primarily memory-bound, so these variations have a significant impact on performance. GPUs on each node were allocated to different NUMA regions, with two GPUs per region. To utilize this configuration optimally, the computation and communication patterns were adjusted during runtime employing the `-mem-affinity`, `-cpu-affinity`, and `-gpu-affinity` parameters in conjunction with the `numactl` tool. Each parameter is a list of indices, separated by colons, that associates data in specific MPI ranks with specific GPUs, blocks of memory, and sets of CPU cores. For optimal performance on one node, the parameters used were `3:3:1:1:7:7:5:5` for memory, `48-55:56-63:16-23:24-31:112-119:120-127:80-87 :88-95` for CPU affinity, and `0:1:2:3:4:5:6:7` for GPU affinity. Here, memory affinity refers to the desired memory location for each MPI rank. In this scenario, the GPU and memory affinities are selected to ensure that the corresponding CPUs have the memory in very close physical proximity, specific to the SCC cluster's NUMA configuration. For instance, the first MPI rank is assigned to CPU cores `48-55` and GPU `0`, and its memory affinity is set to `3`, which means that it will preferentially use the memory located in NUMA region 3 and GPU `0`, which are physically closer to its assigned CPU cores. This will reduce the latency.

The HPCG benchmark's grid size for the conjugate gradient algorithm was adjustable. Increasing the grid size reduced the load on memory bandwidth, allowing for a higher clock rate. However, using small grid sizes that fit critical arrays in the cache was invalid as the benchmark aimed to measure cluster memory performance. After experiments, a grid size of 400^3 was discovered as the optimal choice since it fits in memory and larger sizes caused crashes. Thus, 400^3 was chosen as the best option for the HPCG benchmark. The best performance observed for HPCG in one node is 1866.1 GFLOPs.

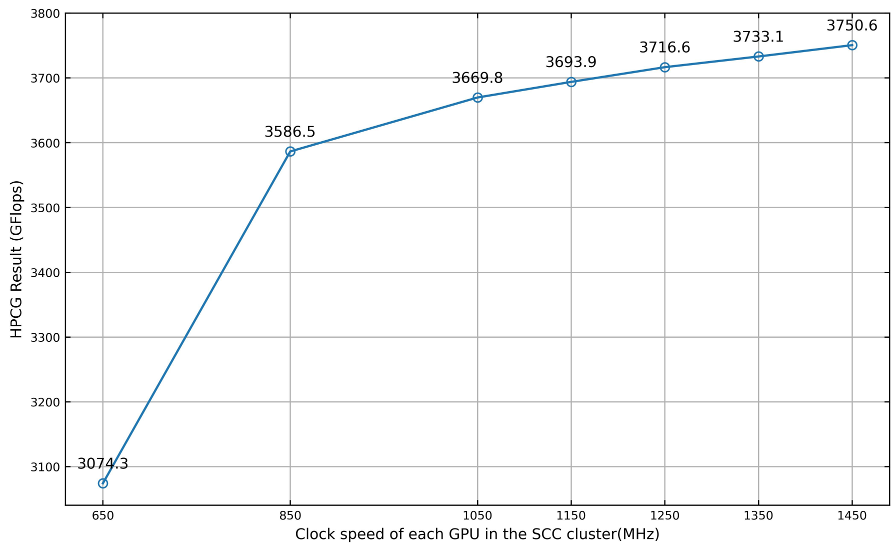


Figure 3.1: Plot shows the relationship between the clock speed of each GPU in the 16 GPUs SCC cluster and the corresponding benchmark result in GFLOPs.

After conducting initial tests on a single node within the SCC cluster, we progressed to assess the benchmark's performance by executing it on both nodes of the cluster, capitalizing on the collective power of all 16 GPUs. During this phase, we took into account the power usage and clock speed of each individual GPU.

To maximize the efficiency of communication between the nodes, we employed the Unified Communication - X Framework (UCX) [51] alongside the `mpirun` command. In terms of optimizing the arrangement of processes, we opted for a different approach. Instead of explicitly designating device indices, we utilized the `--map-by numa` flag in conjunction with `mpirun`. This strategy enabled us to distribute the workload across the two nodes while considering their Non-Uniform Memory Access (NUMA) configurations.

This NUMA-aware mapping proved to be superior to the previous practice of specifying explicit affinity for each process, which we had employed during the single-node experiments. The NUMA-aware approach aligns processes with their respective local memory nodes automatically, thereby reducing memory latency and enhancing overall performance. In contrast, relying solely on explicit affinity might not adapt effectively to dynamic workload changes or system

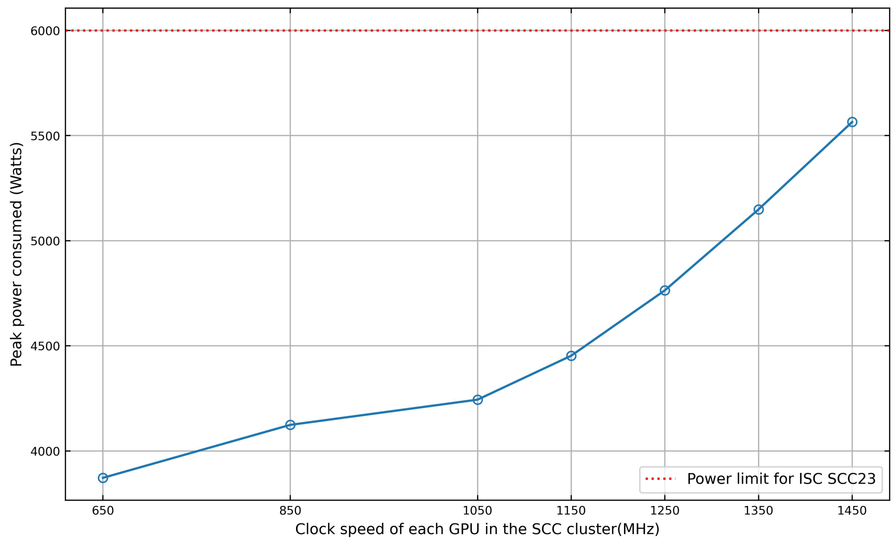


Figure 3.2: Plot displaying the relationship between the clock speed of each GPU in the 16 GPUs SCC cluster and the peak power consumption of the cluster, with a dotted line indicating the 6 kW power limit.

fluctuations, potentially resulting in less-than-optimal memory access patterns and diminished efficiency. Our chosen strategy effectively harnessed the NUMA configuration to achieve optimal benchmark execution.

To monitor power consumption, we utilized the Integrated Lights-Out (iLO) feature connected to each node. This allowed for remote management of the server's hardware, firmware, and software components.

The results we obtained are presented in Figures 3.1 and 3.2. Analyzing these graphs, we found a clear connection between the clock speed of each GPU and the resulting GFLOPs. When the clock speed is higher, the GFLOPs increase, but this also leads to higher power consumption.

It's interesting to note that even when we pushed the GPUs to their maximum clock speed of 1450 MHz, the power consumption remained within the competition limit of 6 kW. This outcome is attributed to the location of the SCC cluster in the Advanced Computing Facility (ACF) [52] during our pre-competition test runs. Here, the environment is maintained at a cooler temperature, allowing the fans to operate at a more moderate speed and consume less power. Throughout our tests, the highest GFLOP count observed was 3750.6, showcasing the peak performance achieved in this study.

3.3.2 POT3D benchmarking and results

To run the POT3D application, the HDF5 library had to be built with the same compiler that would be used for the POT3D application. The MPI compiler wrappers from the nvhpc/22.11 SDK [50] were used to build the HDF5 library.

The source code of the application contains example files that can be used to build the application with MPI-only, MPI+OpenACC, and GPU configurations. The preconditioner (Section 2.3) used in this test can be configured through the input file using the parameter `iprec`, where `iprec=1` for PC1 (diagonal scaling / point-Jacobi) and `iprec=2` for PC2 (ILU0).

The source code also contains a `testsuite` directory that includes several tests with varying problem sizes and memory requirements. These tests are useful for validating the application's performance. The `validation` test has a grid size of $63 \times 91 \times 225$, which corresponds to 1.28 million cells. It requires approximately 1 GB of memory to run when using `ifprec=1`. The `small` test has a grid size of $133 \times 361 \times 901$, which corresponds to 43.26 million cells. It requires approximately 6 GB of memory to run when using `ifprec=1`. The `medium` test has a grid size of $267 \times 721 \times 1801$, which corresponds to 346.7 million cells. It requires approximately 41 GB of memory to run when using `ifprec=1`. The `large` test has a grid size of $535 \times 1441 \times 3601$, which corresponds to 2.78 billion cells. It requires approximately 330 GB of memory to run when using `ifprec=1`.

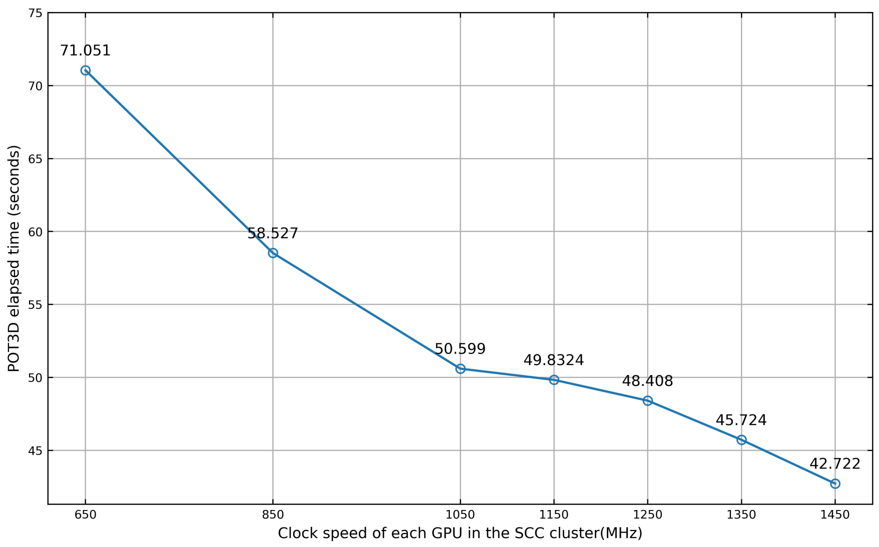


Figure 3.3: Plot shows the relationship between the clock speed of each GPU in the 16 GPUs SCC cluster and the corresponding elapsed time of the POT3D application.

The competition test, called `isc2023`, has a grid size of $325 \times 450 \times 2050$. Initially, the application was built with an MPI-only configuration, and the `isc2023` test was run with 10 CPU cores. The overall execution time was 10594.754653 seconds, which is approximately 3 hours. This confirms the results observed in the literature review section. Due to poor performance with the CPU and limitations in the number of CPU cores in the cluster, the application was rebuilt to use the GPUs. Initially, the `isc2023` test was run on 8 GPUs (One node of the SCC cluster), which elapsed for 187.397 seconds with `iprec=1` and 75.645 seconds with `iprec=2`. The significant performance boost is a consequence of utilizing a more advanced algorithm (ILU0) for preconditioning [17] with the setting `ifprec=2`. ILU0 solves a larger portion of the problem in each iteration, incurring a higher cost due to the initial LU factorization and the direct solve at each iteration. However, in this instance, it has successfully reduced the residual below the desired tolerance much more quickly. Each GPU in the cluster possesses 40 GB of rapid memory, which is essential for ensuring the efficient execution of the application. During each build, the small `validation` test was run to verify the application build.

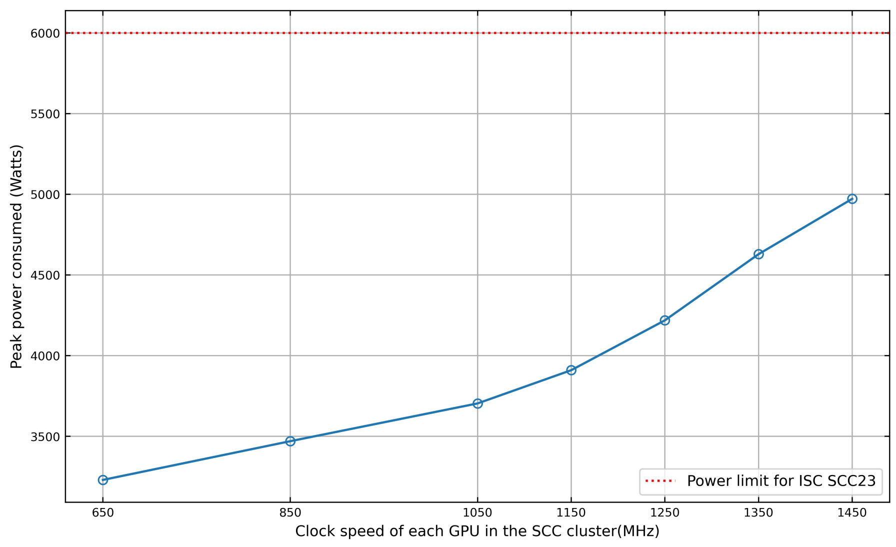


Figure 3.4: Plot displaying the relationship between the clock speed of each GPU in the 16 GPUs SCC cluster and the peak power consumption of the cluster, with a dotted line indicating the 6 kW power limit.

