Exploiting high-level abstractions to extend capabilities and improve performance of domain-specific languages

By:

BALOGH Gábor Dániel

Thesis Supervisor:

Dr. REGULY István Zoltán PhD

PhD Dissertation

Pázmány Péter Catholic University
Roska Tamás Doctoral School of Sciences and Technology

2023

I would like to dedicate this dissertation to my loving family.

Acknowledgements

First of all, I would like to express my gratitude to my supervisor Dr. István Reguly, for introducing me to the world of High-Performance Computing, providing me with countless opportunities, and for his immense support, guidance, and patience that led me through these years. His positive and supportive attitude helped me a tremendous amount during the past years. I am deeply grateful to Dr. Gihan Mudalige for welcoming me into his research group during my stay in Warwick and for his help and insight during our collaboration. I am also grateful to Jacques du Toit, who generously provided knowledge and expertise. I would like to thank Prof. Uwe Neumann and Dr. Johannes Lotz for the opportunity to learn from them during my stay at RWTH Aachen University.

I would like to thank all my friends and colleagues for filling these past few years with laughter and joy. I would like to thank András Attila Sulyok, Bálint Siklósi, Bence Horváth-Keömley, Tamás Rudner, Mihály Vághy, and many others for the myriad thought-provoking discussions we've shared that have been a constant source of inspiration and growth.

I am thankful to the Pázmány Péter Catholic University, Faculty of Information Technology, especially to Prof. Gábor Szederkényi and Prof. Péter Szolgay for the opportunity to participate in the doctoral program and for supporting me throughout. I am especially thankful to Dr. Tivadarné Vida for her kind help during my doctoral studies.

Finally, I am most grateful to my family for their limitless support and especially to Zsófia Balogh-Lantos for tolerating me and helping me through the difficult times.

Abstract

In the past two decades, we observed a paradigm shift in high-performance computing. Where once frequency scaling and single-threaded performance increase were the main sources of gains in compute throughput for applications, nowadays parallel programming and the use of specialized hardware are required. While large CPU clusters are still and will continue to be important in the computing landscape, GPUs and other accelerator architectures took over the focus for scientists developing large-scale applications. To further increase the difficulty, each platform might require specific programming models, languages, or optimizations to fully utilize the capabilities provided by the hardware. It is not clear which hardware solution will provide the best performance for an application, but the differences in the way to express the computation on different hardware make it infeasible to learn the intricacies of each hardware or port and maintain multiple code bases.

In light of the surging programming models and parallel hardware, research focused on increasing the abstraction level of scientific codes to support multiple hardware platforms without the development cost of multiple implementations. Performance, portability, and productivity became the three key concepts for programming models. Without performance, computing results for applications with ever-increasing computational needs simply take too much time. While specialized code for a specific hardware will produce the best performance, it requires significant coding effort and makes adopting new platforms difficult. The ideal programming model lets the scientists focus on the science and express computations close to the scientific model without concerns for the utilized hardware while getting good performance on multiple hardware families.

Aiming to address these problems, Domain-Specific languages (DSLs) and high-level abstractions arose to help scientists through high-level abstractions that encapsulate hardware details while expressing problems naturally. Using domain-specific abstractions enables the libraries to constrict computations in a way that enables them to take advantage of high-level concepts to make low-level optimizations for the target hardware automatically. This approach lets domain scientists express their computations once and get high-performance and future-proof implementations for their applications.

This dissertation explores techniques for improving the capabilities and performance of two DSLs from the Oxford Parallel family. OP2 and OPS aim to express computations as a series of loops with high-level descriptions of the data access patterns for parallelization. From this high-level description, the DSL will generate target-specific optimized implementations for the loops. This allows the developer to support multiple hardware from a single source and get good performance due to the generated low-level implementations without any additional effort to support each platform. OP2 supports computations on unstructured meshes, while OPS uses structured grids and stencil-like data accesses.

In the first part of my research, I study unstructured mesh computations and their mapping to parallel hardware. My research focuses on improving the code generation used by the DSL in two main regards. The code generation of DSLs quickly becomes complex, and writing extensions or new optimizations is proved to be error-prone. I introduced a parallelization skeleton-based approach for code generation, which drastically reduces the amount of code actually generated by the library with the main structure of the loops outlined in the skeleton and only replaces loop-specific patches during code generation. Another main concern for source-to-source translation is the integration with industrial build systems. I improved the robustness of the code generation library, moving it from a text-based Python implementation into clang's compiler infrastructure, taking advantage of the compiler's representation of the source code.

The second part of my research focused on linear solvers for applications using the Alternating Direction Implicit (ADI) method. My goal was to provide scalable solver algorithms for batch-tridiagonal systems for large-scale HPC clusters. I explore solver algorithms and the algorithmic trade-offs required at increasing machine scale. In an ADI application, the solution of multiple batches of 1D problems is used to approximate the solution of higher-dimensional problems. To solve hundreds of batch-tridiagonal problems for each time step, scalable solver implementations are essential. I introduce a new exact, iterative distributed solver algorithm supporting all common memory layouts used in ADI applications. I evaluate the performance of the best implementations that are used to solve a number of large-scale problems on CPU and GPU clusters. With the integration to the OPS library, we extend the supported domain, enabling direct support for ADI applications in OPS.

Finally, I focused on extending OPS with Adjoint-mode (or reverse-mode) Algorithmic Differentiation support. Adjoints efficiently compute sensitivity information for computer programs building on the chain rule. I address two challenges that are present for reverse-mode algorithmic differentiation. AD tools often have large memory overhead introduced by caching intermediate states and the control flow graph in a structure called tape. I show the advantages of using a domain-specific library and the domain restrictions to compute derivatives of large stencil computations on CPU and GPU platforms. Using a loop-level adjoint tape (built by the generated code) to follow the computational steps drastically decreases the memory requirements of storing this information compared to traditional operator-overloading-based tools. Secondly, the reversal of the control flow during derivative propagation makes parallelization a complex problem. I show that the code generation of DSLs, in combination with the high-level description of parallelism in the original code, is a great tool to provide performance-portable implementations for adjoint codes to support multiple hardware from the same source.

Kivonat

Az elmúlt két évtizedben paradigmaváltást figyeltünk meg a nagy-teljesítményű számítástechnikában. Ahol egykor a frekvenciaskálázás és az egyszálas teljesítménynövelés volt az alkalmazások számítási teljesítményény növelésének fő módszere, most párhuzamos programozás és speciális hardverek használata az út. Míg a nagy CPU-klaszterek továbbra is és a jövőben is fontosak lesznek a HPC területén, a GPU-k és más gyorsítók átvették a fókuszt a nagyszabású alkalmazásokat fejlesztő tudósok számára. A helyzetet tovább nehezíti, hogy minden platformon speciális programozási modellekre, nyelvekre vagy optimalizációkra lehet szükség a hardver által biztosított képességek teljes kihasználásához. Nem egyértelmű, hogy melyik hardver fogja a legjobb teljesítményt nyújtani egy alkalmazás számára, de az ezeken a hardvereken használt számítási modellek különbségeinek köszönhetően az összes hardver támogatása portolás vagy több kódbázis párhuzamos fejlesztésével fenntarthatatlanná vált.

Az újonnan születő programozási modellek és a párhuzamos architektúrák fényében a kutatók a tudományos kód absztrakciós szintjének növelésére összpontosítottak, hogy több hardver támogatása extra fejlesztési költség nélkül lehetővé váljon. A teljesítmény, a hordozhatóság és a produktivitás a programozási modellek három kulcsfogalma lett. Az egyre növekvő számítási igényű alkalmazások eredményeinek jó teljesítmény nélküli számítása egyszerűen túl sok időt vesz igénybe. Ezzel szemben egy adott hardveren a speciálizált kód biztosítja a legjobb teljesítményt, ez azonban jelentős kódolási erőfeszítést igényel, és megnehezíti az új platformok támogatását. Az ideális programozási modell lehetővé teszi a tudósok számára, hogy a tudományra összpontosítsanak, és a tudományos modellhez közeli módon fejezzék ki a számításokat anélkül, hogy a felhasznált hardverrel foglalkoznának, majd az így kapott modellből több különböző hardveren is jó teljesítményt érjenek el.

E problémák megoldása érdekében a domén-specifikus nyelvek (DSL) olyan magas szintű absztrakciókon keresztül segítik a tudósokat, amelyek elfedik a hardverre vonatkozó részleteket, miközben a tudósok természetes módon fejezhetik ki a problémákat. A domén-specifikus absztrakciók használata lehetővé teszi a könyvtárak számára, hogy a számítások típusát oly módon szorítsák meg, hogy a magas szintű fogalmak és absztrakciók előnyeit kihasználva automatikusan a célhardverre optimalizált alacsony szintű implementációt adhassanak. Ez a megközelítés lehetővé teszi a tudósok számára, hogy a számításaikat egyszer, egy magas szintű modellben, kifejezve olyan alkalmazás kódot kapjanak, ami több hardveren is jó teljesítményt nyújt és potenciálisan támogatja a jövőben használt hardvereket is.

A disszertációm az Oxford Parallel domén-specifikus nyelvcsalád két DSL-jének kiegészítésére és javítására szolgáló technikákat vizsgál. Az OP2 és az OPS célja, hogy a számításokat ciklusok sorozataként fejezze ki az adathozzáférési minták magas szintű leírásával a párhuzamosításhoz. Ebből a magas szintű leírásból a DSL optimalizált implementációkat generál a számítási kernelek számára. Ez lehetővé teszi a fejlesztő számára, hogy több hardvert támogasson egyetlen forráskódból, és jó teljesítményt érjen el a generált alacsony szintű megvalósításoknak köszönhetően anélkül, hogy az egyes platformokra explicit módon írna kódot. Az OP2 a strukturálatlan hálókon végzett számítások támogatására, míg az OPS strukturált rácsokon értelmezett számítások támogatására ad lehetőséget.

Kutatásom első részében strukturálatlan hálókon értelmezett számítások és azok párhuzamos hardverekre való leképezésének módszerét vizsgáltam. Kutatásom a DSL által használt kódgenerálás fejlesztésére irányul. A DSL-ek kódgenerálási lépése gyorsan bonyolulttá válik, és azok kiegészítése nehéz feladat. Ezt a problémát célozva egy párhuzamosítási váz alapú megközelítést írtam le, amely drasztikusan csökkenti a könyvtár által ténylegesen generált kód mennyiségét. A ciklusok szerkezetét a vázból nyerve, csak a ciklus-specifikus részleteket helyettesíti be a kódgenerálás során.

Kutatásom második része a Alternating Direction Implicit módszert alkalmazó alkalmazások lineáris megoldóira fókuszált. Célom az volt, hogy skálázható megoldó algoritmusokat biztosítsak batch-tridiagonális rendszerekhez nagyméretű HPC klaszterekhez. Egy ADI-alkalmazásban az 1D-s problémák batch-elt megoldását használják a magasabb dimenziós problémák megoldásának közelítésére. Időlépésenként több száz batch-tridiagonális rendszer megoldásához elengedhetetlenek a skálázható megoldó algoritmusok. Bevezetek egy új, egzakt, iteratív elosztott megoldó algoritmust, amely támogatja az összes gyakori memóriaelrendezést, ami ADI alkalmazásokban előfordul. Majd a megvalósítások teljesítményét kiértékelem CPU és GPU klasztereken. Az OPS könyvtárba való integrációval kiterjesztjük a DSL által támogatott alkalmazások osztályát, lehetővé téve az ADI alkalmazások direkt támogatását OPS-ben.

Végül az OPS kiterjesztésére összpontosítottam az Adjoint módú (vagy fordított módú) algoritmikus differenciálás támogatásával. Az AAD a láncszabályra építve hatékonyan számítja ki a programokhoz tartozó derivált inormációkat. Két olyan kihívással foglalkozom, amelyek az adjoint módú algoritmikus differenciálásnál lépnek fel. Az AD-eszközök gyakran nagy memória igénnyel rendelkeznek, amelyet a közbülső állapotok elmentése és a control-flow gráf követése okoz, amelyeket egy tape nevű struktúrában tárolnak. Megmutatom, hogy a domén-specifikus könyvtárak és az általuk használt absztrakciók és megkötések a milyen előnyökkel járnak stencil kódok esetén. A (generált kód által épített) számítási ciklus szintű adjoint tape használata a számítási lépések követésére drasztikusan csökkenti ezen információk tárolásának memóriaigényét a hagyományos operátor túlterhelés alapú eszközökhöz képest. Másodszor, a control flow megfordítása a derivált propagálás során kihívást állít a párhuzamos programok esetén. Megmutatom, hogy a DSL-ek kódgenerálása az eredeti kódban a párhuzamosság magas szintű leírásával kombinálva ígéretes eszköz a teljesítmény-hordozható megvalósítások biztosításához adjoint kódokhoz, amelyek több hardvert is támogatnak ugyanabból a forráskódból.

Contents

Glossary

AAD Adjoint-mode Algorithmic Differentiation.

AD Algorithmic (Automatic) Differentiation.

ADI Alternating Direction Implicit.

AoS Array of Structures.

API Application Programming Interface.

AST Abstract Syntax Tree.

CDE Convection-Diffusion Equation.

CFD Computational Fluid Dynamic.

CPU Central Processing Unit.

CR Cyclic Reduction.

CUDA Compute Unified Device Architecture.

DSL Domain-Specific Language.

eDSL Embedded Domain-Specific Language.

FPGA Field Programmable Gate Array.

GPU Graphical Processing Unit.

HPC High Performance Computing.

IR Intermediate Representation.

LLVM Low Level Virtual Machine.

MPI Message Passing Interface.

NUMA Non-Uniform Memory Access.

OP-DSL Oxford Parallel Domain-Specific Languages.

OP2	Oxford Parallel Library for Unstructured mesh solvers.
OPS	Oxford Parallel Library for Structured mesh solvers.
PCR	Parallel Cyclic Reduction.
PDE	Partial Differential Equation.
SIMD	Single Instruction Multiple Data.
SIMT	Single Instruction Multiple Thread.
SM	Streaming Multiprocessors.
SoA	Structure of Arrays.

List of Figures

List of Tables

List of Listings

List of Algorithms

1 Introduction

The relentless pace of innovation in computing technology has profoundly impacted how programmers and scientists can achieve high performance. While twenty years ago, a few gigaflops might reside within a single CPU, today's most powerful supercomputers deliver trillions of calculations per second across hundreds of thousands of hardware threads. Keeping pace with this exponential growth has required fundamental changes in both hardware and software.

For decades, compute performance has been steadily increasing due to hardware effects described by Dennard scaling and Moore's law. Proposed in the 1970s, Dennard scaling observed that as transistor feature sizes shrunk by Moore's law, keeping the power supply voltage constant caused power and heat densities to remain constant. This meant CPU clock frequencies and throughput could approximately double every 18 months as the number of transistors per unit area doubled. Such regular and exponential increases drove the widespread adoption of computing across all sectors.

However, from around 2005, it became clear that silicon technology reached its limits, and Dennard scaling broke down. Shrinking chip features without a proportional rise in power consumption was rendered infeasible. Performance could no longer freely scale at constant power, and economy of scale diminished.

While still exponential, Moore's law transistor density growth has also flattened in recent manufacturing process nodes. Where feature sizes halved every 24 months in the 1990s, the reduction significantly slowed down today. Many projections foresee outright saturation of Moore's law within this decade. As a result, the computing industry faces a post-Dennard, post-Moore's law era where past exponential increases are simply infeasible without fundamentally new microarchitectures. Novel accelerators, including Graphical Processing Units (GPUs), Field Programmable Gate Arrays (FPGAs), and even quantum and neuromorphic platforms emerge in step.

In the early 2000s, performance revolved around optimizing for single superscalar Central Processing Units (CPUs). Programmers leveraged techniques like loop unrolling, prefetching, and blocking to minimize latency and maximize instruction throughput. However, with the breakdown of Dennard scaling, multi-core architectures became necessary to extract further throughput gains within the existing power budgets. With the emergence of multi-core chips, performance needs demanded algorithms designed for concurrent execution. This paradigm shift posed software challenges around parallel programmability that remain a work in progress today. Shared memory and locks became tantamount, necessitating careful orchestration to avoid contention bottlenecks.

As problem sizes and computational demands grew exponentially. Traditional CPU-based architectures began to struggle with the complexity and scale of modern applications, leading to the widespread adoption of large, distributed clusters comprising both CPUs and GPUs. These hybrid architectures allowed supercomputers to harness the strengths of both types of processors – CPUs for general-purpose tasks and GPUs for highly parallel workloads – enabling more efficient execution of massive computations.

However, these pre-exascale supercomputers faced several challenges. Distributed computing introduced issues of communication latency and data movement between nodes, which were critical bottlenecks. In large, distributed systems, communication overhead can severely degrade performance if not managed efficiently. The time and energy cost of moving data between nodes becomes a critical bottleneck, particularly in applications that require frequent data exchanges. Communication latency can limit the overall speedup achieved from parallelization, diminishing the expected gains of adding more processors. The wide adoption of hybrid architectures and accelerators ads an additional layer of communication between the host and the accelerator memory space, which further increase the importance of hiding the cost of data movement. Additionally, hybrid systems demanded new algorithmic approaches tailored to both CPU and GPU architectures, as many algorithms needed to be re-engineered to efficiently exploit the parallelism offered by GPUs while balancing the workload across CPUs. As supercomputers grew in scale and the computational throughput of the GPUs and CPUs increased having data ready near these processors always became a key challenge for sustaining high computational throughput.

For HPC, these realities necessitated a strategic evolution. The dawning exascale era builds on computational power across hundreds of thousands of concurrent threads. Hardware specialization is inevitable - general-purpose designs struggle to meet Exascale's constraints. Programming such systems demands refactoring algorithms into maximally concurrent forms, preferably with leverage from higher levels of abstraction.

1.1 Parallel architectures in HPC

CPUs have historically played a critical role in this arena, undergoing significant transformations to adapt to the evolving demands of HPC applications. In the early 2000s, enhancing single-core CPU performance was paramount. Each new CPU generation brought substantial performance gains through increased clock speeds and other architectural advancements. However, the end of Dennard scaling around the mid-2000s marked a paradigm shift in CPU design. Challenges such as excessive heat dissipation and power consumption limitations led to a halt in the continuous increase of clock speeds. To further increase single thread performance, modern CPUs use advanced caching mechanisms and complex instruction sets and logic, increasing vector sizes with the use of Single Instruction Multiple Data (SIMD) model where the instructions are executed on a vector of data instead of scalar effectively multiplying the number of operations executed per cycle. However, the single-thread performance still could not keep up with the increasing computational demands. Instead, chip manufacturers pivoted towards a multi-core CPU architecture, wherein a single chip housed multiple processor cores, enhancing performance by enabling parallel processing.

Contemporary high-end server CPUs integrate numerous advanced features to optimize performance, energy efficiency, and parallel processing capabilities. Such CPUs can have up to 64 cores, each supporting two threads, increasing vector unit sizes up to 512-bit wide vectors. To fully take advantage of these CPUs' compute capabilities and avoid starvation due to memory latency, modern CPUs have multiple levels of cache memories with large L3 caches up to 256 MB.

Modern HPC systems frequently deploy multi-socket CPU configurations to bolster further performance, wherein multiple CPUs are installed on a single motherboard. Despite the benefits, this approach introduces Non-Uniform Memory Access (NUMA) issues. In NUMA architectures, memory access time depends on the memory location relative to a processor, leading to variability in performance. The memory allocated by a thread will be located on the closest memory chips directly connected to the socket hosting the thread. Accessing this memory from another socket or NUMA region results in increased latency. Efficiently managing these NUMA issues is crucial for ensuring optimal performance in HPC environments.

Multiple programming abstractions support parallelism for CPU architectures. OpenMP is the most commonly used model for shared memory parallelism, allowing developers to write parallel code for shared memory systems quickly and efficiently. Using compiler directives, OpenMP can automatically handle thread creation, synchronization, and workload distribution among threads. Alternatively, applications can consider each core as a standalone process and use Message Passing Interface (MPI) to handle communications between the processes. In this distributed memory paradigm, individual processes operate independently and exchange data via message passing. This model is adept for large-scale parallelism across many CPUs or even across different machines in a cluster. Additionally, hybrid models combining OpenMP and MPI are employed to exploit parallelism at multiple levels, maximizing computational efficiency by utilizing both thread-level and process-level parallelism.

The last ten years have seen the widespread adoption of GPUs by the HPC community. They offer higher performance and efficiency for a wide range of highly parallel workloads. Initially designed to accelerate graphics rendering, GPUs have emerged as powerful accelerators for a wide range of computationally intensive tasks. Their pivotal role in HPC stems from their architectural and operational characteristics, tailored to handle a high degree of parallelism.