Initially, we ran the application on a single node to assess its behaviour. Subsequently, we extended the evaluation to both nodes of the cluster, utilizing all 16 GPUs to maximize performance. To optimize the execution time, we configured the application with `ifprec=2`, which yielded faster results during our initial analysis. We took into account the power consumption and clock speed of each GPU during the tests. To facilitate efficient communication between the two nodes, we employed the Unified Communication - X Framework (UCX) [51] with `mpirun`. Additionally, we recorded power consumption using Integrated Lights-Outs (iLO) connected to each node, providing remote management capabilities for the server's hardware, firmware, and software.

The outcomes of our evaluation are depicted in Figures 3.3 and 3.4. Analyzing the plots, we observed a reduction in the elapsed time of the POT3D solver with increasing clock speeds of the GPUs in the SCC cluster. This was accompanied by a corresponding rise in power consumption, with higher clock speeds leading to higher power consumption. However, even when running each GPU at the highest clock speed of 1450 MHz, the peak power consumption stayed below the established 6 kW limit for the competition. This achievement was possible because the SCC cluster operated within the Advanced Computing Facility (ACF) [52] during our pre-competition test runs. This controlled environment ensures lower temperatures, reducing the need for the fans to operate at higher speeds and consume additional power. Notably, the fastest elapsed time achieved during this stage was 42.722 seconds.

3.4 Competition results

The competition lasted for three days, with the HPCG benchmark and POT3D application results scheduled for submission on separate days. On the first day, we submitted the HPCG benchmark result, taking advantage of the cool environment caused by rain to run all the GPUs at the maximum 1450 MHz clock speed. This enabled us to draw just under 6 kW, which was within the competition limit. The benchmark result we submitted was 3732.5 GFLOPs.

The next day, during which the temperature was relatively warmer, we submitted the POT3D application results. Due to the higher temperature, we could only go up to 1300 MHz, resulting in a peak power consumption of 5.92 kW. The elapsed time we submitted for POT3D was 44.45 seconds. Remarkably, both of these results were the best among all the participants in their respective applications, significantly contributing to our team's overall victory in the competition.

Chapter 4

Experimental Implementation

In the implementation phase, PETSc preconditioners and solvers were integrated into the existing POT3D application. This involved strategic code modifications to enable PETSc functionalities, facilitating communication between POT3D and PETSc components. The adaptation covered single-process (Serial) execution, as well as multi-process execution on both CPU and GPU architectures, laying the foundation for subsequent analysis and assessment of performance improvements in the following chapter.

4.1 Experimental environment

The experimental implementation was conducted primarily on the Cirrus machine, supplemented by the SCC cluster 3.2 for prototyping PETSc implementations. The hardware configuration and specifications of the Cirrus machine are summarized in Table 4.1. For more comprehensive information, please refer to the documentation available at https://cirrus.readthedocs.io/en/main/. This environment enabled the integration and evaluation of PETSc preconditioners and solvers within the existing POT3D application.

In terms of the necessary software, we made use of the pre-existing OpenMPI compiler wrappers available on the Cirrus system. Additionally, we build the PETSc library with GPU compatibility and ensured it was equipped to accommodate external solver libraries, notably HYPRE [53], within our local environment. To further facilitate our work, we utilized the existing parallel HDF5 modules already integrated within Cirrus, which were essential for the operation of our application.

Compute Nodes	
Total Number of Nodes	318 (280 CPU + 38 GPU)
CPU Compute Nodes	Processors: 2.1 GHz, 18-core Intel Xeon E5-2695 Cores per Processor: 18 Hyperthreads per Core: 2
GPU Compute Nodes	Processors: 2.5 GHz, 20-core Intel Xeon Gold 6248 Cores per Processor: 20 Hyperthreads per Core: 2 GPUs per Node: 4 NVIDIA Tesla V100 (16G) GPUs Memory per Node: 384 GB
Infiniband Fabric	
Bandwidth	FDR, 54.5 Gb/s
Lustre File System	
Total Storage	406 TiB

Table 4.1: Specifications of Cirrus Machine.

4.2 Correctness testing

The correctness of the PETSc preconditioners and solver was assessed through a series of test runs. Firstly, utilizing the existing test suites embedded within the POT3D source code, we conducted comparative evaluations between the results produced by the PETSc preconditioners and solver and those generated by the native tests. This validation process served to confirm the accuracy and reliability of the PETSc implementations.

Moreover, for custom input scenarios, we performed thorough cross-validation by contrasting the outcomes of the PETSc preconditioners and solver with those attained through the native POT3D solver, utilizing identical input conditions. Given that the inherent nature of the POT3D solver is approximate, we focused on evaluating the statistical significance of differences rather than demanding an exact match. We concentrated on the 'Magnetic energy' value computed by both implementations and calculated the error disparity using the expression:

$$\text{Error} = \frac{\text{Magnetic energy from POT3D} - \text{Magnetic energy from PETSc}}{\text{Magnetic energy from POT3D}} \tag{4.1}$$

The given relative error, calculated using the equation (4.1) measures the difference between magnetic energy values obtained from POT3D and PETSc, relative to the value from POT3D. The consistently low error values below 2×10^{-5}

that we observed during all our test runs indicate that PETSc's results closely match those of POT3D, affirming the accuracy of PETSc's solver and preconditioners for the problem domain.

4.3 Implementation approach for preconditioning replacement with PETSc in POT3D

We began by attempting to replace the existing preconditioners 2.4.2 in the POT3D application as a starting point. As explained in Section 2.4.2.1, the point-Jacobi/diagonal scaling preconditioning involves a straightforward elemen-twise scalar multiplication with diagonal scaling factors. Because this is a simple operation to create a preconditioner matrix, we chose not to explore the option of using a PETSc function call for this purpose. Instead, our focus was on obtaining the factorized matrix using ILU0 preconditioning from PETSc.

To achieve the factorized matrix LU, the native POT3D employs two subroutines outlined in Snippet 4.1. The `diacsr` routine converts the sparse DIA format matrix `A` into a sparse CSR matrix, while also removing radial coefficients and zero approximations, as detailed in Section 2.4.2.2. The `ilu0` routine calculates the LU factorized matrix. Our goal was to replace the `ilu0` routine with PETSc function calls.

```
...
call diacsr(N,M,a,a_offsets,a_csr,a_csr_ja,a_csr_ia,a_csr_dptr)
call ilu0(N,M,a_csr,a_csr_ja,a_csr_ia,a_csr_dptr,icode)
...
```

Snippet 4.1: Function calls of ILU0 preconditioning in POT3D

Our process commenced by integrating the PETSc library into the POT3D application and introducing a new option for utilizing the PETSc preconditioner. By specifying `ifprec=3` in the input `pot3d.dat` file, we invoke a routine that applies preconditioning using PETSc. The implementation for obtaining the LU factorized matrix through PETSc is presented in Snippet 4.2. Here, we employ the `MatCreateSeqAIJWithArrays` function to construct a sparse PETSc matrix `a_mat` using arrays derived from the `diacsr` routine. The indices in the Fortran arrays `a_csr_ia` (row indices) and `a_csr_ja` (column indices) are adjusted to zero-based indexing suitable for PETSc function calls. The values `a_i`, `a_j`, and `a_data` correspond to the values in `a_csr_ia`, `a_csr_ja`, and `a_csr` and are used with PETSc Vector types and zero-based indices.

It's important to note that `PETSC_COMM_SELF` is employed to create local matrices and vectors, as we compute the LU matrix for the local `A` matrix values, which are already distributed. Once `a_mat` is established, we create a `lu_mat` matrix initialized to zero, serving as the container for the resulting LU matrix.

In PETSc, the LU factorized matrix can be obtained using the `KSPSolve` function, with the KSP type (Krylov Sub Space) set to `KSPPREONLY` through the `KSPSetType` function. This configures the solver to exclusively perform preconditioning. Given that we aim to replace the ILU0 preconditioner, we designate the preconditioner type as `PCILU` using the `PCSetType` function.

```
...
call MatCreateSeqAIJWithArrays(PETSC_COMM_SELF, n_pet , n_pet , ai , aj ,
    a_data ,a_mat , ie )
...
call MatConvert(a_mat ,MATSEQDENSE, MAT_INPLACE_MATRIX, a_mat , ie )
call MatCreateSeqDense(PETSC_COMM_SELF, n_pet , n_pet ,0.d0 , lu_mat , ie )
...
call VecSet(x_vec ,0.d0 , ie )
call VecSet(rhs_vec ,0.d0 , ie )
...
call KSPCreate(PETSC_COMM_SELF, ksp , ie )
call KSPGetPC(ksp ,pc , ie )
call KSPSetType(ksp ,KSPPREONLY, ie )
call PCSetOperators(pc ,a_mat ,a_mat , ie )
call PCSetType(pc ,PCILU, ie )
call PCFactorSetLevels(pc ,0 , ie )
...
call KSPSolve(ksp , rhs_vec , x_vec , ie )
call PCFactorGetMatrix(pc , lu_mat , ie )
...
```

Snippet 4.2: ILU0 preconditioning using PETSc

After performing the `KSPSolve` operation, we can obtain the factorized matrix using the `PCFactorGetMatrix` function. Since `KSPSolve` is primarily responsible for preconditioning, there is no need to focus on `x_vec` and `rhs_vec`, both of which are initialized as placeholders (set to zero). It is crucial to note that `lu_mat` is generated as a dense matrix, and similarly, `a_mat` is transformed into a dense matrix. This approach is chosen because the resulting LU matrix exhibits accurate values (consistent with those from the `ilu0` routine in POT3D) only when both matrices are in dense format. If either of the matrices is in sparse format, incorrect results are obtained. We observed this behaviour during our experimentation, and unfortunately, the PETSc documentation does not cover this aspect.

However, this implementation exhibited scalability issues, as employing the dense matrix format caused the application to exhaust memory resources for larger matrix sizes. Consequently, we endeavoured to implement a Conjugate Gradient (CG) solver using PETSc function calls, a subject discussed in the subsequent section.

4.4 Adapting matrix structure and boundary coefficients for CPU-only serial PETSc implementation of CG solver in POT3D

To implement a Conjugate Gradient (CG) solver using PETSc, we initially looked into creating a simpler version of the PETSc solver that works serially (single process). Our primary goal during this initial phase was to adapt the matrix data structure used in the POT3D code to a format suitable for the PETSc solver, in order to obtain comparable results to those generated by the POT3D.

One specific challenge we encountered involved how the radial boundary coefficients are handled during preconditioning and during each iteration of the POT3D solver. This process is explained in detail in Section 2.4.3. However, when it comes to implementing this in PETSc, we found that directly incorporating this specialized handling during preconditioning and iterations wasn't feasible. PETSc requires the matrix and vectors to be properly prepared before invoking the `KSPSolver` routine.

Upon further investigation into the interaction between the matrix `A` and the direction vector in each iteration of the native POT3D solver, we noted that the radial coefficients affect the diagonal elements of the matrix. In the case of the Open Field (OF) and Potential Field Source Surface (PFSS) options, the radial boundary coefficients are added to the diagonal values. Conversely, for the Potential Field Current Sheet (PFCS) option, these coefficients are subtracted from the diagonal elements.

```
...
do jj=1,IDIAG
  if (ioffok(jj).eq.0) then
    if (mi.eq.nrm1.and.option.ne.'potential') then
      Acsr(Adptr(i))=Acsr(Adptr(i))-real(Adia(i,jj),r_typ_pc)
    else
      Acsr(Adptr(i))=Acsr(Adptr(i))+real(Adia(i,jj),r_typ_pc)
    endif
  endif
enddo
...
```

Snippet 4.3: Handling boundary coefficients for PETSc

The process of removing the boundary coefficient is handled in the `diacsr` routine (as explained in Section 4.3). To accommodate this in PETSc, we created a new version of the routine called `diacsr_petsc`. At the end of this routine, we introduced additional code (Snippet 4.3) to handle the boundary coefficients. In this code, the `Adptr` array contains indices pointing to the diagonal values in each row, and the `ioffok` array helps identify whether a value in a row corresponds to a boundary coefficient (by setting the value at the column index to

0). Depending on the chosen option, the coefficient value is either added to or subtracted from the diagonal element of each row.

Since we now have a distinct routine to generate the Compressed Sparse Row (CSR) matrix for PETSc, we made adjustments to update the data for the `a_csr_ia` (row indices) and `a_csr_ja` (column indices) arrays to follow zero-based indexing.

In the previous set of experiments, as described in Section 4.3, we worked a routine using PETSc. This routine initially acted as a preconditioner, but we made modifications to transform it into a solver due to problems encountered with larger problem sizes.