Unlike the traditional CPU that features a limited number of cores optimized for single-threaded performance with sophisticated scheduling, a GPU houses thousands of smaller, more efficient cores designed for multi-threaded performance with simpler scheduling capabilities. This architectural distinction makes GPUs especially well-suited for tasks that can be parallelized, where large blocks of data can be processed simultaneously.

A single GPU may contain more than 100 Streaming Multiprocessors (SM), each with up to 128 CUDA cores executing instructions in parallel. To include such a high amount of cores, GPUs reduce the logic for thread scheduling by running the same instruction in 32 thread groups called warps following the Single Instruction Multiple Thread (SIMT) modell. These warps are grouped into thread blocks bound to one of the SMs and executing the same computation, and finally, a set of blocks executing the same computation grouped together for each kernel. Only threads within a thread block share data and are synchronized. The blocks run entirely independently from each other. The memory hierarchy of GPUs is also a critical consideration. GPUs typically have smaller memory compared to CPUs, albeit the memory size increases with every generation. Modern GPUs only have up to 80 GB of memory, but they compensate with significantly higher memory bandwidth. GPUs have relatively small caches, having up to 50 MB of L2 cache and 256 KB programmable L1 cache shared by the threads within a block. Efficient memory management and data transfer optimization are essential to maximize the performance benefits offered by GPUs.

Programming techniques for GPUs have also evolved significantly. The Compute Unified Device Architecture (CUDA) [1] language extensions to C/C++ and the OpenCL language [2] provide a low-level programming abstraction that gives fine-grained control over GPU architectures. CUDA/OpenCL allows exploiting low-level features like scratch pad memory, warp operations, and block-level synchronization. However, converting existing applications to use CUDA or OpenCL is a substantial undertaking that requires significant effort and considerable changes to the design of the program and the source code. Furthermore, getting good performance can entail detailed work in orchestrating parallelism.

High-level directive-based programming abstractions were introduced to simplify the adoption of GPUs, particularly for existing codes. OpenACC [3], introduced in 2011, was one of the first supporting GPUs. Subsequently, the OpenMP standard introduced support for accelerators starting from version 4 [4], with refinements in 4.5 and 5.0. Of particular note is the evolution of directive-based approaches driven by the acquisition of large US DoE systems such as Titan[1] and the Summit[2], and Sierra[3] systems. To utilize these systems efficiently, existing codes needed to be modified to support GPUs with relative ease. Many of these codes are written in Fortran, and as such, there is now compiler support for writing CUDA, OpenACC, and OpenMP with Fortran in various compilers.

While directive-based programming models make GPU adoption much faster thus increasing the productivity of the developers, it is generally agreed that the best performance can be achieved by using CUDA. It is crucial to understand what potential performance benefits can be achieved in return for more development costs of adopting lower-level programming models. However, the performance difference between CUDA and directive-based approaches varies significantly based on a multitude of factors. These primarily include the type of computation being parallelized, the language being used (C or Fortran), and the compiler.

In conclusion, the differences between the two architecture families create a gap between how to express computations on these hardware. As the computational requirements of HPC applications grow, the trend of hardware specialization to meet these requirements will persist. Developing applications in multiple languages and porting large-scale applications for each new hardware is infeasible. Hence, there is a need for a way to provide uniform APIs and models supporting parallel hardware from a high-level abstraction while still capitalizing on the potential performance of these specialized hardware.

1.2 Motivation for my research

Due to the rapidly changing hardware and programming models that run the most powerful computers in the world, performance portability and productivity became the focus point of any discussion on future-proof high-performance software. In this ever-changing landscape,

[1]https://www.olcf.ornl.gov/olcf-resources/compute-systems/titan/
[2]https://www.olcf.ornl.gov/summit/
[3]https://hpc.llnl.gov/hardware/compute-platforms/sierra

future-proof applications have become synonymous with performance portable applications. The ultimate dream is to support all current and future hardware with the best performance from a single source [5].

On modern hardware, the parallel execution of algorithms is necessary to solve larger and more complicated problems. There are many abstractions to write parallel algorithms with different advantages. MPI, OpenMP, and their combination have been the most common way to target CPUs for a long time. By contrast, the landscape for available frameworks for GPUs is much more complicated, with vendor-specific frameworks providing the highest performance. Maintaining a code base to target all these hardware for an application is infeasible. To solve this problem, general purpose frameworks have arisen in recent years to target all platforms from a single source, such as SYCL [6], KOKKOS [7], [8] or RAJA [9], [10]. SYCL provides a template library to describe general parallel computations on a relatively low level. However, due to the differences in the hardware, this approach achieves portability among hardware but is missing hardware-specific tuning and leaves performance on the table. The other approach is to focus on performance portability using higher-level abstractions, such as in KOKKOS and RAJA, and sacrifice generality in terms of ways to express computations.

1.2.1 Domain-Specific Languages

Domain-Specific Languages (DSLs) or other high-level abstractions are the key approaches for trading generality for performance portability. By sacrificing generality, DSLs can express algorithms using higher-level concepts specific to that domain and have a natural meaning to the application developer. The DSL can take advantage of this meaning and any accompanying restrictions to perform a wide range of optimizations to generate efficient code for the target hardware. Domain-specific languages and declarative programming hold promise, raising abstraction to separate concerns of performance and the description of the computations.

In HPC, DSLs have shown particular promise in addressing two key challenges. Firstly, they facilitate performance portability through architecture-agnostic problem descriptions that abstract away low-level processor details. The high-level descriptions effectively allow the application developer to describe the problem to be computed instead of how the problem should be computed. Secondly, they aid application productivity by raising the level of abstraction for algorithm expression while retaining control over optimization opportunities. Together, these advantages have driven extensive research into DSLs for HPC applications over the past years.

One such domain-specific language family is the Oxford Parallel Domain-Specific Languages, consisting of two active libraries or embedded DSLs OPS [11] and OP2 [12]. Embedded Domain-Specific Languages (eDSLs) are DSLs that are embedded into a host programming language and behave like a traditional library. This approach takes advantage of the capabilities and flexibility of the host language while providing the advantages of the traditional DSL approach to the application where needed. These libraries provide DSL abstractions targeted at Partial Differential Equation (PDE) solvers embedded in C/C++ and Fortran. Domain experts author PDE solvers within these DSLs through high-level parallel loops, expressing computations through element kernels while abstracting away parallel execution details and data movement. During compilation, the libraries use a code generation step to generate target-specific parallel implementations for the parallel loops, applying a wide range of optimizations.

My primary motivation is to research the boundaries and capabilities of the domain-specific language approach and push it to create performance-portable solutions for new areas and problem classes. My research focuses on domain-specific languages for computations on structured and unstructured meshes.

As a first step of my research, I concentrated on increasing the robustness of the code generation step of DSLs, making it more applicable in industrial environments. I aimed to move the complexity of supported programming models from the code generator to parallelization skeletons or templates used to generate the parallel implementations for the applications and, simultaneously, take advantage of using a compiler toolchain to generate the code.

The second half of my research focuses on providing performance portable solutions for new application classes. I worked on two extensions to the OPS DSL. The first area is support for linear solvers for scalable batch-tridiagonal equations for ADI applications, and the second is using OPS to compute sensitivities for the output of OPS applications with Algorithmic Differentiation.

1.3 Oxford Parallel Domain-Specific Languages

Oxford Parallel Domain-Specific Languages (OP-DSL) family[13] contains two eDSL targeting high-level performance portable solutions for two of the 13 dwarfs of high-performance computing [14]: structured and unstructured grid computations. The 13 dwarfs are classes of important application areas such that algorithms in the same class can be implemented differently, and the underlying numerical methods may change over time. However, the claim is that the underlying patterns have persisted through generations of changes and will remain important into the future[14].

With restrictions on the supported class of applications DSLs can move the abstractions to an even higher level and provide better performance and more specialized code compared to general tools. DSLs often use code generation techniques to produce code that includes sophisticated orchestration of parallel executions, such as various coloring strategies (to handle data races) and modifications to the elementary kernel that the high-level application programmer would not have to implement themselves manually.

The OP-DSL family provides abstractions to describe the applications at a parallel loop level with high-level constructs and verbose descriptions of the used data and access patterns. Based on this information, the DSL can generate a target-specific optimized implementation of the parallel loop with drastic optimizations such as loop tiling[11], [15]. The code generation approach starts from a hardware-agnostic high-level code, which allows the developer to write the application code once and use the platform-specific highly efficient implementations for all the supported architectures, improving the productivity of the developer and the performance portability of the created application. Other frameworks targeting a different domain include Devito [16], STELLA [17] (and its successor GridTools) and PSyclone [18].

Oxford Parallel Library for Structured mesh solvers (OPS) and OP2 have similar architecture and Application Programming Interface (API). Figure 1.1 shows the high-level architecture of the DSLs. OPS and OP2 applications are written in C/C++ or Fortran where the DSLs behave

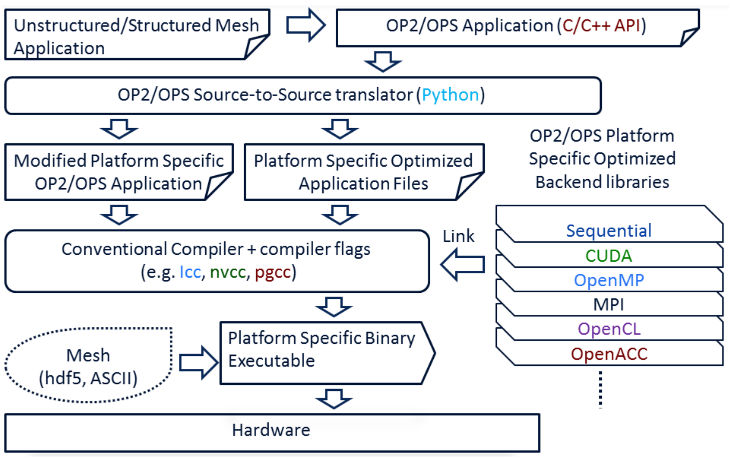

Unstructured/Structured Mesh Application
OP2/OPS Application (C/C++ API)
OP2/OPS Source-to-Source translator (Python)
Modified Platform Specific OP2/OPS Application
Platform Specific Optimized Application Files
OP2/OPS Platform Specific Optimized Backend libraries
Conventional Compiler + compiler flags (e.g. Icc, nvcc, pgcc)
Link
Sequential
CUDA
OpenMP
MPI
OpenCL
OpenACC
Mesh (hdf5, ASCII)
Platform Specific Binary Executable
Hardware

Figure 1.1: The high-level architecture of OP2 and OPS Domain-Specific Languages.

as traditional libraries meaning the OP2/OPS application on the figure is a valid sequential application. The user can compile and run the application, which enables quick and easy development with traditional methods. Both DSLs consider the application as a series of computational loops and use verbose API calls to describe these loops in order to generate target-specific implementations for them, which in turn can be compiled with traditional methods linking to the target-specific backends. The key part of the structure is the source-to-source translation layer of the DSLs. OPS and OP2 parse the C++ application into an intermediate representation containing the description and metadata of the computational loops and relevant data. From this representation, they will generate two main components. The first is a modified application code that is compatible with the different parallel or even distributed memory backends of the DSLs. The second is a set of platform-specific optimized implementations for each computational loop. The modified application files then can be compiled together with the kernel implementations and linked to one of the OP2/OPS platform-specific optimized backend libraries. These backends handle all details about data allocation, movement, communication, scheduling, and other details that are required for performance on the given hardware and configuration. The advantage of this design is the direct support for different hardware, often with widely different programming models, from the same high-level application without compromising on the performance achievable by the low-level approaches of different platforms. Moreover, through the support of communication libraries the OP2/OPS applications can scale on multinode systems or large clusters. The main difference between the two DSLs comes naturally from the common computational patterns used for the two application classes.

1.3.1 OP2 for Unstructured Mesh Solvers

The Oxford Parallel Library for Unstructured mesh solvers (OP2) DSL is the second version of the OPlus library, focusing on automatically parallelizing unstructured mesh computations. The initial version was a traditional software library that supported MPI parallelization, whereas OP2 can be referred to as an “active” library or an embedded DSL. OP2 was among the first high-level abstraction frameworks to apply this approach to production applications [19]. Similar frameworks for unstructured mesh applications include FeniCS [20], Firedrake [21], [22] and

PyFR [23].

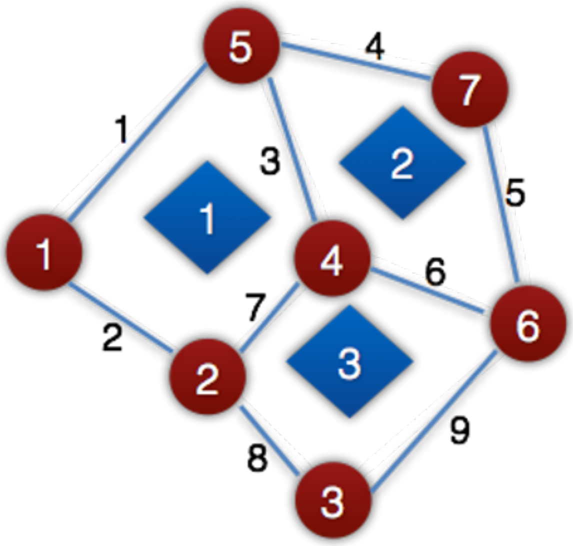


Figure 1.2: Example unstructured grid with cells, edges, and vertices. The key attribute of the unstructured grids is that the connectivity of elements can't be determined solely from the indices, but it requires explicit mappings. In OP2 the user can define data on these mesh elements and the connectivity between the sets through mappings.

The library builds on an API that expresses the computations on unstructured meshes. The abstraction consists of four major components. The first is a mesh made up of a number of sets (such as cells, edges, and vertices), the second is the connections between sets (e.g., an edge is connected to two vertices or cells connected to edges), the third is the data defined on sets (such as coordinates on vertices, or pressure/velocity on a cell center) and finally the fourth is the computations performed on every element of a given set in the mesh. This abstraction enables the expression of various static unstructured meshes. Figure 1.2 shows the declaration of a mesh consisting of three sets, the blue rectangles marking the indices of the cells, the red circles showing the vertices, and the numbers on the lines are the indices of the edges. Note that we cannot give a formula to compute the index of a neighboring element in the grid from the indices. Hence, unstructured grids use explicit mappings to connect neighboring grid elements within the sets and across multiple sets.

A user begins by establishing a mesh and provides all relevant data and metadata to the library through the OP2 API. This API is presented as a conventional software interface integrated into C/C++ or Fortran. From there on, any interaction with the data given to OP2 must be via these API methods. In essence, OP2 creates an internal private replica of the data, allowing it to modify the data structure to optimize performance for the intended platform. After mesh configuration, calculations are expressed as parallel loops on a specific set. The "computational kernel" is applied to each set element, utilizing data accessed either directly from the iteration set or through one level of indirection. The access type is also specified, whether it's read, write, read-write, or associative increment. Algorithms solvable by OP2 are limited to those where the sequence of element execution doesn't influence the final outcome within the machine's precision. Moreover, users can provide mesh-invariant data to the elemental computational kernel and

Listing 1.1 Specification of a mesh, datasets, and an OP2 parallel loop on the edges reading data on the edges and incrementing on the cells of the grid

```
/* ----- elemental kernel function in res.h ------*/
void res(const double *edge,
         double *cell0, double *cell1 ){
  //Computations, such as:
  cell0 += *edge; *cell1 += *edge;
}
/* ---------- in the main program file -----------*/
// Declaring the mesh with OP2
// sets
op_set edges = op_decl_set(numedge, "edges");
op_set cells = op_decl_set(numcell, "cells");
// mappings - connectivity between sets
op_map edge2cell = op_decl_map(
                      edges, cells, 2, etoc_mapdata, "edge2cell");
// data on sets
op_dat p_edge = op_decl_dat(edges, 1, "double", edata, "p_edge");
op_dat p_cell = op_decl_dat(cells, 4, "double", cdata, "p_cell");

// OP2 parallel loop declaration
op_par_loop(res, "res", edges,
  op_arg_dat(p_edge, -1, OP_ID    , 4, "double", OP_READ),
  op_arg_dat(p_cell,  0, edge2cell, 4, "double", OP_INC ),
  op_arg_dat(p_cell,  1, edge2cell, 4, "double", OP_INC));
```

conduct reductions.

The design of these parallel loops was deliberately crafted to support just a few computational and memory access patterns: direct access, indirect reading, and indirect writing or incrementing. This high-level definition grants OP2 the flexibility to parallelize these loops, choosing the most suitable implementation and optimizations for a specific platform. Essentially, this abstraction enables OP2 to produce code that's customized to the situation. The diverse parallelizations and performance outcomes of real-world applications employing OP2 have been documented in prior works, as seen in [19], [24]. These showcase almost peak performance across various architectures, encompassing multi-core CPUs, GPUs, and clusters of both CPUs and GPUs [19], [24]–[28]. The generated parallelization uses an even more extensive range of programming models such as OpenMP, OpenMP4.0, CUDA, and OpenACC, and their combinations with MPI and even simultaneous heterogeneous execution.

A mesh similar to the one shown in Figure 1.2 is defined on OP2 through a series of API calls shown in Listing 1.1. The figure also shows a definition of an OP2 parallel loop using the description of data access used in the loop. In this example, the loop operates on the set of the `edges` in the mesh, performing the same computation per edge (the elemental kernel) defined in the function `res`, reading the data from the edges `p_edge` directly while updating the data for the two neighboring cells, `p_cell`, adjacent to the edge, indirectly using the `edge2cell` mapping. The `op_arg_dat` object holds all the important details of how an `op_dat`'s data is accessed in the loop. The descriptor requires the `op_dat`, followed by its indirection index, the `op_map` used to access the data indirectly, the arity of the data in the `op_dat`, and the type of the data. The final argument is the access mode of the data, read-only, increment, and others (such as read/write and write only).

The `op_par_loop` function encapsulates all the essential details about the computational loop required for parallelization. Consequently, because of this abstraction, the parallelization hinges on just a few parameters. These include the presence of indirectly accessed data or reductions within the loop, in addition to the data access patterns that are conducive to optimizations. OP2 group the computational loops into two main categories direct and indirect loops. A parallel loop is direct if it does not write or increment datasets through indirections. In this case, the loop does not require any synchronization (apart from reductions on scalar values). In the case of indirect loops, the loop will write datasets on indirectly accessed datasets. In most cases, this leads to multiple iterations writing the same values e.g. two cells increment data on the connecting edge. The connectivity of the elements is only known at runtime. OP2 supports multiple solutions to avoid the arising data races. The most common is to use coloring of the grid, and running the elements of different colors separately with synchronization between the colors. The DSL handles the application as a series of loops a single `op_par_loop` holds enough information to parallelize within a loop and the abstraction lets OP2 to follow the data accesses and dependencies between loops at runtime. OP2 generates calls to MPI halo exchanges using `op_mpi_halo_exchanges()` inside the parallel loop implementations to facilitate distributed memory parallelism together with OpenMP. OP2 implements distributed memory parallelization as a classical library in the backend. As the computation requires values from neighboring mesh elements during a distributed memory parallel run, halo exchanges are needed to carry out the computation over the mesh elements in the boundary of each MPI partition. Additionally, data races over MPI are handled by redundant computation over the halo elements [25]. This approach with the runtime information available through the API enables the DSL to minimize data movement between MPI nodes, perform checkpointing automatically, handle data movement between different memory environments in the case of GPUs, and provide similar application-level features.

1.3.2 OPS for Structured Mesh Solvers

The Oxford Parallel Library for Structured mesh solvers (OPS) [11] is an eDSL designed to bridge the gap between high-level scientific stencil computations and the optimal use of parallel and distributed systems. OPS is integrated into C, C++, and Fortran, similarly to OP2, and offers computational scientists and engineers a set of abstractions tailored for structured mesh calculations. By separating the high-level application code from the low-level parallel implementation, OPS alleviates the burden on scientists and engineers to delve into the intricate aspects of parallelism and data movement. This approach enables them to craft a singular high-level source code, while the library handles the nuances of optimizing for the designated architectures.