To facilitate this transformation, we obtained data from the `diacsr_petsc` routine, specifically the `a_csr_ia`, `a_csr_ja`, and `a_csr` arrays. These arrays were then converted into PETSc vectors named `a_i`, `a_j`, and `a_data`. The code snippet presented in Snippet 4.4 demonstrates the implementation of the PETSc CG solver with ILU0 preconditioning, intended for single-processor execution.

```fortran
...
call MatCreateSeqAIJWithArrays(PETSC_COMM_SELF,n_pet,n_pet,ai,aj,
   a_data,a_mat,ie)
...
do i = 1,n_pet
  call VecSetValue(x_vec,i-1,x(i),INSERT_VALUES,ie)
  call VecSetValue(rhs_vec,i-1,rhs(i),INSERT_VALUES,ie)
end do
...
call KSPCreate(PETSC_COMM_SELF,ksp,ie)
call KSPGetPC(ksp,pc,ie)
call KSPSetType(ksp,KSPCG,ie)
call PCSetOperators(pc,a_mat,a_mat,ie)
call PCSetType(pc,PCILU,ie)
call PCFactorSetLevels(pc,0,ie)
call KSPSetTolerances(ksp,epscg,PETSC_DEFAULT_REAL,PETSC_DEFAULT_REAL
    ,ncgmax,ie)
...
call KSPSolve(ksp,rhs_vec,x_vec,ie)
...
```

Snippet 4.4: CPU-only serial PETSc CG solver with ILU0 preconditioner

For this solver, limited to MPI size = 1, we continued to use the `PETSC_COMM-_SELF` to create local matrices and vectors. In order to set up the solver, we initialized the `x_vec` and `rhs_vec` vectors with appropriate values extracted from the `x` and `rhs` arrays in POT3D.

Next, we configured the solver by specifying the Krylov subspace solver as `KSPCG` and setting the preconditioner to `PCILU` to enable ILU0 preconditioning. Once the matrix and vectors were properly set, we employed the `KSPSolve` function to execute an iterative solving process. The behaviour of `KSPSolve` adhered to the tolerance settings, which were established using the `KSPSet-Tolerances` function. These tolerances were derived from the `epscg` value (indicating the residual limit) and the `ncgmax` value (indicating the maximum number of iterations) as defined in the `pot3d.dat` input file. For example, we set `epscg=1.e-12` and `ncgmax=10000000` in all our runs in this study.

After executing the `KSPSolve` function, the outcome consisted of scalar values representing Φ (as described in equation (2.3)), stored in the `x_vec` vector. These values were then utilized to compute magnetic fields using the existing POT3D implementation. An important observation to note is that the sparse matrix format of `a_mat` prevented memory issues when handling larger problems.

4.4.1 Solver convergence and analysis for different options

During our validation, we noticed an issue with the PETSc solver when using the potential field current sheet (PFCS) option (models/options supported by POT3D are described in Section 2.4.3). Specifically, for certain problem sizes, the solver fails to converge due to a divergence in the residuals. The PETSc solver displays a message: `Linear solve did not converge due to DIVERGED_INDEFINITE_MAT`.

Upon examining the residual values during each iteration, we observed a pattern where the residuals decrease initially but start increasing after a certain point in the PETSc solver i.e., divergence. This convergence problem becomes more prominent for larger problem sizes. It's worth noting that the POT3D solver when utilizing the same PFCS option, manages to converge within the specified residual tolerance for various problem sizes.

In order to quantify the discrepancy between the diverged PETSc solver and the converged POT3D solver for the PFCS option, we calculated the error using the formula mentioned in Section 4.2. The computed error values range from 2×10^{-5} to 1×10^{-4}. Although this difference in error values isn't excessively significant, we didn't have sufficient confidence to proceed with test runs for the potential field current sheet (PFCS) option. Our suspicion is that this variance between the solvers might stem from the distinct treatment of boundary coefficients, as discussed in Section 2.4.3.

Adding to this challenge, the test dataset in POT3D's source code exclusively pertains to the potential field source surface (PFSS) option. Consequently, we lack a standard dataset to verify the solver's performance for the PFCS option. To address and resolve this issue, a potential approach would involve the development of a custom solver using lower-level components of PETSc, akin to POT3D. However, this undertaking falls beyond the scope of this dissertation.

Due to the aforementioned problems, we made the decision to exclude the solver for the potential field current sheet (PFCS) option from the subsequent analysis within this dissertation.

Referring to Snippet 4.3, we can observe that the matrix remains consistent for both the potential field source surface (PFSS) and open field (OF) options. Through our testing, we found that the PETSc solver consistently converged with error values below 2×10^{-5} for all problem sizes we examined with these options. Based on this observation, we will focus our further analysis in this dissertation solely on the potential field source surface (PFSS) and open field (OF) options.

4.5 Enhancing parallelism: Adapting PETSc solver for multi-process utilization

We initially developed and tested a serial PETSc solver. Later, our goal was to extend this solver's functionality to support multiple processes. During our analysis, we encountered an issue related to distributed input data loading in the POT3D application, discussed in Section 2.4.1. This problem emerged within the context of the CG solver's iterative process in POT3D, explained in Section 2.4.3. The challenge stemmed from adapting the standard MPI communication of direction vector elements used in POT3D's CG solver to the PETSc solver. Unlike the direct control over data in POT3D, PETSc's abstracted iterative solver component lacked similar data control. POT3D loads the data in a distributed manner, considering the specialized communication of direction vector elements during each iteration. Although we attempted to analyze the data loading algorithm of POT3D to adapt it for PETSc, its complexity rendered it impractical within the time constraints of the dissertation.

As a workaround, we loaded the complete data in each process and then initialized the PETSc matrix and vectors in a distributed manner. We generated a distributed matrix named `a_mat` using the `MatCreate` function. The matrix's type was designated as `MATMPIAIJ`, indicating that it is a distributed sparse matrix.

We then set up the distributed vectors and configured the solver environment. To do this, we utilized the `VecCreateMPI`and `KSPCreate` functions. It's important to emphasize that the arrays `a_csr_ia`, `a_csr_ja`, `a_csr`, `x`, and `rhs` contain all the input values for each process. In contrast to the native POT3D implementation, we did not load these values in a distributed manner, resulting in higher memory consumption with this specific approach. For initializing the values in the `a_mat` matrix, `x_pet`, and `rhs_pet` vectors, we employed the `MatSetValue` and `VecSetValue` functions. This initialization process involved determining the range of indices owned by each process and initializing only the corresponding segment of the distributed `a_mat`, `x_pet`, and `rhs_pet` objects.

```
...
call MatCreate(PETSC_COMM_WORLD,a_mat,ie)
call MatSetSizes(a_mat,PETSC_DECIDE,PETSC_DECIDE,n_pet,n_pet,ie)
call MatSetType(a_mat,MATMPIAIJ,ie)
call MatGetOwnershipRange(a_mat,rs,re,ie)
do i = 1, n_pet
    cs = a_csr_ia(i)
    ce = a_csr_ia(i+1) - 1
    do idx = cs, ce
            j = a_csr_ja(idx)
            if(i-1.ge.rs.and.i-1.lt.re) then
              call MatSetValue(a_mat,i-1,j-1,a_csr(idx),ADD_VALUES,ie)
            endif
    end do
end do
...
call VecCreateMPI(PETSC_COMM_WORLD,PETSC_DECIDE,n_pet,rhs_vec,ie)
call VecCreateMPI(PETSC_COMM_WORLD,PETSC_DECIDE,n_pet,x_vec,ie)
...
call VecGetOwnershipRange(rhs_vec,rs,re,ie)
do i = 1, n_pet
   if(i-1.ge.rs.and.i-1.lt.re) then
      call VecSetValue(rhs_vec,i-1,rhs(i),INSERT_VALUES,ie)
   endif
end do
call VecGetOwnershipRange(x_vec,rs,re,ie)
do i = 1, n_pet
   if(i-1.ge.rs.and.i-1.lt.re) then
   call VecSetValue(x_vec,i-1,x(i),INSERT_VALUES,ie)
   endif
end do
...
call KSPCreate(PETSC_COMM_WORLD,ksp,ie)
call KSPGetPC(ksp,pc,ie)
call KSPSetType(ksp,KSPCG,ie)
call PCSetOperators(pc,a_mat,a_mat,ie)
call PCSetType(pc,PCJACOBI,ie)
call KSPSetTolerances(ksp,epscg,PETSC_DEFAULT_REAL,PETSC_DEFAULT_REAL
     ,ncgmax,ie)
...
call KSPSolve(ksp,rhs_vec,x_vec,ie)
...
call VecGetLocalSize(x_vec,lcount,ie)
...
call VecGetValues(x_vec,lcount,ix,x,ie)
...
call MPI_Gatherv(x(1),lcount,ntype_real,x(1),counts,displs,ntype_real
     ,0,PETSC_COMM_WORLD,ie)
...
```

Snippet 4.5: CPU-only parallel PETSc CG solver with Jacobi preconditioner

With this parallel configuration, the `KSPSolve` function was executed to obtain results in the `x_vec` vector. The values within the resulting `x_vec` vector are distributed across processes. As a result, we collected all values from `x_vec` in the root process and exclusively computed the required magnetic field values there.

We verified this implementation by following the methodology outlined in Section 4.2 for the potential field source surface (PFSS) and open field (OF) options only due to reasons discussed in Section 4.4.1. Since this is a parallel implementation, we conducted extensive testing by running the application with numerous MPI processes. This approach allowed us to validate the accuracy and reliability of the implementation.

4.6 Enabling GPU utilization for the PETSc solver

To make use of GPUs with the PETSc solver, we follow the same approach as explained in Section 4.5 and shown in Snippet 4.5. However, we make some changes to the types of the matrix and vectors used.

For GPU utilization, we adjust the matrix `a_mat` to be of the type `MATMPIAIJ-CUSPARSE` instead of `MATMPIAIJ`. Similarly, we create the vectors using the `VecCreateMPICUDA` function instead of `VecCreateMPI`. Once these changes are applied, the `KSPSolve` function harnesses the power of GPUs to compute the solution.

We made sure that this modified implementation is accurate by following the steps described in Section 4.2 for the potential field source surface (PFSS) and open field (OF) options only due to the reasons discussed in Section 4.4.1. Since this is also a parallel implementation, we tested the results by running the application with multiple MPI processes (equivalent to the number of GPUs) to confirm the correctness of the implementation.

Chapter 5

Analysis of PETSc Preconditioners and Solvers: A Comparison with POT3D Application

In this chapter, we will examine how well PETSc's preconditioners and solvers perform when using the implementations we talked about in Chapter 4. We will compare this performance with how well the POT3D application solves the same problem. Additionally, we'll investigate how adaptable the PETSc implementations are.

5.1 Solver and preconditioner selection for analysis

Name	**Type**	**PETSc**	**POT3D**
Conjugate Gradient (CG)	Solver	Yes	Yes
GMRES (Generalized Minimal Residual)	Solver	Yes	No
BCGS (Stabilized Biconjugate Gradient)	Solver	Yes	No
Diagonal Scaling (Point Jacobi)	Preconditioner	Yes	Yes
ILU0 (Incomplete LU Factorization)	Preconditioner	Yes	Yes
AMG (Algebraic Multigrid)	Preconditioner	Yes	No

Table 5.1: Utilization of Solvers and Preconditioners in Native POT3D and PETSc Implementations.

In our study, we examined the efficiency of different solvers and preconditioners. We utilized three iterative solvers: Conjugate Gradient (CG), Generalized

Minimal Residual (GMRES), and a stabilized version of Biconjugate Gradient (BCGS). We also employed three preconditioners: point-Jacobi, Incomplete LU factorization (ILU0), and Algebraic Multigrid (AMG) (description of these preconditioners and solvers in Section 2.7). For our analysis, we employed a combination of these solvers and preconditioners in both the POT3D and PETSc solver versions. The specific usage scenarios are outlined in Table 5.1.

Our choice of solvers and preconditioners was guided by the capabilities of the POT3D application. We directly compared the POT3D and PETSc solver/preconditioner combinations, so we opted for solvers and preconditioners that are supported by both. This involved using the Conjugate Gradient solver along with the Point-Jacobi and ILU0 preconditioners. Furthermore, to broaden our assessment and due to our academic interests, we also introduced the GMRES and BCGS solvers, along with the AMG preconditioner. These selections not only enabled us to compare against unavailable setups but also provided a unique perspective by introducing distinct solver and preconditioner combinations.

5.2 Methodology

We conducted performance tests on the POT3D application, utilizing both CPU-only and GPU setups. We evaluated the application's performance using two solver options: the native POT3D solver and the PETSc solver with different preconditioners (Section 5.1). Our analysis focused specifically on the potential field source surface (PFSS) and open field (OF) configurations. This selection was based on the considerations outlined in Section 4.4.1.

These tests were carried out on the Cirrus machine, as detailed in Section 4.1. The Cirrus machine comprises CPU compute nodes as well as GPU nodes. Our experiments involved solving problems of varying sizes. The problem size is determined by the number of data points in the matrix `A` (equation (2.3)). The dimensions of this matrix are controlled by the values of `nr`, `nt`, and `np` i.e., the number of grid points in r, θ and ϕ directions mentioned in the input file `pot3d.dat`. In other words, the problem size is calculated as Problem size = $\mathtt{nr} \times \mathtt{nt} \times \mathtt{np}$.

In our performance assessment, our main focus was on the time taken to complete the POT3D and PETSc solver function calls (refer to Figures 5.1 and 5.2). We calculated these times by averaging results from three separate iterations. This approach ensures that all time-related values in this chapter are presented as averages across three iterations. To profile the implementations using different solvers, we employed the Arm Forge MAP application.