Stencil computations apply a specific operation to each element of a Cartesian grid accessing a fixed pattern of neighboring elements, called a stencil. Essentially, a stencil defines the method to combine the values of adjacent cells in a grid to determine a new value for a particular cell. These computations are prevalent in numerous domains, such as CFD [29], [30], climate modeling [31], [32] and computational finance [33], [34]. They are efficient and scalable, making them suitable for large-scale simulations on parallel computing architectures. The process of parallelizing

Listing 1.2 Example declaration of an OPS dataset and its use in an OPS parallel loop.

```
// User kernel
void stencil(const ACC<double> &u, const ACC<double> &f,
             ACC<double> &u2) {
  u2(0, 0) = ((u(-1, 0) + u(1, 0)) * dy * dy +
              (u(0, -1) + u(0, 1)) * dx * dx -
              f(0, 0) * dx * dx * dy * dy) /
             (2.0 * (dx * dx + dy * dy));
}
// ...
// Declaring a dataset on a block
int size[2] = {size_x, size_y}
int base[2] = {0, 0};
// max halo depths for the dat in all direction
int d_p[2] = {1, 1};
int d_m[2] = {-1, -1};
ops_dat u = ops_decl_dat(block, 1, size, base, d_m, d_p,
                         nullptr, "double", "u");
// Execute a given loop on the block
int iter_range[] = {0, size_x, 0, size_y};
ops_par_loop(stencil, "stencil", block, 2, iter_range,
         ops_arg_dat(u, 1, S2D_4PT, "double", OPS_READ),
         ops_arg_dat(f, 1, S2D_00, "double", OPS_READ),
         ops_arg_dat(u2, 1, S2D_00, "double", OPS_WRITE));
```

stencil codes is relatively straightforward. The techniques to optimize and execute stencil loops effectively on modern parallel platforms have been extensively researched [35]–[37].

OPS defines several fundamental abstractions for expressing computations. *Blocks* represent multi-dimensional containers on which datasets are defined. Similarly to OP2, *datasets* are collections of data entities associated with blocks (instead of sets) and are used in the *computational kernels* through the user-defined *stencils* describing the access pattern of the datasets. Computations within OPS are generally formulated using the *parallel loop* calls, where custom, user-defined kernels are applied across designated iteration spans and datasets on the same block. In the phase of automatic transformation, OPS substitutes these parallel loop calls with calls to parallel implementations specific to the intended target. The loops require the users to define the mode of access for each dataset by the kernel – such as data being read, written, or incremented – along with the distinct stencil design that directs data access. OPS requires the parallel operation to be insensitive to the order of execution on individual grid points (within machine precision). Through this abstraction, domain experts can express algorithms for structured meshes without detailing the execution method in a parallel environment with heterogeneous memory spaces.

Listing 1.2 illustrates the API for forming a dataset on a block and subsequently utilizing the datasets in an OPS parallel loop, taken from the Poisson example application. Scientists convey the computation through these computational loops, supplying the loop body to be enacted for each grid point, the block, the iteration span, and a descriptor for every loop body argument. These descriptors contain details about the dataset, the access mode, the stencil, and the inherent data type. The source-to-source translation layer of OPS processes this information and generates the hardware-specific low-level parallel implementations for the kernel. OPS supports and generates code for a range of target hardware with different parallel programming models such as OpenMP, OpenACC, CUDA, and their combination with MPI. There's no need for users to alter the OPS application source to take advantage of these frameworks. The transformed source code is then compiled using traditional compilers and linked against OPS backend libraries.

It is important to highlight that OPS takes ownership of all datasets and can exclusively be accessed through APIs. This allows OPS to monitor and follow the state of each dataset, determining when data transfers are needed, for example, between CPUs and GPUs. The ability to track data movement and dependencies across the application has proven crucial in enabling OPS to implement sophisticated optimizations, including loop tiling and lazy execution[11].

OPS enhances the productivity of researchers by enabling them to develop a single high-level application source, which then automatically benefits from all the optimizations OPS offers for both existing and upcoming architectures. While adopting the OPS abstraction does involve certain costs, like refactoring loops into outlined loop bodies and `ops_par_loop` calls, the payoff is substantial. By maintaining a singular codebase, the application can effortlessly gain access to highly optimized implementations for an extensive variety of hardware platforms.

1.4 Source-to-source translation in HPC

Source-to-source transformations and code generation is used in many HPC frameworks, particularly as a means to help application developers write programs for new hardware. Some of the earliest were motivated by the emergence of NVIDIA GPUs for scientific computations. Williams et al. [38], [39] showed that with proper annotations on the source written in the MINT programming model [40], the ROSE compiler tool-chain [41] can be used to generate transformations to utilize GPUs. ROSE was also previously explored as a source-to-source translator toolchain in the initial stages of the OP2 project [42]. However, it proved hard to maintain and required substantial coding effort to change the generated code or adopt new parallelization models. Ueng et al. with CUDAlite [43] showed that the memory usage of existing annotated CUDA codes can be optimized with source-to-source tools based on the Phoenix compiler infrastructure. Other notable works include translators such as O_2G [44] based on the Cetus compiler framework [45], which is designed to perform source-to-source transformations based on static data dependence analysis and the hiCUDA [46] programming model which use a set of directives to transform C/C++ applications to CUDA using the front-end of the GNU C/C++ compiler. Another notable source-to-source transformation tool is Scout [47], which uses Low Level Virtual Machine (LLVM)/Clang to vectorize loops using SIMD instructions at the source level. The source-to-source transformation entails replacing expressions with their vectorized intrinsic counterparts. Other parallelizations are not supported. To achieve this, Scout modifies the AST of the source code directly. The tools' capabilities have been applied to the production-level Computational Fluid Dynamic (CFD) codes at the German Aerospace Centre.

There are two main issues with the above works: (1) difficulties in extending them to generate new parallelizations or generating multiple target parallel code and (2) the underlying source-to-source translation tool relying on unmaintained software technologies due to their lack of adaptation by the community. Both these issues make them difficult to use in eDSLs such as OP2, which has so far relied on tools written in Python to carry out the translation of higher-level API statements to their optimized platform-specific versions. In fact, Python has been and continues to be used in related DSLs and active library frameworks for target code generation. These include, OPS [48], Firedrake [21], OpenSBLI [49], DeVito [16] and others. However, the tools written in Python significantly lack the robustness of compiler frameworks such as Clang/LLVM or GNU. These tools are capable of limited syntax or semantic checking only, have even limited error/bug reporting to ease the development, and become complicated very quickly when adding different optimizations, which in turn affect its maintainability and extensibility.

Previous works with LibTooling include [50], [51], which demonstrate its use for translation of annotated C/C++ code to CUDA using the MINT programming model. While the goals of my research in 2 are similar, it focuses on the use of LibTooling not only for generating CUDA code but to support other parallelizations such as OpenMP, SIMD, and MPI, as well.

1.5 Alternating Direction Implicit (ADI) Method

The OPS and OP2 DSLs allow for the description of a wide range of structured- and unstructured-mesh algorithms, making them applicable to a relatively broad range of problems. However, there are a handful of frequently used algorithms for the solution of linear systems of equations - implicit solvers - that are highly computationally expensive, and given a relatively narrow set of implementation parameters, they lend themselves to specialized implementations. They also commonly use algorithms that do not fit the abstraction of OP2/OPS; for example by violating the order-independence restriction. A prime example is the ScaLAPACK [52], [53] - the Scalable Linear Algebra PACKage - numerical applications often spend a high fraction of their execution time in implicit solves, therefore any performance improvements have a relatively large impact on overall application performance; there is a long history of developing tuned implementations for various hardware architectures [54]–[58].

The Alternating Direction Implicit (ADI) method is a numerical technique used primarily to solve partial differential equations (PDEs), especially those that are parabolic or elliptic. Originally ADI was developed to solve the 2D diffusion equation on a Cartesian grid using finite differences[59]. It is particularly useful for multidimensional problems, such as those involving two or three spatial variables. The method is a form of operator splitting that allows for implicit time-stepping while reducing the complexity of solving the resulting systems of equations. Implicit methods solve PDEs by solving a system of equations at each time step. The solution of these equations can be computationally expensive. Operator splitting methods (such as ADI) the multidimensional PDE into a series of lower-dimensional problems that are easier to solve and approximating the solution of the original equations. In the case of ADI, the resulting lower-dimensional problems are one-dimensional tridiagonal or pentadiagonal systems. ADI applies these implicit steps one after the other, alternating the direction in which they are applied. While the library support for these linear system solvers on different hardware is growing, batched tridiagonal and pentadiagonal system solvers are often missing or lack performance in these libraries.

As an example let's look at how would ADI approach a 2D heat equation on a uniform grid.

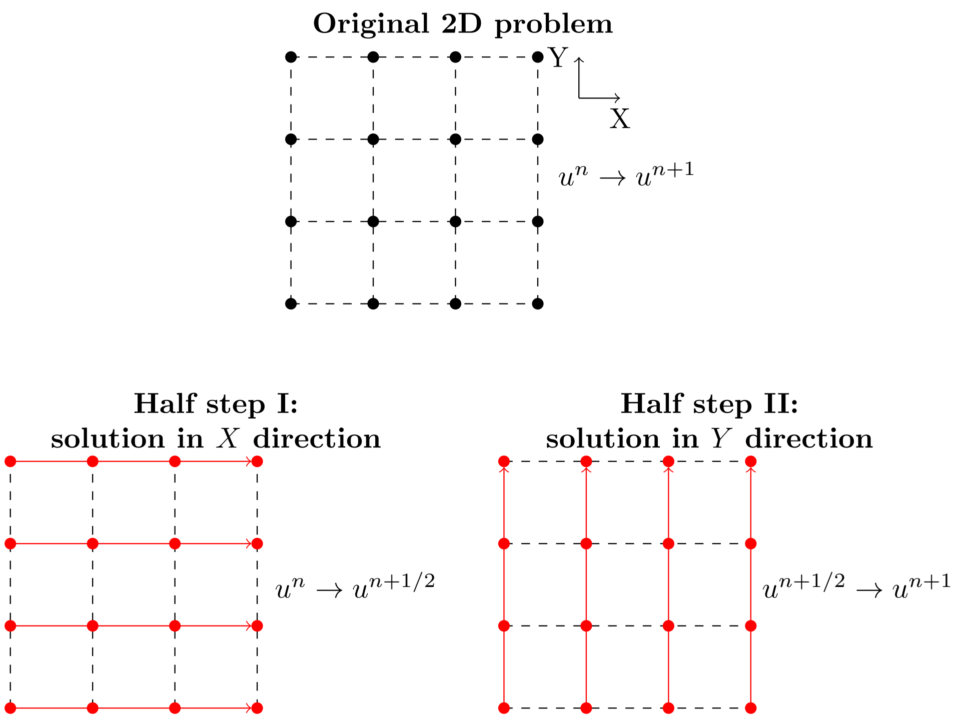


Figure 1.3: ADI half steps on a 2D uniform grid. In the half steps, the red arrows mark the required independent 1D systems that require solutions to advance the state.

The 2D heat equation:

$$\frac{\partial u}{\partial t} = \frac{\partial^2 u}{\partial x^2} + \frac{\partial^2 u}{\partial y^2}$$

After discretisation we have $u^n_{i,j}$ representing the temperature at grid point (x_i, y_j) at time step n. Using finite differences the second derivatives are approximated as:

$$\frac{\partial^2 u}{\partial x^2} = \frac{u_{i+1,j} - 2u_{i,j} + u_{i-1,j}}{\Delta x^2}$$

$$\frac{\partial^2 u}{\partial y^2} = \frac{u_{i,j+1} - 2u_{i,j} + u_{i,j-1}}{\Delta y^2}$$

The ADI method will define two half steps to compute u^{n+1} at timestep n. The first half step will compute $u^{n+1/2}$ as:

$$\frac{u^{n+1/2}_{i,j} - u^n_{i,j}}{\Delta t/2} = \frac{u^{n+1/2}_{i+1,j} - 2u^{n+1/2}_{i,j} + u^{n+1/2}_{i-1,j}}{\Delta x^2} + \frac{u^n_{i,j+1} - 2u^n_{i,j} + u^n_{i,j-1}}{\Delta y^2}$$

Note that the unknowns in the equation rearranging for $u^{n+1/2}_{i,j}$ form tridiagonal equation systems along the X dimension. After solving these equations the second half step will compute u^n from the intermediate solution using similar tridiagonal systems along the Y dimension:

$$\frac{u^{n+1}_{i,j} - u^{n+1/2}_{i,j}}{\Delta t/2} = \frac{u^{n+1/2}_{i+1,j} - 2u^{n+1/2}_{i,j} + u^{n+1/2}_{i-1,j}}{\Delta x^2} + \frac{u^{n+1}_{i,j+1} - 2u^{n+1}_{i,j} + u^{n+1}_{i,j-1}}{\Delta y^2}$$

The ADI method will apply these alternating half steps at each timestep. Figure 1.3 shows how each half step requires the solution of a batch of tridiagonal equation systems forming along one dimension of the grid. Together, the solution of these systems will provide the state of the next time iteration. To compute a single half-step ADI methods will require the solution of independent tridiagonal equation systems along the other directions, in other words computing the half-step in X direction requires solving N_Y independent systems, similarly the half-step in Y will require N_X solves. With increasing the number of spatial dimensions the number of independent systems (the batch size) increases by a factor of the size of the new dimension while the number of implicit steps (batch-tridiagonal problems) required to compute a timestep also increases. The key advantage of the ADI method is the lower computational complexity which becomes more and more significant as the number of spatial dimensions increases. At the same time, the increasing number of batch problems and the increasing size of these problems create new challenges and make the efficient solution of large batch tridiagonal systems along different directions of the computational grids a crucial point for performance in ADI applications.

1.6 Tridiagonal Systems Solver Algorithms

Tridiagonal systems solvers arise from the need to solve a system of linear equations as given in (Equation (1.1)) or its matrix form of $Ax = d$ given in (Equation (1.2)), where $a_0 = c_{N-1} = 0$.

$$\begin{aligned} &a_i u_{i-1} + b_i u_i + c_i u_{i+1} = d, \\ &i = 0, 1, \ldots, N-1 \end{aligned} \tag{1.1}$$

$$\begin{bmatrix} b_0 & c_0 & 0 & \dots & 0 \\ a_1 & b_1 & c_1 & \dots & 0 \\ 0 & a_2 & b_2 & \dots & 0 \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \dots & a_{N-1} & b_{N-1} \end{bmatrix} \begin{bmatrix} u_0 \\ u_1 \\ u_2 \\ \vdots \\ u_{N-1} \end{bmatrix} = \begin{bmatrix} d_0 \\ d_1 \\ d_2 \\ \vdots \\ d_{N-1} \end{bmatrix} \tag{1.2}$$

The solution to such systems is well known. Thomas [60] presented a sequential algorithm - the specialization of the well-known Gaussian elimination for tridiagonal systems - while Cyclic Reduction (CR) [61] and PCR [62] are inherently parallel. The latter has been used extensively to implement solvers on GPUs [63]–[65]. Additionally, combinations of Thomas and PCR have been used in a hybrid algorithm, demonstrating better performance in several cases [65], [66]. In most applications, the tridiagonal systems are scalar, with only one unknown per grid point. However, multiple unknowns leading to block-tridiagonal structures do occur in areas such as CFD. This chapter focuses on scalar tridiagonal systems, noting that the same algorithms extend naturally to block-tridiagonal systems.

The *Thomas Algorithm* (Algorithm 1), just like Gaussian elimination, consists of a forward pass to eliminate the lower diagonal elements, a_i of the tridiagonal matrix, by adding a multiple of the row above. A backward pass follows using the modified c_i values from the last index to the first. This algorithm is inherently serial, as each iteration of the loops has a dependency on the previous iteration, taking $2N$ steps.

The CR algorithm uses two phases to solve the tridiagonal systems. The first phase. the

Algorithm 1 `thomas`(a, b, c, d)

```
1: $d_0^* \leftarrow d_0/b_0$
2: $c_0^* \leftarrow c_0/b_0$
3: for $i = 1, 2, ..., N-1$ do
4:     $r \leftarrow 1/(b_i - a_i c_{i-1}^*)$
5:     $d_i^* \leftarrow r(d_i - a_i d_{i-1}^*)$
6:     $c_i^* \leftarrow r c_i$
7: end for
8: for $i = N-2, ..., 1, 0$ do
9:     $d_i \leftarrow d_i^* - c_i^* d_{i+1}$
10: end for
11: return $d$
```

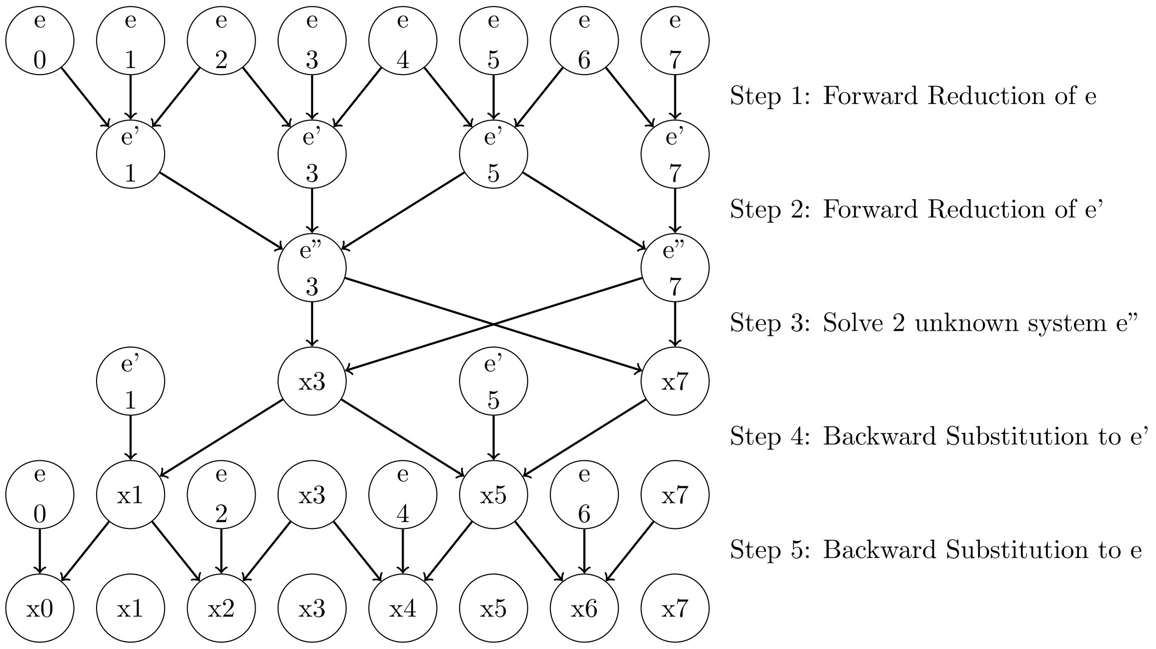

e 0
e 1
e 2
e 3
e 4
e 5
e 6
e 7
Step 1: Forward Reduction of e
e' 1
e' 3
e' 5
e' 7
Step 2: Forward Reduction of e'
e" 3
e" 7
Step 3: Solve 2 unknown system e"
x3
x7
Step 4: Backward Substitution to e'
x1
x5
Step 5: Backward Substitution to e
x0
x2
x4
x6

Figure 1.4: Steps of the CR algorithm for a system of size 8.

forward reductions, halve the number of unknowns in every step by eliminating every second unknown from the equation system. Figure 1.4 shows the steps of the CR algorithm on a system with $N = 8$ unknowns. When there are only two unknowns remain the CR algorithm computes these variables and start the backward substitution phase by back substituting the equations from the forward reductions in reverse order.

In contrast, the *PCR algorithm* (Algorithm 2), assumes the matrix is normalized so that $b_i = 1$ and then, similarly to CR, for each matrix row i, subtracts multiples of rows $i \pm 2^0, 2^1, 2^2, ..., 2^{P-1}$, where P is the smallest integer such that $2^P \geq N$. Each iteration of the PCR algorithm reduces each of the current systems $(a^{(p-1)}, c^{(p-1)}, d^{(p-1)})$ into two systems of half the size $(a^{(p)}, c^{(p)}, d^{(p)})$ as Figure 1.5 shows. After P steps, all of the modified $a^{(P)}$ and $c^{(P)}$ coefficients become zero, leaving values for the unknowns u_i in $d_i^{(P)}$. The key difference to CR is the fact that the PCR algorithm trades the backward substitution phase from CR to redundant computations which are computed at the same time.