5.2.1 Analysis of residual convergence

In the preceding chapter (Chapter 4), we examine different implementations of the solver. To analyze each version's performance, we first investigate the rate of convergence for the residual error. We do this by looking at how quickly each iterative solver-preconditioner pairing converges (details in Table 5.1). We measure the number of iterations required by each pairing to solve a given problem and the time it takes to achieve this solution.

In this analysis, we have chosen substantial problem sizes: $200 \times 200 \times 200$ for CPU runs, and $133 \times 361 \times 901$ for GPU runs. The data for GPU runs is sourced from the test suite in POT3D. These sizes strike a balance between being large enough to yield meaningful results and manageable enough to ensure reasonable testing durations. It serves as a representative problem size, typical of those solved by POT3D for performance assessment. We arrived at this specific size after conducting initial trial runs.

Using the data from this analysis, we focus on identifying the combinations that are directly comparable with the POT3D and PETSc implementations for scaling analysis. Furthermore, we consider additional combinations that perform well in solving the problem. These extra combinations are chosen based on their ability to achieve a solution in fewer iterations and within a reasonable time frame compared to equivalent setups.

5.2.2 Scaling analysis

We conducted a strong scaling analysis to assess how efficiently the POT3D and PETSc implementations utilize additional resources in parallel runs. We selected specific combinations as outlined in Section 5.2.1 and executed them using varying numbers of processes. For CPU runs, we used 1, 2, 4, 8, 16, 32, 62, and 128 processes, distributed across 1 to 4 nodes on the Cirrus system. For GPU runs, the combinations were executed on 1, 2, 4, 8, 12, and 16 GPUs, utilizing 1 to 4 GPU nodes on the Cirrus machine. The problem size was consistent: $200 \times 200 \times 200$ for CPU runs and $133 \times 361 \times 901$ for GPU runs (test suite), as detailed in Section 5.2.1. To evaluate the achieved performance enhancement, we analyzed the elapsed times of solver function calls in both the PETSc and POT3D implementations. This analysis allowed us to compute the speedup and create a graph depicting its correlation with the number of employed processes. This graph provides valuable insights into the scaling behaviour of the system. The formula used to calculate speedup, as shown in Equation 5.1, is defined as:

$$\text{Speedup}(N) = \frac{T_s}{T_p(N)} \qquad (5.1)$$

Here, T_s represents the execution time on a single processor, while $T_p(N)$ represents the execution time on N processors.

In the context of parallel computing analysis, a clear understanding of parallel efficiency ($E(N)$) is crucial. This efficiency quantifies how effectively the parallel system utilizes additional processors. It is expressed as:

$$E(N) = \frac{\text{Speedup}(N)}{N} = \frac{T_s}{N \cdot T_p(N)} \tag{5.2}$$

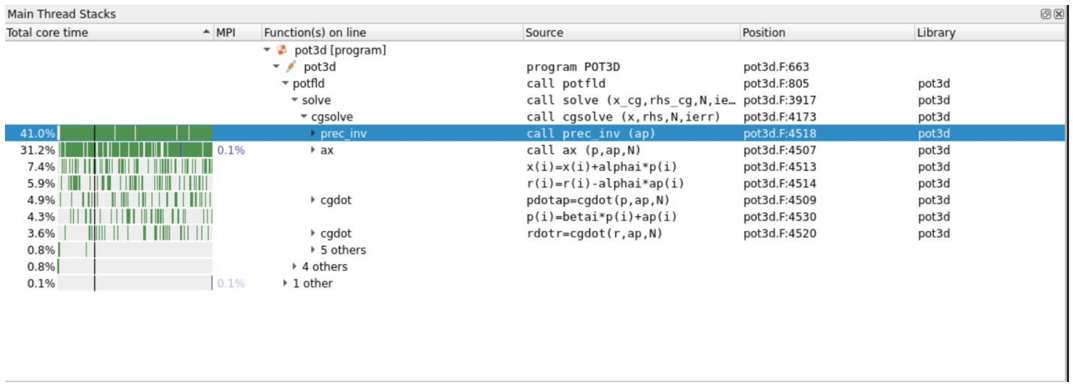
Visualizing the parallel efficiency through plotting enables us to assess the scalability of parallel algorithms. A higher parallel efficiency reflects efficient resource utilization. On the other hand, a descending curve in the parallel efficiency plot might indicate diminishing returns due to factors like communication overhead or load imbalance.

In the context of linear solvers like POT3D, weak scaling analysis is not applicable. This is because, when the problem size is increased with the number of processes (N), the actual problem solved by each N is different, necessitating varying numbers of iterations for solving each problem. Consequently, even with distributed data, the computational workload borne by each process differs across different combinations of problem sizes and numbers of processes during weak scaling analysis. Thus, weak scaling analysis is not performed.

5.3 Analysis of serial PETSc implementation and POT3D application in serial mode

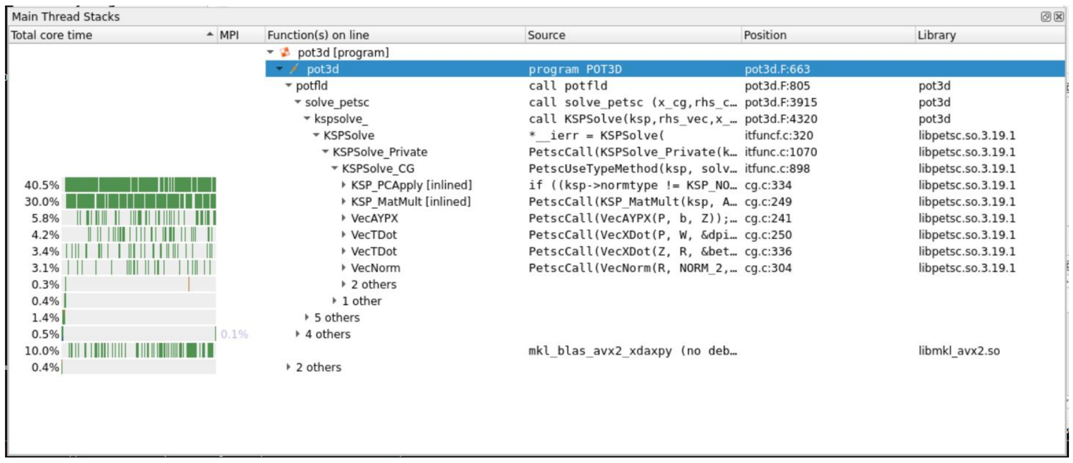
We initiated our analysis by comparing the serial PETSc implementation discussed in Section 4.4 with the POT3D application when executed in a serial manner, meaning with a single MPI process (MPI processes = 1). Our rationale for beginning with the serial implementation was to encompass intrinsic PETSc preconditioners such as `PCILU`. It's worth noting that PETSc's native `PCILU` is not available for parallel solvers, as explained in reference [33].

Our initial approach included using the ARM Forge MAP tool to analyze both versions of the application. The outcomes of this analysis are displayed in Figures 5.1 and 5.2. Notably, in both variations, the preconditioning steps (`KSP-_PCApply` and `prec_inv`) took up a similar proportion of the overall runtime. Similarly, the stages involving matrix multiplication (`KSP_MatMult` and `ax`) showed a comparable distribution in terms of runtime percentage. These two phases were the primary time-consuming components in each run for both versions.



```
Main Thread Stacks
Total core time   MPI    Function(s) on line        Source                               Position        Library
                         pot3d [program]
                          pot3d                     program POT3D                        pot3d.F:663
                           potfld                   call potfld                          pot3d.F:805     pot3d
                            solve                   call solve (x_cg,rhs_cg,N,ie…        pot3d.F:3917    pot3d
                             cgsolve                call cgsolve (x,rhs,N,ierr)          pot3d.F:4173    pot3d
41.0%                         prec_inv              call prec_inv (ap)                   pot3d.F:4518    pot3d
31.2%             0.1%        ax                    call ax (p,ap,N)                     pot3d.F:4507    pot3d
7.4%                                                x(i)=x(i)+alphai*p(i)                pot3d.F:4513    pot3d
5.9%                                                r(i)=r(i)-alphai*ap(i)               pot3d.F:4514    pot3d
4.9%                          cgdot                 pdotap=cgdot(p,ap,N)                 pot3d.F:4509    pot3d
4.3%                                                p(i)=betai*p(i)+ap(i)                pot3d.F:4530    pot3d
3.6%                          cgdot                 rdotr=cgdot(r,ap,N)                  pot3d.F:4520    pot3d
0.8%                          5 others
0.8%                        4 others
0.1%              0.1%     1 other
```

Figure 5.1: Performance profile of the POT3D application when executed with a single MPI process. The graph provides insights into the distribution of runtime across different stages of the computation.



```
Main Thread Stacks
Total core time   MPI    Function(s) on line          Source                               Position        Library
                         pot3d [program]
                          pot3d                       program POT3D                        pot3d.F:663
                           potfld                     call potfld                          pot3d.F:805     pot3d
                            solve_petsc               call solve_petsc (x_cg,rhs_c…        pot3d.F:3915    pot3d
                             kspsolve_                call KSPSolve(ksp,rhs_vec,x_…        pot3d.F:4320    pot3d
                              KSPSolve                *__ierr = KSPSolve(                  itfuncf.c:320   libpetsc.so.3.19.1
                               KSPSolve_Private       PetscCall(KSPSolve_Private(k…        itfunc.c:1070   libpetsc.so.3.19.1
                                KSPSolve_CG           PetscUseTypeMethod(ksp, solv…        itfunc.c:898    libpetsc.so.3.19.1
40.5%                            KSP_PCApply [inlined] if ((ksp->normtype != KSP_NO…       cg.c:334        libpetsc.so.3.19.1
30.0%                            KSP_MatMult [inlined] PetscCall(KSP_MatMult(ksp, A…       cg.c:249        libpetsc.so.3.19.1
5.8%                             VecAYPX              PetscCall(VecAYPX(P, b, Z));…        cg.c:241        libpetsc.so.3.19.1
4.2%                             VecTDot              PetscCall(VecXDot(P, W, &dpi…        cg.c:250        libpetsc.so.3.19.1
3.4%                             VecTDot              PetscCall(VecXDot(Z, R, &bet…        cg.c:336        libpetsc.so.3.19.1
3.1%                             VecNorm              PetscCall(VecNorm(R, NORM_2,…        cg.c:304        libpetsc.so.3.19.1
0.3%                             2 others
0.4%                            1 other
1.4%                         5 others
0.5%              0.1%      4 others
10.0%                                                 mkl_blas_avx2_xdaxpy (no deb…                        libmkl_avx2.so
0.4%                       2 others
```

Figure 5.2: Performance profile of the serial PETSc implementation. The graph provides insights into the distribution of runtime across different stages of the computation.

It's important to highlight that the solver functions utilized in the POT3D and PETSc implementations are `cgsolve` and `KSPSolve`, respectively. The timing results we present in this report exclusively pertain to the execution durations of these solver functions.

Given that the ILU0 preconditioner emerges as the more effective option between the two, it becomes evident that for sequential execution, the PETSc CG solver with ILU0 preconditioning offers the most favourable solution in the context of computing potential field solutions for both Potential Field Source Surface (PFSS) and Open Field (OF) models. While the standard PETSc library appears more suitable for single-threaded runs, practical application dictates that POT3D would be used mostly for larger problem sizes, requiring parallel computing capabilities to attain timely solutions.

5.3.1 Analysis of residual convergence

We began our comparative analysis between the serial PETSc implementation (Section 4.4) and the POT3D implementation by examining the convergence of residuals, as outlined in Section 5.2.1. The results are presented in Figures 5.3 and 5.4. The analysis involved executing both versions using different solver-preconditioner configurations on a single processing unit (CPU cores).

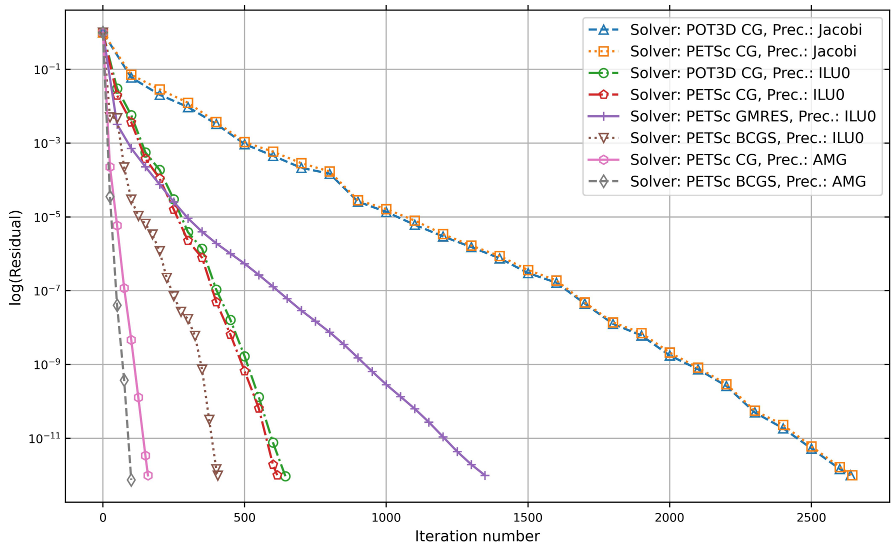


Figure 5.3: Convergence of residual errors for various solver-preconditioner combinations when solving a problem of dimensions $200 \times 200 \times 200$ using a single process. The residual values are presented in a logarithmic scale.

Figure 5.3 illustrates the convergence of the residual error for different combinations of solvers and preconditioners. The graph presents the number of iterations needed by each configuration to achieve a solution. When we directly compare the setups in both POT3D and PETSc implementations, specifically using the Conjugate Gradient (CG) solver with point-Jacobi and ILU0 preconditioners, we observe that runs with point-Jacobi preconditioners show similar results between the implementations. However, PETSc outperforms POT3D slightly when the ILU0 preconditioner is applied. When we extend our analysis to additional setups within PETSc, we find that the Generalized Minimal Residual (GMRES) solver takes more iterations to converge compared to equivalent setups. On the other hand, the Stabilized Biconjugate Gradient (BCGS) solver converges more quickly in terms of iterations when compared to those equivalent setups. Among all the configurations, the BCGS solver with Algebraic Multigrid (AMG) preconditioner demands the fewest iterations to reach a solution. Particularly, setups utilizing AMG preconditioners exhibit significantly fewer iterations for achieving a solution compared to other preconditioners.

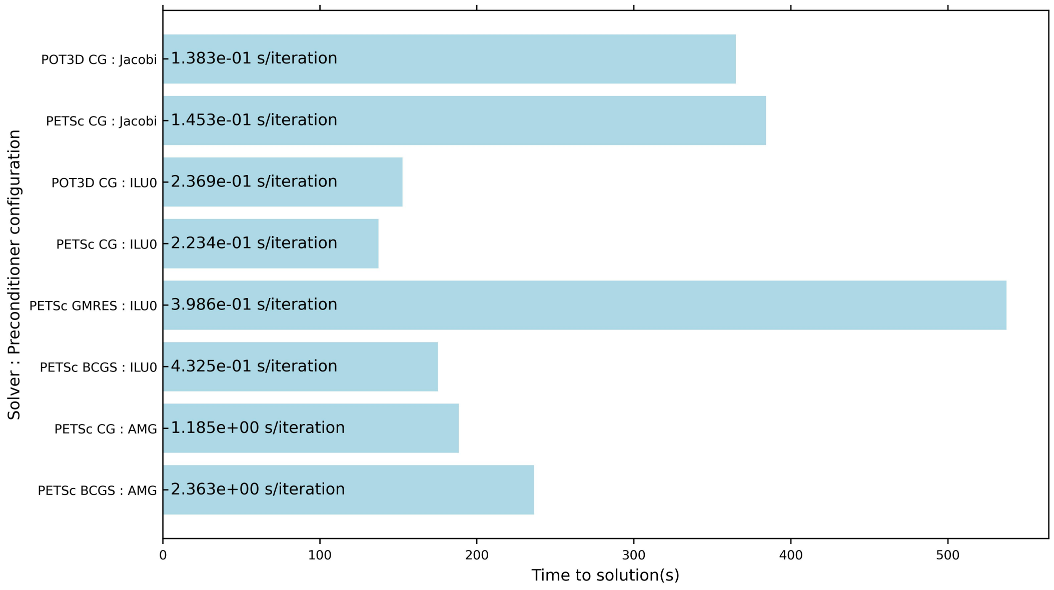


Figure 5.4: Time taken by various solver-preconditioner configurations to solve a problem of dimensions 200 $\times$ 200 $\times$ 200, all using a single processing unit. Each setup's elapsed time per iteration (seconds/iteration) is labelled on the graph.

In our further analysis, we referred to figure 5.4, which displays the actual time in seconds taken by different setups to reach a solution. We found that the setups utilizing the ILU0 preconditioner performed the best in solving the problem. Specifically, PETSc's ILU0 implementation with the Conjugate Gradient (CG) solver outperformed the corresponding POT3D setup by a slight margin. On the other hand, the Generalized Minimal Residual (GMRES) solver exhibited the poorest performance.

Interestingly, the Algebraic Multigrid (AMG) preconditioner required fewer iterations to arrive at a solution compared to the Incomplete LU Factorization (ILU0) preconditioner. However, the AMG preconditioner took longer per iteration than ILU0 due to its involvement in more complex operations like coarsening and interpolation. These additional steps introduced computational overhead. While AMG can achieve quicker convergence rates for specific problems, it's initial setup and intricate algorithmic steps contributed to longer per-iteration times compared to the simpler and faster-to-compute ILU0 preconditioner. This is evident in the time taken per iteration, as shown in figure 5.4.

Additionally, our observations revealed that the Stabilized Biconjugate Gradient (BCGS) solver needed fewer iterations to solve the problem than the CG solver. However, the BCGS solver took more time to reach a solution. The CG solver exhibited faster results for symmetric positive definite (SPD) matrices, such as the matrix A solved by the POT3D application. This speed advantage arises from CG's reduced extra operations per iteration, thanks to its exploitation of matrix symmetry. In contrast, BCGS and GMRES solvers required supplementary computations, leading to slower convergence for SPD matrices. This trend is also evident in the time taken per iteration, as depicted in figure 5.4.

5.4 Analysis of PETSc implementation and POT3D application in parallel CPU-only mode

The POT3D application is designed to efficiently solve large datasets by using multiple processing units simultaneously. In order to assess how well POT3D performs in parallel, we compare its performance with the parallel version of the PETSc solver, as explained in Section 4.5. Our observations and findings are detailed in this section. It's important to mention that the native PETSc ILU0 preconditioner, denoted as `PCILU`, cannot be used for multi-process runs. To address this limitation, we employed an external library called Hypre [53] through PETSc for ILU0 preconditioning. We utilized the `PCHYPRE` preconditioner with type `euclid`, as recommended in reference [33].

5.4.1 Analysis of residual convergence

We initiated our comparative study between the parallel PETSc implementation (see Section 4.5) and the POT3D implementation by investigating the reduction of errors over iterations, as explained in Section 5.2.1. The outcomes are shown in Figures 5.5 and 5.6. This examination encompassed the operation of both variants with varying solver-preconditioner combinations with 36 processing cores, corresponding to a single standard CPU compute node within the Cirrus machine.

The results closely align with the analysis of similar setups detailed in Section 5.3.1. The reasoning behind the conclusions explored in Section 5.3.1 appears to be applicable to the parallel runs as well. Furthermore, there's an observable fluctuation in how well the Biconjugate Gradient (BCGS) solver converges, as depicted in Figure 5.5. Symmetric Positive Definite (SPD) matrices, as used in POT3D, possess distinct characteristics that render them suitable for certain solvers. Conjugate Gradient (CG) is well-suited for symmetric positive definite matrices due to its efficient convergence, leveraging matrix symmetry to minimize errors in orthogonal directions. However, BCGS can exhibit slow convergence or divergence with these matrices, and GMRES, designed for nonsymmetric matrices, may demand higher resources without harnessing the symmetry advantage efficiently.

The sole difference between the analysis in this section and Section 5.3.1 lies in the inclusion of an Incomplete LU factorization (ILU) preconditioner from an

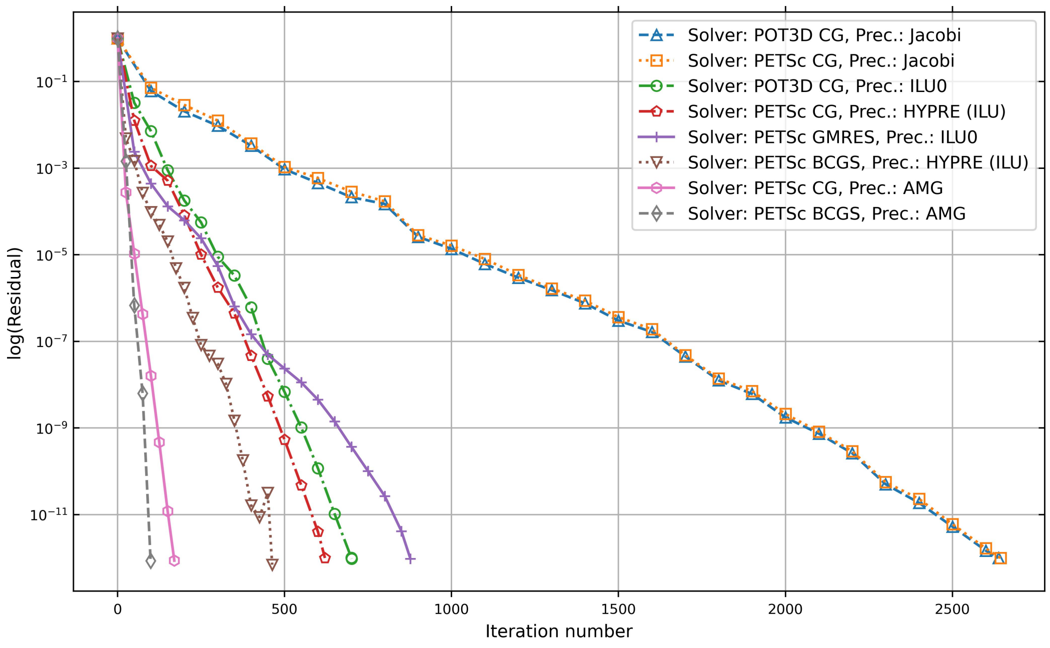


Figure 5.5: Convergence of residual errors for various solver-preconditioner combinations when solving a problem of dimensions 200 $\times$ 200 $\times$ 200 using 36 CPU cores. The residual values are presented in a logarithmic scale.

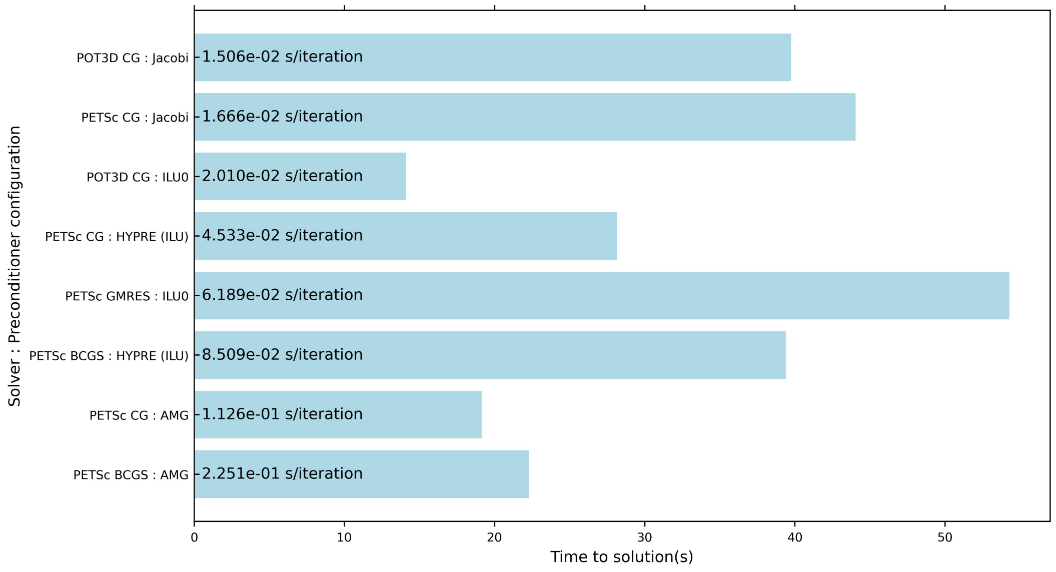


Figure 5.6: Time taken by various solver-preconditioner configurations to solve a problem of dimensions 200 $\times$ 200 $\times$ 200, all using 36 CPU cores. Each setup's elapsed time per iteration (seconds/iteration) is labelled on the graph.

external library (Hypre) in PETSc's parallel implementation. Notably, configurations that make use of this preconditioner (`PCHYPRE`) demonstrate inferior performance compared to their POT3D counterparts. This inconsistency arises due to the distinct algorithms used in Hypre and the additional overhead introduced by calls to the external Hypre library within PETSc. Notably, the most effective configuration identified in this analysis involves the POT3D Conjugate Gradient (CG) solver, coupled with the ILU0 preconditioner. This demonstrates that native POT3D solvers outperform PETSc solvers in solving the potential field problem.

5.5 Strong scaling analysis

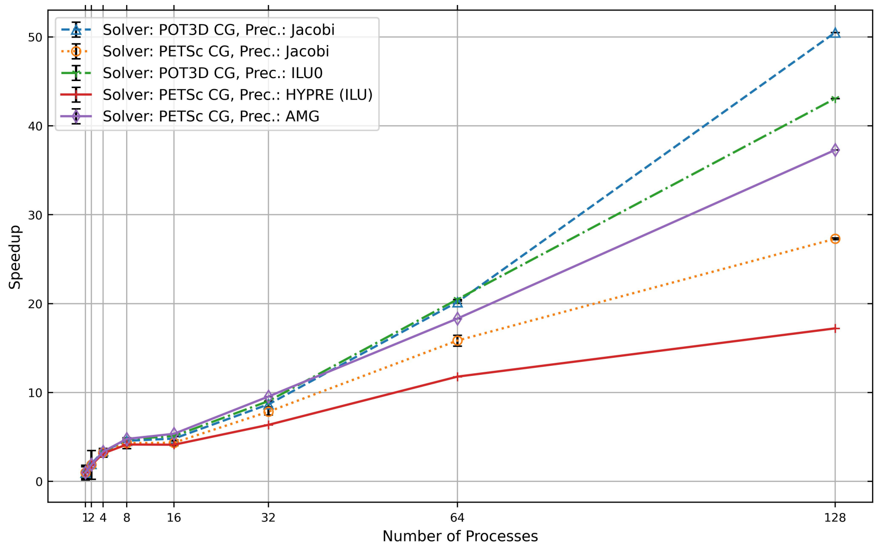


Figure 5.7: Speedup comparison of solver configurations for the POT3D and PETSc implementations on varying CPU cores/processes. The problem size is 200 × 200 × 200. The error bars represent standard deviations.