In PCR, the iterations of the inner loop do not depend on each other, allowing multiple

Algorithm 2 `pcr`(a, c, d)

```
1:  for $p = 1, 2, ..., P$ do
2:      $s \leftarrow 2^{p-1}$
3:      for $i = 0, 1, ..., N-1$ do
4:          $r \leftarrow 1/(1 - a_i^{(p-1)} c_{i-s}^{(p-1)} - c_i^{(p-1)} a_{i+s}^{(p-1)})$
5:          $a_i^{(p)} \leftarrow -r(a_i^{(p-1)} a_{i-s}^{(p-1)})$
6:          $c_i^{(p)} \leftarrow -r(c_i^{(p-1)} c_{i+s}^{(p-1)})$
7:          $d_i^{(p)} \leftarrow r(d_i^{(p-1)} - a_i^{(p-1)} d_{i-s}^{(p-1)} - c_i^{(p-1)} d_{i+s}^{(p-1)})$
8:      end for
9:  end for
10: return $d^{(P)}$
```

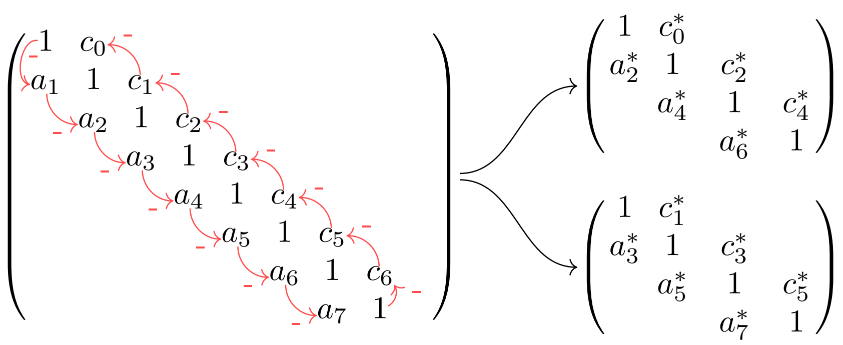


Figure 1.5: Result of a single iteration of the PCR algorithm. After the iteration, every second row will form a separate tridiagonal system.

threads to be used to solve each tridiagonal system. PCR is more computationally expensive than the Thomas algorithm. Nevertheless, it is well suited for implementations on modern multi-core/many-core architectures with high computational capabilities. The *CR algorithm* is similar to PCR but consists of a forward and backward pass. The forward pass of CR is the same as the PCR algorithm but with an additional reverse pass that performs a back solve. This results in fewer operations overall but exhibits less parallelism and requires twice as many passes.

$$\left(\begin{array}{cccccccccccc}
\mathbf{1} & & & \boldsymbol{c_0^*} & & & & & & & & \\
a_1^* & 1 & & c_1^* & & & & & & & & \\
a_2^* & & 1 & c_2^* & & & & & & & & \\
\boldsymbol{a_3^*} & & & \mathbf{1} & \boldsymbol{c_3^*} & & & & & & & \\
\hline
 & & & \boldsymbol{a_4^*} & \mathbf{1} & & & \boldsymbol{c_4^*} & & & & \\
 & & & & a_5^* & 1 & & c_5^* & & & & \\
 & & & & a_6^* & & 1 & c_6^* & & & & \\
 & & & & \boldsymbol{a_7^*} & & & \mathbf{1} & \boldsymbol{c_7^*} & & & \\
\hline
 & & & & & & & \boldsymbol{a_8^*} & \mathbf{1} & & & \boldsymbol{c_8^*} \\
 & & & & & & & & a_9^* & 1 & & c_9^* \\
 & & & & & & & & a_{10}^* & & 1 & c_{10}^* \\
 & & & & & & & & \boldsymbol{a_{11}^*} & & & \mathbf{1}
\end{array}\right)
\left(\begin{array}{c}
\boldsymbol{u_0} \\ u_1 \\ u_2 \\ \boldsymbol{u_3} \\ \hline \boldsymbol{u_4} \\ u_5 \\ u_6 \\ \boldsymbol{u_7} \\ \hline \boldsymbol{u_8} \\ u_9 \\ u_{10} \\ \boldsymbol{u_{11}}
\end{array}\right)
=
\left(\begin{array}{c}
\boldsymbol{d_0^*} \\ d_1^* \\ d_2^* \\ \boldsymbol{d_3^*} \\ \hline \boldsymbol{d_4^*} \\ d_5^* \\ d_6^* \\ \boldsymbol{d_7^*} \\ \hline \boldsymbol{d_8^*} \\ d_9^* \\ d_{10}^* \\ \boldsymbol{d_{11}^*}
\end{array}\right)$$

Figure 1.6: Tridiagonal matrix split into 3 subsystems after the hybrid Thomas-PCR forward pass [65]. The reduced system is shown in bold and $M = 4$.

Algorithm 3 `hybrid_forward`(a, b, c, d)

```
1: for $i = 0, 1$ do
2:     $d_i^* \leftarrow d_i / b_i$
3:     $a_i^* \leftarrow a_i / b_i$
4:     $c_i^* \leftarrow c_i / b_i$
5: end for
6: for $i = 2, 3, ..., M-1$ do
7:     $r \leftarrow 1/(b_i - a_i c_{i-1}^*)$
8:     $d_i^* \leftarrow r(d_i - aid_{i-1}^*)$
9:     $a_i^* \leftarrow -r a_i a_{i-1}^*$
10:    $c_i^* \leftarrow r c_i$
11: end for
12: for $i = M-3, M-4, ..., 1$ do
13:    $d_i^* \leftarrow d_i^* - c_i^* d_{i+1}^*$
14:    $a_i^* \leftarrow a_i^* - c_i^* a_{i+1}^*$
15:    $c_i^* \leftarrow -c_i^* c_{i+1}^*$
16: end for
17: $r \leftarrow 1/(1 - c_0^* a_1^*)$
18: $d_0^* \leftarrow r(d_0^* - c_0^* d_1^*)$
19: $a_0^* \leftarrow r a_0^*$
20: $c_0^* \leftarrow -r c_0^* c_1^*$
21: return $a^*, c^*, d^*$
```

1.6.1 Hybrid Algorithms

Combining Thomas with PCR has been shown to result in the best performance on GPUs [65]. The tridiagonal system is split into subsystems of size M, each of which is handled by a separate thread. Each subsystem is solved using a modified Thomas algorithm where in a forward pass (see Algorithm 3) each unknown is expressed in terms of two unknowns, u_0 and u_{M-1}:

$$a_i^* u_0 + u_i + c_i^* u_{M-1} = d_i^*, \quad i = 1, 2, ..., M-2.$$

The forward pass results in a reduced tridiagonal system made up of the unknowns at the beginning and end of each subsystem, as can be seen in Figure 1.6. How this reduced system is solved has a significant impact on the overall performance of the solver. Options for solving this include the previously mentioned algorithms. Finally, the result of the reduced system is substituted back into the individual subsystems, running a backward pass to solve each subsystem (see Algorithm 4).

The Thomas-PCR implementation on NVIDIA GPUs developed by László et al. [65] has a key advantage where up to a certain size of the system, the entire subsystem can be stored in the registers of a warp (32 CUDA threads). It can outperform the other previously described algorithms until the subsystem becomes too large to fit in a warp's registers. Once this limit is reached, a Thomas algorithm optimized for GPU memory accesses was shown to outperform the Thomas-PCR hybrid.

Other hybrid algorithms have been proposed, such as a CR-PCR hybrid in [63], [66]. This solver again targets GPUs and starts by using the CR algorithm due to its low algorithmic complexity. Similarly to the Thomas-PCR, the hybrid CR-PCR uses the forward reduction of the

Algorithm 4 `hybrid_backward`(a^*, c^*, d^*)

```
1: $d_0 \leftarrow d_0^*$
2: for $i = 1, 2, ..., M-1$ do
3:     $d_i \leftarrow d_i^* - a_i^* d_0^* - c_i^* d_{M-1}^*$
4: end for
5: $d_{M-1} \leftarrow d_{M-1}^*$
6: return $d$
```

first algorithm (CR in this case) to create smaller reduced systems of size M. It then switches to PCR when there is no longer enough parallelism to fully exploit the GPU. Solve the remaining reduced system using the PCR algorithm. Finally, the algorithm substitutes the solved unknowns back into the original systems using the backward substitution phase of CR.

A further alternative is the PCR-Thomas algorithm [66], which uses PCR to decompose the tridiagonal system into multiple smaller tridiagonal systems and then solve these smaller systems using the standard Thomas algorithm. The key difference between the Thomas-PCR [65] and PCR-Thomas [66] is in the memory access patterns used during the solver algorithm. In the case of Thomas-PCR the Thomas algorithm operates on consecutive indices to form the reduced system (which is solved by PCR), which provides more optimizations on the memory accesses, while starting with PCR will inherently start the algorithm with strided indices.

Regardless of the algorithm used, solving tridiagonal systems on parallel systems is considered to be memory bandwidth bound, especially on GPUs [65]. Designing the memory access patterns of the algorithms to give coalesced and aligned memory accesses is a key issue.

This is particularly true for the X dimension solve where the natural memory access pattern prohibits coalesced memory operations. Various solutions to this issue have been proposed, including global transposes [67] and local transposes [65].

1.6.2 Iterative Solutions

We have considered only algorithms providing exact solutions to tridiagonal systems. Iterative methods with approximate solutions, such as the widely used Jacobi method, are also available. Such approaches are applicable to solving general systems resulting in diagonally dominant matrices, not only tridiagonal systems. The solution to $Ax = d$ is sought starting from an initial guess for the unknowns, iterating until a given convergence criterion is met. Algorithm 5 details the Jacobi method applied to a tridiagonal system, where a, b and c are arrays holding the three diagonals and d holds the right hand side of the equation.

Each iteration requires 5 operations per grid point for tridiagonal matrices, with an additional cost of checking the convergence of the solution within a desired tolerance (approximate solution). So, while an iteration of the Jacobi method is cheaper than a direct approach, multiple iterations are needed, with many iterations required for poorly conditioned systems.

1.7 Current Library Support to Solve Tridiagonal Systems

Several algorithms have been proposed [68]–[71] for which the tridiagonal matrices can be split over multiple processes. Partitioning-based algorithms like PCR-pThomas[72] proved to be

Algorithm 5 `jacobi`(a, b, c, d)

```
1: $p \leftarrow 1$
2: while Not Converged do
3:     $x_0^{(p)} \leftarrow \left(d_0 - c_0 x_1^{(p-1)}\right)/b_0$
4:     for $i = 1, ..., N-2$ do
5:         $x_i^{(p)} \leftarrow \left(d_i - a_i x_{i-1}^{(p-1)} - c_i x_{i+1}^{(p-1)}\right)/b_i$
6:     end for
7:     $x_{N-1}^{(p)} \leftarrow \left(d_{N-1} - a_{N-1} x_{N-2}^{(p-1)}\right)/b_{N-1}$
8:     checkIfConverged()
9:     $p \leftarrow p + 1$
10: end while
11: return $x^{(p)}$
```

efficient on GPUs. For a small number of GPUs solving a single system, Chang et al.[73] introduced a numerically stable solver based on the SPIKE algorithm [74] with diagonal pivoting. Efficient multi-GPU batch tridiagonal solvers were introduced in Diéguez et al. [75], but the implementation assumes that the GPUs own separate batches with full systems - which is not the case in ADI use cases

TridiagLU [76] provides a state-of-the-art implementation for batches of tridiagonal systems for CPU clusters. Many of these algorithms divide the system into partitions and form a smaller decoupled tridiagonal system connecting the partitions (reduced system). In TridiagLU, the partitioning is the MPI decomposition, and each MPI process holds a single row from the reduced system.