In our analysis of parallel implementations, we conducted a strong scaling assessment on specific configurations from both the POT3D and PETSc implementations. This evaluation was carried out using the approach outlined in Section 5.2.2, utilizing the CPU compute nodes of the Cirrus system. This analysis aimed to provide insights into the efficiency of both implementations in leveraging additional resources during parallel operations to solve the targeted problem within the POT3D application.

Figure 5.7 depicts the speedup performance, while Figure 5.8 displays the effectiveness of various solver configurations when executed on different numbers of CPU cores or processes. The specific problem under consideration involves dimensions of 200 × 200 × 200. It's important to note that the POT3D implementation demonstrates notably better speedup and efficiency compared to the

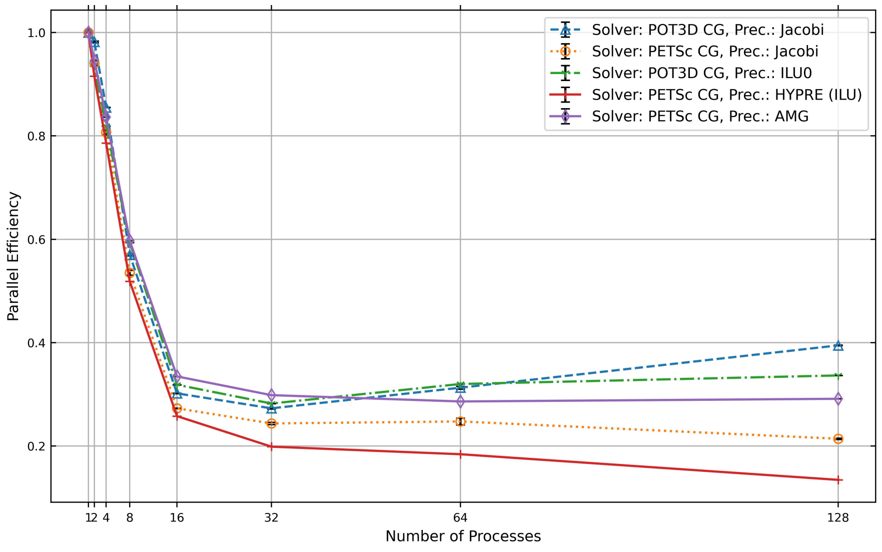


Figure 5.8: Parallel efficiency of solver configurations for the POT3D and PETSc implementations on varying CPU cores/processes. The problem size is 200 $\times$ 200 $\times$ 200. The error bars represent standard deviations.

PETSc implementation. This difference is attributed to the tailored optimization within the POT3D solver, which has been carefully handled to address the intricacies of the potential field problem. However, the parallel efficiency of all the setups drops to approximately 60% (see Figure 5.8) when we utilize more than 8 CPU cores/processes. This makes the solvers inefficient in solving the given problem with more than 8 processes. We observed identical scaling and efficiency for a larger problem size (133 $\times$ 361 $\times$ 901) as well.

The time taken for each iteration in various setups, when executed in parallel with different numbers of processes, is depicted in Figure 5.9. This observation aligns with the data presented in Figure 5.6 in most scenarios. However, in cases where a higher number of processes (128) are employed, the PETSc CG solver with Jacobi preconditioner exhibits slightly longer execution times compared to the POT3D CG solver with ILU0 preconditioner. This discrepancy in the observed results might arise from inefficiencies in the processing of the PETSc Jacobi preconditioner, particularly when a substantial number of processes are utilized.

Among the setups, the PETSc's CG solver with an Algebraic Multigrid (AMG) preconditioner demonstrated the closest speedup performance to the POT3D implementation. However, even with the utilization of a sophisticated preconditioner like AMG, we couldn't achieve performance that matched or surpassed the results attained by the customized POT3D implementation with PETSc.

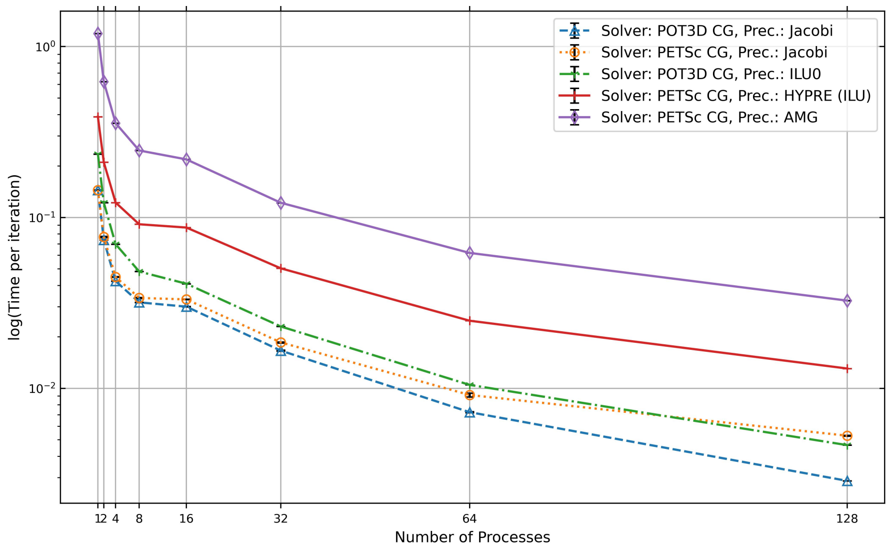


Figure 5.9: Time per iteration of solver configurations for the POT3D and PETSc implementations on varying CPU cores/processes. The problem size is 200 $\times$ 200 $\times$ 200. The time per iteration values are presented in a logarithmic scale. The error bars represent standard deviations.

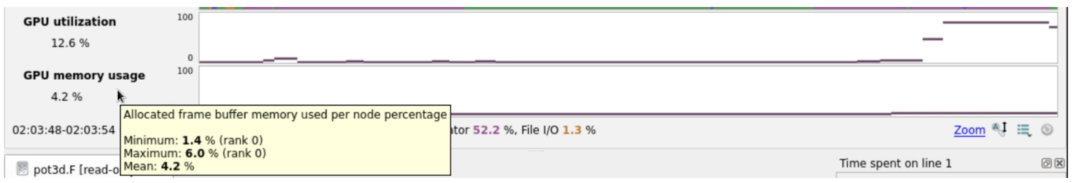
5.6 Analysis of PETSc implementation and POT3D application in parallel GPU mode

The POT3D application can utilize GPUs for improved performance (see Section 2.3). Our objective was to compare the GPU utilization of POT3D with PETSc's GPU implementation, discussed in Section 4.6, as we have done in previous sections. However, there were certain limitations we encountered.

Firstly, it's important to note that PETSc's GPU implementation lacks support for ILU preconditioning when utilizing multiple GPUs. Furthermore, the utilization of the external Hypre library for GPU acceleration is also not possible. These limitations hindered our ability to directly compare PETSc's ILU0 preconditioning with the cuSPARSE library's utilization in POT3D for ILU0 preconditioning (Section 2.3).

Secondly, when considering the use of the Algebraic Multigrid (AMG) preconditioner for a problem size of 133 $\times$ 361 $\times$ 901, as detailed in Section 5.2.1, for GPU runs, we faced feasibility issues with a limited number of GPUs. Even with 16 Tesla V100 (16GB) GPUs in the Cirrus system, AMG preconditioning was insufficient to solve a problem of this magnitude. The primary cause is the substantially higher memory requirement of AMG preconditioning compared to other methods. To demonstrate this, we conducted profiling using the ARM

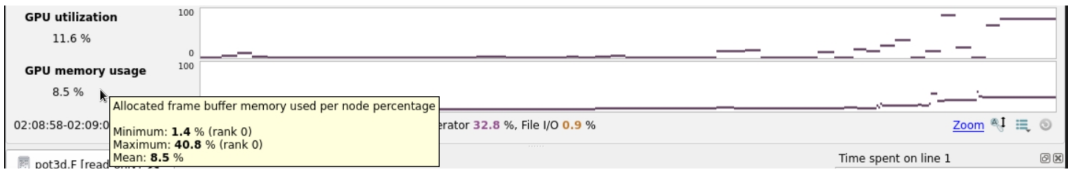
Forge MAP profiler on the PETSc CG solver with two different preconditioners: point-Jacobi (`PCJACOBI`) and AMG (`PCGAMG`). Our analysis employed the smallest available test case in POT3D, denoted as `validation`, with dimensions 63 × 95 × 225, and was executed using a single GPU.



Allocated frame buffer memory used per node percentage

Minimum: **1.4** % (rank 0)
Maximum: **6.0** % (rank 0)
Mean: **4.2** %

Figure 5.10: GPU compute and memory utilization of the PETSc Conjugate Gradient (CG) solver with a point-Jacobi preconditioner were measured while solving a small problem of size 63 × 95 × 225 using a single Tesla V100 (16GB) GPU.



Allocated frame buffer memory used per node percentage

Minimum: **1.4** % (rank 0)
Maximum: **40.8** % (rank 0)
Mean: **8.5** %

Figure 5.11: GPU compute and memory utilization of the PETSc Conjugate Gradient (CG) solver with an Algebraic Multi-Grid (AMG) preconditioner were measured while solving a small problem of size 63 × 95 × 225 using a single Tesla V100 (16GB) GPU.

The profiling results, shown in Figures 5.10 and 5.11, are enlightening. Figure 5.10 illustrates that the solver using `PCJACOBI` requires a mere 6% of GPU memory throughout the run, while Figure 5.11 indicates that the solver employing `PCGAMG` demands up to 40% of GPU memory at its peak. It's crucial to note that the memory demands of `PCGAMG` exceed 100% for the problem size we used for analysis (133 × 361 × 901) while profiling, leading to termination of solver execution with CUDA out-of-memory errors. Even attempting to reduce the problem size to a level manageable by AMG for full GPU utilization results in an extremely brief solver elapsed time (less than one second), rendering it impractical for a meaningful analysis. As a consequence, we were unable to make use of the AMG preconditioner in our study.

5.7 Analysis of residual convergence

We used the remaining options to analyze how well the GPU implementations are converging. We followed the procedure outlined in Section 5.2.1 and created plots shown in figures 5.12 and 5.13. The results indicate that the configurations utilizing the point-Jacobi (`PCJACOBI`) preconditioner exhibit comparable

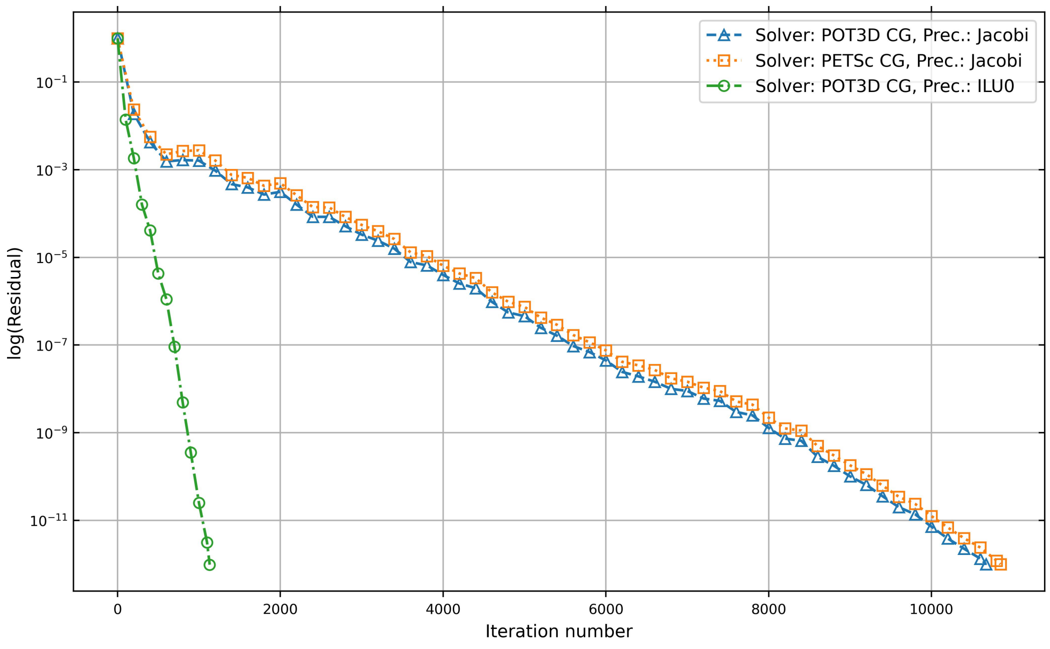


Figure 5.12: Convergence of residual errors for various solver-preconditioner combinations when solving a problem of dimensions 133 $\times$ 361 $\times$ 901 using 8 CPU cores. The residual values are presented in a logarithmic scale.

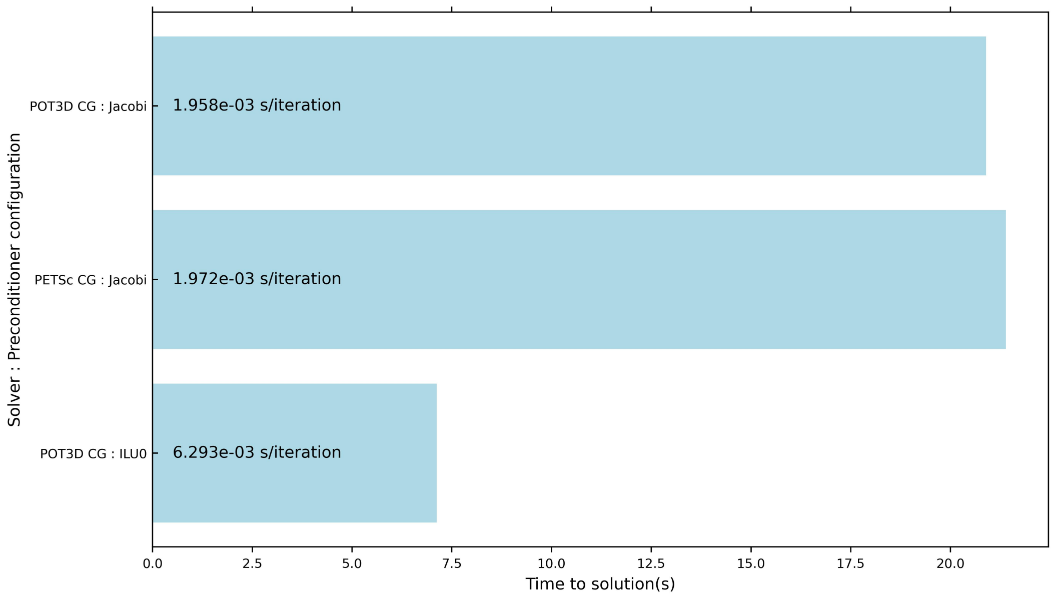


Figure 5.13: Time taken by various solver-preconditioner configurations to solve a problem of dimensions 133 $\times$ 361 $\times$ 901, all using 8 GPUs. Each setup's elapsed time per iteration (seconds/iteration) is labelled on the graph.

performance. Notably, for the initial iterations (see figure 5.12), the residual drop is rapid, then stabilizing to small residual reductions for further iterations.

However, the setup employing the ILU0 preconditioner outperforms the others, making the configuration using the POT3D CG solver with ILU0 preconditioner the most effective choice.

5.8 Strong scaling analysis

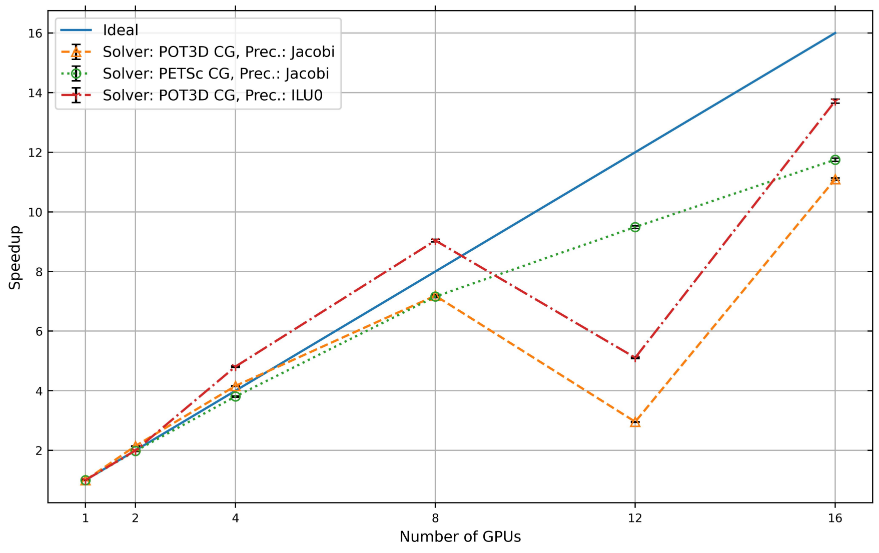


Figure 5.14: Speedup comparison of solver configurations for the POT3D and PETSc implementations on varying GPUs/processes. The problem size is 133 $\times$ 361 $\times$ 901. The error bars represent standard deviations.

We assessed the efficiency of GPU implementations for PETSc and POT3D using the approach outlined in Section 5.2.2, and we depicted the results in Figures 5.14 and 5.15. It's evident that the speedup achieved with the point-Jacobi (`PCJACOBI`) preconditioner's setup is comparable. However, the ILU0 preconditioner's setup offers better performance across various numbers of GPUs, except when using 12 GPUs/processes. The decline in performance observed in the graphs could be attributed to POT3D's custom data decomposition and distribution, resulting in suboptimal load balancing for that specific number of processes and the problem under study.

Furthermore, Figures 5.14 and 5.15 illustrate that the POT3D solver occasionally surpasses the expected performance. This could be attributed to POT3D's customized algorithm, sub-optimal cache utilization, and localization when employing a single GPU. Conversely, PETSc's solver demonstrates consistent performance with smooth curves, showcasing balanced load distribution thanks to

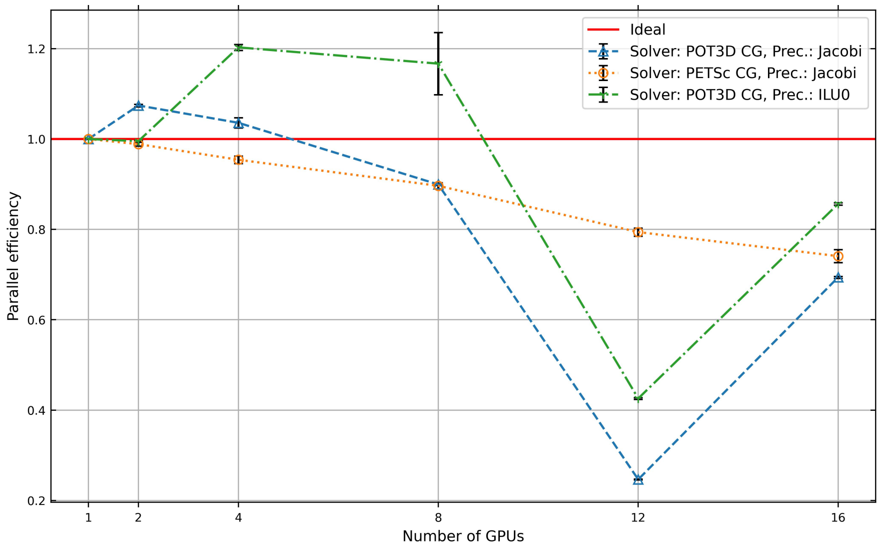


Figure 5.15: Parallel efficiency of solver configurations for the POT3D and PETSc implementations on varying GPUs/processes. The problem size is 133 $\times$ 361 $\times$ 901. The error bars represent standard deviations.

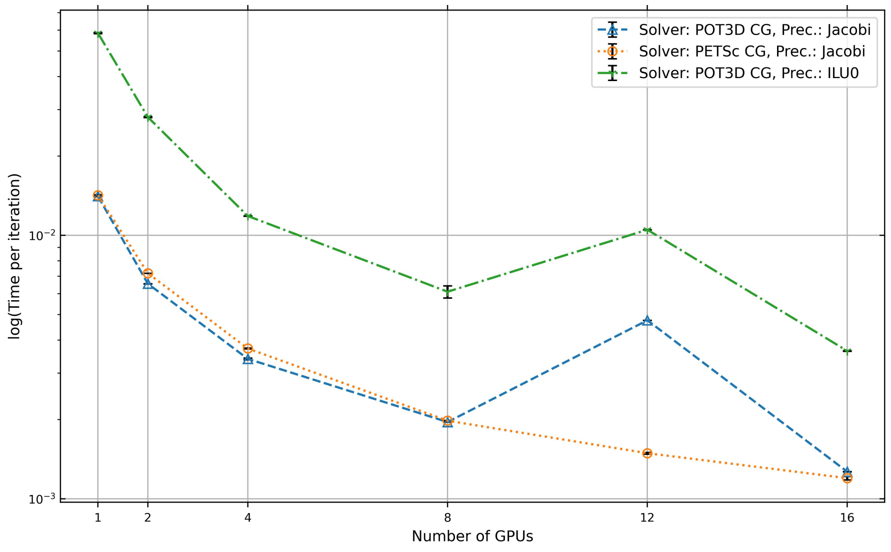


Figure 5.16: Time per iteration of solver configurations for the POT3D and PETSc implementations on varying GPUs. The problem size is 133 $\times$ 361 $\times$ 901. The time per iteration values are presented in a logarithmic scale. The error bars represent standard deviations.

its abstracted data decomposition and parallel implementation within the generalized PETSc library. Notably, all the setups show consistently good parallel efficiency (greater than 60% for all configurations except when the number of GPUs = 12, shown in figure 5.15), demonstrating that the solvers utilize additional GPUs effectively to solve the given problem.

Figure 5.16 illustrates the time taken per iteration for various configurations when executed concurrently using different numbers of GPUs. The findings align with the information presented in figure 5.13 in the majority of instances. However, the POT3D setups exhibit notably prolonged durations when employing 12 GPUs. This divergence might stem from suboptimal custom load distribution within the POT3D framework for specific problem dimensions. In contrast, the PETSc setup demonstrates a consistent and gradual decrease in processing time.

5.9 Analyzing the implementation effort and utility of the PETSc library

In this chapter, our analysis employed various solvers and preconditioners using PETSc within the same program. To switch solvers, we simply adjusted a parameter in the `KSPSetType` function, and likewise for preconditioners using `PCSetType` function. Moreover, PETSc offers specialized functions for solvers and preconditioners, enabling further optimization and control.

The PETSc library's abstraction level facilitated swift performance comparisons for different POT3D application setups. It directed our focus towards data preparation and representation, rather than delving into intricate parallelism within solver implementation. Had we attempted a similar analysis directly within the native POT3D application, manual implementation of solvers and preconditioners would have been necessary. This would have consumed significantly more time than what was available for the dissertation.

This highlights the utility of a library like PETSc during the initial stages of custom solver development, such as POT3D. It allows concentrating on data preparation and assessing optimal solver-preconditioner combinations. Subsequently, customized versions of these components can be implemented for precise problem-solving, as done in the POT3D application.

Chapter 6

Conclusions

6.1 Main takeaways and results

In this study, we conducted a comprehensive analysis of the performance, convergence, and efficiency of the serial PETSc implementation and the POT3D application, both in serial and parallel modes. Our investigation delved into various solver-preconditioner configurations, shedding light on their suitability for potential field computations. Through this analysis, we gained valuable insights that contribute to a holistic understanding of the strengths and limitations of these implementations.

Our study commenced with a comparison of the serial PETSc implementation and the POT3D application executed in a single MPI process. This approach allowed us to encompass intrinsic PETSc preconditioners, notably the Incomplete LU factorization (ILU0) preconditioner. Our findings emphasized the effectiveness of the ILU0 preconditioner across both implementations, particularly highlighting its suitability for sequential execution scenarios. Further examination of the convergence of residual errors offered deeper insights into the solver-preconditioner combinations' behaviours. The POT3D CG solver with ILU0 preconditioning consistently exhibited superior convergence rates, showcasing its potential for achieving accurate solutions efficiently. While the Algebraic Multigrid (AMG) preconditioner demonstrated rapid convergence, specifically for the potential field problem solved using POT3D. However, its trade-off in terms of longer per-iteration times and increased memory demands tempered its overall effectiveness, especially for larger problems.

Transitioning into parallel CPU-only mode, we evaluated the implementations' performance with multiple processing units. The POT3D application's custom optimizations led to superior speedup and efficiency compared to the PETSc library. Notably, the POT3D CG solver with ILU0 preconditioning emerged as the most efficient choice, aligning with our earlier findings.

In the context of GPU implementations, our analysis underscored the POT3D application's edge, albeit with constraints imposed by GPU memory limitations. The ILU0 preconditioner again demonstrated its efficiency, showcasing its versatility across varied scenarios. Although the point-Jacobi preconditioner produced competitive results, the ILU0 preconditioner consistently outperformed it in terms of both convergence and runtime performance for the potential field problem solved in POT3D.

It should be noted that the conjugate gradient (CG) solver performed well for this specific potential field problem, utilizing a symmetric positive definite matrix in comparison to the other solvers investigated in our study. However, it is important to recognize that this outcome might be specific to the current cases under consideration; therefore, the observed results here cannot be generalized as a universal conclusion.

As a noteworthy takeaway, the PETSc library's abstraction facilitated rapid experimentation with solver-preconditioner configurations, guiding the selection of optimal setups. This utility underscores the importance of libraries in easing the initial stages of custom solver development, as evident from our comparative analysis.

In summary, our research provides a comprehensive understanding of the performance dynamics between the POT3D application and the PETSc library for potential field computations. While PETSc offers a versatile framework for solver experimentation, the customized POT3D implementation, equipped with tailored solvers and preconditioners, emerges as the superior choice in terms of both efficiency and convergence. The insights garnered in this study contribute to the broader knowledge landscape of solver-preconditioner dynamics in computational simulations and will serve as a guide for future advancements in this domain.

6.2 Reflection on results and project goals

The primary goal of this project was to seamlessly integrate PETSc preconditioners and solvers into the POT3D application, with a specific focus on systems utilizing only CPUs. Additionally, our objective was to gather benchmarking data for a thorough analysis. Our efforts have provided valuable insights into the performance and behaviour of different solver-preconditioner configurations. We have effectively accomplished the integration of PETSc solvers within POT3D, thereby fulfilling the main aim of the project. Through our analysis, we have obtained significant information regarding convergence, runtime distribution, and the efficiency of various solver-preconditioner combinations, thus meeting another aspect of the project's goals aimed at deriving conclusions from the observed data.

With the early attainment of our project objectives, we have identified promising avenues for expansion. These possibilities include investigating the performance impact of PETSc solvers when paired with GPU acceleration, exploring the influence of diverse preconditioners on system performance, and evaluating the integration of alternative solvers into the POT3D application. We have effectively executed these additional tasks and presented the corresponding outcomes.

6.3 Future work

In light of the observed divergence issue in the PETSc solver when applied to the potential field current sheet (PFCS) option (Section 4.4.1), a potential avenue for future research involves a deeper analysis of this phenomenon. This could entail investigating alternative solver configurations, preconditioning strategies, and numerical techniques to address the convergence challenges encountered. Additionally, exploring the impact of different boundary coefficient treatments on solver performance could offer valuable insights. Furthermore, the usage of standardized test cases specifically tailored for the PFCS models within POT3D could facilitate a more comprehensive evaluation of solver behaviour and validity.

Bibliography

[1] R. M. Caplan, C. Downs, J. A. Linker, and Z. Mikic, “Variations in finite-difference potential fields,” *The Astrophysical Journal*, vol. 915, no. 1, p. 44, Jul. 2021. DOI: 10.3847/1538-4357/abfd2f. [Online]. Available: https://doi.org/10.3847%2F1538-4357%2Fabfd2f.

[2] R. M. Caplan, Z. Mikic, and J. A. Linker, *From mpi to mpi+openacc: Conversion of a legacy fortran pcg solver for the spherical laplace equation*, 2017. arXiv: 1709.01126 [cs.MS].

[3] J. T. Hoeksema, J. M. Wilcox, and P. H. Scherrer, “The structure of the heliospheric current sheet: 1978–1982,” *Journal of Geophysical Research*, vol. 88, no. A12, p. 9910, 1983. DOI: 10.1029/ja088ia12p09910.

[4] C. J. Schrijver, A. W. Sandman, M. J. Aschwanden, and M. L. DeRosa, “The coronal heating mechanism as identified by full-sun visualizations,” *The Astrophysical Journal*, vol. 615, no. 1, pp. 512–525, 2004. DOI: 10.1086/424028.

[5] S. K. Antiochos, C. R. DeVore, J. T. Karpen, and Z. Mikić, “Structure and Dynamics of the Sun’s Open Magnetic Field,”, vol. 671, no. 1, pp. 936–946, Dec. 2007. DOI: 10.1086/522489. arXiv: 0705.4430 [astro-ph].

[6] J. W. Harvey, F. Hill, R. P. Hubbard, *et al.*, “The global oscillation network group (GONG) project,” *Science*, vol. 272, no. 5266, pp. 1284–1286, May 1996. DOI: 10.1126/science.272.5266.1284. [Online]. Available: https://doi.org/10.1126/science.272.5266.1284.

[7] P. H. Scherrer, J. Schou, R. I. Bush, *et al.*, “The helioseismic and magnetic imager (HMI) investigation for the solar dynamics observatory (SDO),” *Solar Physics*, vol. 275, no. 1-2, pp. 207–227, Oct. 2011. DOI: 10.1007/s11207-011-9834-2. [Online]. Available: https://doi.org/10.1007/s11207-011-9834-2.

[8] C. N. Arge, J. G. Luhmann, D. Odstrcil, C. J. Schrijver, and Y. Li, “Stream structure and coronal sources of the solar wind during the May 12th, 1997 CME,” *Journal of Atmospheric and Solar-Terrestrial Physics*, vol. 66, no. 15-16, pp. 1295–1309, Oct. 2004. DOI: 10.1016/j.jastp.2004.03.018.

[9] *Home — ccmc.gsfc.nasa.gov*, https://ccmc.gsfc.nasa.gov/.

[10] *Student cluster competition*. [Online]. Available: https://www.isc-hpc.com/student-cluster-competition.html.

[11] *Petsc overview*. [Online]. Available: https://petsc.org/release/overview/.

[12] *Linear solvers and preconditioners*. [Online]. Available: https://petsc.org/release/manualpages/LinearSolvers/.

[13] A. E. P. Veldman and K. Rinzema, "Playing with nonuniform grids," *Journal of Engineering Mathematics*, vol. 26, no. 1, pp. 119–130, Feb. 1992. DOI: 10.1007/BF00043231.

[14] N. Bell and M. Garland, "Efficient sparse matrix-vector multiplication on cuda," Nvidia Technical Report NVR-2008-004, Nvidia Corporation, Tech. Rep., 2008.

[15] R. Barrett, M. Berry, T. F. Chan, *et al.*, *Templates for the solution of linear systems: building blocks for iterative methods*. SIAM, 1994.

[16] J. R. Shewchuk *et al.*, *An introduction to the conjugate gradient method without the agonizing pain*, 1994.

[17] Y. Saad, *Iterative methods for sparse linear systems*. SIAM, 2003.

[18] Jul. 2023. [Online]. Available: https://docs.nvidia.com/cuda/cusparse/index.html.

[19] L. Yuan, Y. Zhang, X. Sun, and T. Wang, "Optimizing sparse matrix vector multiplication using diagonal storage matrix format," in *2010 IEEE 12th International Conference on High Performance Computing and Communications (HPCC)*, 2010, pp. 585–590. DOI: 10.1109/HPCC.2010.67.

[20] J. L. Greathouse and M. Daga, "Efficient sparse matrix-vector multiplication on gpus using the csr storage format," in *SC '14: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, 2014, pp. 769–780. DOI: 10.1109/SC.2014.68.

[21] B. Smith and H. Zhang, "Sparse triangular solves for ilu revisited: Data layout crucial to better performance," *The International Journal of High Performance Computing Applications*, vol. 25, no. 4, pp. 386–391, 2011. DOI: 10.1177/1094342010389857. eprint: https://doi.org/10.1177/1094342010389857. [Online]. Available: https://doi.org/10.1177/1094342010389857.

[22] *Conjugate gradient method*. [Online]. Available: https://en.wikipedia.org/wiki/Conjugate_gradient_method.

[23] *Petsc website*. [Online]. Available: https://petsc.org/release/.

[24] 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 for scientific computing*, Springer, 1997, pp. 163–202.

[25] S. Balay, S. Abhyankar, M. Adams, *et al.*, *Petsc users manual (tech. rep. no. anl-95/11—revision 3.12). argonne national laboratory*, 2019.

[26] S. T. Mukhambetzhanov, D. V. Lebedev, N. M. Kassymbek, T. S. Imankulov, B. Matkerim, and D. Z. Akhmed-Zaki, "Gmres based numerical simulation and parallel implementation of multicomponent multiphase flow in porous media," *Cogent Engineering*, vol. 7, no. 1, D. Pham, Ed., p. 1 785 189,

2020. DOI: 10.1080/23311916.2020.1785189. eprint: https://doi.org/10.1080/23311916.2020.1785189. [Online]. Available: https://doi.org/10.1080/23311916.2020.1785189.

[27] L. Ge and F. Sotiropoulos, "A numerical method for solving the 3d unsteady incompressible navier–stokes equations in curvilinear domains with complex immersed boundaries," *Journal of Computational Physics*, vol. 225, no. 2, pp. 1782–1809, 2007, ISSN: 0021-9991. DOI: https://doi.org/10.1016/j.jcp.2007.02.017. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0021999107000873.

[28] N. Aage, E. Andreassen, and B. S. Lazarov, "Topology optimization using petsc: An easy-to-use, fully parallel, open source topology optimization framework," *Structural and Multidisciplinary Optimization*, vol. 51, pp. 565–572, 2015.

[29] G. Seemann, F. Sachse, M. Karl, D. Weiss, V. Heuveline, and O. Dössel, "Framework for modular, flexible and efficient solving the cardiac bidomain equations using petsc," *Progress in industrial mathematics at ECMI 2008*, pp. 363–369, 2010.

[30] A. Logg, K.-A. Mardal, and G. Wells, *Automated solution of differential equations by the finite element method The fenics book*. Springer Berlin, 2016.

[31] *Petsc c/fortran api*. [Online]. Available: https://petsc.org/release/manualpages.

[32] *Petsc for fortran users*. [Online]. Available: https://petsc.org/release/manual/fortran/.

[33] *Summary of sparse linear solvers available in petsc*. [Online]. Available: https://petsc.org/main/overview/linear_solve_table/.

[34] *Petsc preconditioners (pc)*. [Online]. Available: https://petsc.org/release/manualpages/PC/.

[35] P. D'Ambra, F. Durastante, and S. Filippone, "AMG preconditioners for linear solvers towards extreme scale," *SIAM Journal on Scientific Computing*, vol. 43, no. 5, S679–S703, Jan. 2021. DOI: 10.1137/20m134914x. [Online]. Available: https://doi.org/10.1137/20m134914x.

[36] Y. Saad and M. H. Schultz, "Gmres: A generalized minimal residual algorithm for solving nonsymmetric linear systems," *SIAM Journal on scientific and statistical computing*, vol. 7, no. 3, pp. 856–869, 1986.

[37] H. A. van der Vorst, "Bi-CGSTAB: A fast and smoothly converging variant of bi-CG for the solution of nonsymmetric linear systems," *SIAM Journal on Scientific and Statistical Computing*, vol. 13, no. 2, pp. 631–644, Mar. 1992. DOI: 10.1137/0913035. [Online]. Available: https://doi.org/10.1137/0913035.

[38] N. Zhang, Z. Rong, Y. Chen, S. Sun, and J. Hu, "Hierarchical LU direct solver based on higher order basis function," in *2021 International Applied Computational Electromagnetics Society (ACES-China) Sympo-*

sium, IEEE, Jul. 2021. DOI: 10.23919/aces-china52398.2021.9581757. [Online]. Available: https://doi.org/10.23919/aces-china52398.2021.9581757.

[39] *Petsc: Gpu support roadmap*. [Online]. Available: https://petsc.org/release/overview/gpu_roadmap/.

[40] V. Minden, B. Smith, and M. G. Knepley, "Preliminary implementation of petsc using gpus," *GPU solutions to multi-scale problems in science and engineering*, pp. 131–140, 2013.

[41] S. Cuomo, A. Galletti, G. Giunta, and L. Marcellino, "Toward a multi-level parallel framework on gpu cluster with petsc-cuda for pde-based optical flow computation," *Procedia Computer Science*, vol. 51, pp. 170–179, 2015.

[42] P. Kumbhar, "Performance of petsc gpu implementation with sparse matrix storage schemes," 2011.

[43] *Pot3d: High performance potential field solver*. [Online]. Available: https://github.com/predsci/POT3D.

[44] *Hpcg benchmark*. [Online]. Available: https://hpcg-benchmark.org/index.html.

[45] *Hpl - a portable implementation of the high-performance linpack benchmark for distributed-memory computers*. [Online]. Available: https://netlib.org/benchmark/hpl/.

[46] *Hpc challenge benchmark*. [Online]. Available: https://hpcchallenge.org/hpcc/.

[47] *Flutas (fluid transport accelerated solver)*. [Online]. Available: https://github.com/Multiphysics-Flow-Solvers/FluTAS.

[48] *Quantum espresso*. [Online]. Available: https://www.quantum-espresso.org/.

[49] *NVIDIA HPC-Benchmarks | NVIDIA NGC — catalog.ngc.nvidia.com*, https://catalog.ngc.nvidia.com/orgs/nvidia/containers/hpc-benchmarks, [Accessed 08-08-2023].

[50] *NVIDIA HPC SDK Version 22.11 Documentation — docs.nvidia.com*, https://docs.nvidia.com/hpc-sdk/archive/22.11/index.html, [Accessed 08-08-2023].

[51] *Unified communication - x framework library*. [Online]. Available: https://docs.nvidia.com/networking/display/HPCXv29/Unified+Communication+-+X+Framework+Library.

[52] *Advanced Computing Facility | EPCC — epcc.ed.ac.uk*, https://www.epcc.ed.ac.uk/hpc-services/advanced-computing-facility, [Accessed 08-08-2023].

[53] R. D. Falgout, J. E. Jones, and U. M. Yang, "The design and implementation of hypre, a library of parallel high performance preconditioners," in *Numerical solution of partial differential equations on parallel computers*, Springer, 2006, pp. 267–294.