The reduced system is solved iteratively using the Jacobi method, computing an approximate solution only. Instead of checking for convergence, an optional estimate of iterations for convergence can be provided to TridiagLU (to avoid the residual calculation with extra global communications). However, this limits its use to domains where it is possible to provide a good estimate of the iterations required. For cases where estimating an iteration count for convergence is not practical, TridiagLU can calculate a global norm to check for convergence at the expense of some performance.

TridiagLU also has the option to gather a reduced system, corresponding to one tridiagonal system, onto a single MPI process and solve it on that MPI process. The result is then scattered back to the relevant MPI processes after the reduced solution. Different reduced systems in the batch of tridiagonal systems will be gathered to different MPI processes so that the load is balanced. Naturally, the use of global collectives degrades performance when using these options. Therefore, these implementations do not scale beyond a certain number of MPI nodes.

The PaScal_TDMA library[77], [78] provides support for distributed solution of batched tridiagonal systems for CPU clusters similarly to TridiagLU. The main difference that while PaScal_TDMA supports solver calls along different dimensions, it only supports all-to-all communication based solver algorithms for the reduced systems. Leading to the same issues on large number of processes.

Although these libraries show relevant work, they lack support for common memory and MPI layouts that are present in an ADI application, resulting in the use of expensive transpose operations. My research aims to provide a performance portable solver library supporting both

GPU and CPU clusters computing exact solutions of the batch-tridiagonal systems for ADI applications.

1.8 Sensitivities in structured-mesh applications

Sensitivity (mathematical derivative) information has a wide range of uses, such as gradient-based design optimization for Computational Fluid Dynamic (CFD) applications [79]–[83], machine learning [84], inverse problems [85], computational finance [86], [87], image processing [88], medical imaging[89] or real-time risk management [90]. Algorithmic Differentiation (AD) is an essential tool for obtaining sensitivity information in such applications: finite differences are often too expensive (or inaccurate) while deriving and then evaluating symbolic gradient information is often very laborious and can lead to brittle, error-prone code.

Algorithmic Differentiation, also known as Automatic Differentiation or AD, is a set of techniques that can compute the exact derivatives of a function with respect to its input variables by repeatedly applying the chain rule of calculus. Depending on the direction of the evaluation of the chain rule, we distinguish between two different modes of AD: forward (tangent) mode AD, where the evaluation order follows the order of the original computation, and reverse (adjoint) mode AD, which evaluates the chain rule in reverse order. AD is particularly useful in optimization problems where objective functions are not available in closed form and require many hundreds or thousands of lines of computer code to evaluate. The function's derivatives are crucial in determining the direction and rate of change that will lead to the optimal solution. One evaluation with forward mode AD will compute the derivatives of all outputs with respect to a single input, while one evaluation with adjoint mode AD will produce the derivatives of one output with respect to all inputs. A significant advantage of AD over finite difference approaches is that it can compute derivatives of arbitrary order to machine precision, and in cases where the application produces far fewer outputs than inputs, adjoint mode AD can compute the gradient hundreds of times faster[91].

Adjoint-mode Algorithmic Differentiation (AAD) has been applied successfully for multiple application areas in the past years. In CFD, AAD is used for optimizing the design of business jets[79]. Guasch et al.[85] used AAD to compute accurate three-dimensional images of the brain with high resolution from ultrasound recordings. In computational finance, AAD is the industry standard to compute Greeks in option pricing problems[86], [87], and in medical imaging, AAD can be used in registration problems to match the alignment of images from multiple sources like X-ray and CT data[89].

Implementing derivatives of applications is difficult, error-prone, and particularly challenging in a parallel environment [92]. Ideally, users would like to be able to describe what needs to be computed in simple terms, including any derivative information, and then have a tool or framework to generate an efficient parallel implementation, including choosing the most appropriate method for obtaining the derivatives.

1.9 Algorithmic Differentiation

Algorithmic Differentiation (AD) is a set of techniques for computing derivatives of functions represented as computer programs. Assume we have an application with some input variables $x \in \mathbb{R}^n$, which perform a series of steps and then compute the final result $y \in \mathbb{R}^m$. If we take this program as a mathematical function, $F : \mathbb{R}^n \to \mathbb{R}^m$, AD will compute the value

$$\bar{x} = \frac{\partial F(x)}{\partial x} = \frac{\partial y}{\partial x}.$$

Working out and manually writing code to compute analytical derivatives directly could provide the fastest solution, but it is error-prone and simply becomes infeasible for large applications. Most commonly, there are three ways to compute the derivative information automatically: symbolic differentiation in computer algebra systems, numeric differentiation with finite differences, and algorithmic differentiation [92]. Symbolic differentiation computes derivatives by manipulating symbolic expressions. It could provide exact derivatives, but it can be slow and requires the problem to be defined as closed-form expressions [84], which in many cases is just not possible. Using finite differences to estimate derivatives is sometimes the easiest to implement, but the derivatives are subject to floating point precision, and getting accurate second or third-order information can be a real challenge. Finally, AD evaluates derivatives by differentiating the sequence of elementary arithmetic operations and elementary functions which make up a user's application, thus ensuring accuracy.

Algorithmic differentiation builds on the fact that a computer program, no matter how complex, will always be constructed from a series of elementary operations such as additions, multiplications, or math functions like sine. The derivatives of these elementary operations are known and easy to compute. If we consider the function to be a composite with K elementary steps $F = f_1 \circ f_2 \circ \cdots \circ f_K$ then

$$y = F(x) = f_K(\ldots f_2(f_1(x)))$$

and the derivative of y can be computed by applying the chain rule to all of the elementary functions

$$\frac{\partial y}{\partial x} = \frac{\partial f_1}{\partial x}\frac{\partial f_2}{\partial f_1}\cdots\frac{\partial f_K}{\partial f_{K-1}} \tag{1.3}$$

There are two main modes of algorithmic differentiation based on which direction we start to evaluate Equation (1.3). Let J note the $m \times n$ Jacobian matrix of F with $(i,j)^{th}$ component $J_{ij} = \frac{\partial f_i}{\partial x_j}$ and $u \in \mathbb{R}^n$ a vector in the input space. The tangent or forward-mode AD will compute the action of the Jacobian matrix on u:

$$Ju = J_K J_{K-1} \cdots J_1 u$$

The forward-mode AD evaluates the derivatives in the same order as the original function evaluation (if F is implemented in a computer program, this is the same order in which the program evaluates statements). Choosing u to be a vector such that it is 1 for input j and 0 otherwise, a single evaluation of the tangent model will compute the j^{th} column of the Jacobian, in other words, the derivatives of all outputs with respect to a single input.

For adjoint or reverse-mode AD, let us consider $w \in \mathbb{R}^m$, a vector in the output space. Adjoint-mode AD will compute the action of the transpose of J on w:

$$J^T w = J_1^T J_2^T \cdots J_K^T w$$

Similarly to the previous case, let us choose w such that it has a single 1 at position i and all other values are 0. A single evaluation of the adjoint model on w will produce the i_{th} row of J, or the derivatives of a single output with respect to all inputs. Evaluation of the tangent model requires twice as much memory and typically takes roughly twice as long as the original application. However, often derivatives with respect to multiple inputs are required.

The adjoint-mode AD evaluates derivatives in the reverse order, equivalent to running your computer program backwards. Doing this requires storing a Directed Acyclic Graph (DAG) of the program control flow in memory, along with local derivative information. This can cause dramatic increases in memory requirements. The data structure that records the intermediate states and the computational graph is often referred to as the tape. One of the main challenges is to efficiently save all intermediate state and the computational steps (and their local Jacobians) while computing the result of the program, so that one can propagate derivative information from outputs back to inputs.

The choice between forward and reverse mode AD often depends on the target application. The forward mode can perform better for applications with a few inputs and a lot of outputs, and reverse mode AD shines in scenarios where a function has a large number of inputs but a relatively smaller number of outputs, like gradient-based optimization and machine learning. Due to its efficiency in such contexts, adjoint mode AD often receives special attention. This work focuses on adjoint mode AD, where OPS can take advantage of a high-level decomposition of F using computational loops as the f_i composite functions instead of elementary operations. Most adjoint mode AD tools belong to one of two categories: either they apply source transformation to generate the adjoints of the function or use operator overloading techniques [91]. Source transformation tools automatically transform the original code to produce a new program that calculates the adjoint values, while operator overloading tools leverage the object-oriented capabilities of C++ and other object-oriented languages to overload basic arithmetic operations and functions to compute derivatives.

While tracing the original data flow with operator overloading is easier, the resulting tape quickly increases in size. Source transformation tools can reduce the performance overhead of adjoint calculations and potentially allow more drastic optimizations. In summary, algorithmic differentiation provides an effective and accurate means to compute derivatives, with adjoint mode AD being especially pertinent in contexts with numerous input variables and only a small number of outputs. Its strategic application ensures the derivation of gradients with both computational efficiency and high precision. Our approach uses source transformation techniques that allow it to store the control flow on a higher level (at the level of entire parallel loops) while also having the ability to apply transformations and optimizations on the loops to achieve near-peak performance on parallel hardware.

I refer to the evaluation of the function F (with the additional caching and tape recording) as the forward pass. The computations that are part of F are called *primal*, and their counterparts computing the local Jacobian are called the *adjoint of the computation.* The evaluation of $J^T w$ executing the adjoints of the computational steps in reverse order we call the *reverse pass.* We refer to variables and computations that take part in the derivative accumulation as *active* and all other data and code as *passive.* Finally, the variables holding the values and states during the forward pass are called *primal data*, and for active data, we will refer to the variables holding derivative values as *adjoint data.*

1.10 Algorithmic Differentiation in HPC

The ability to produce exact derivatives efficiently for many inputs made Adjoint mode AD (AAD) tools popular. Despite the challenges of efficient implementation, AAD is used in many different areas and applications using both source transformation based techniques and operator overloading. Application areas include for example shape optimization tasks [93]–[95], or in CFD solvers like the NASA ice Sheet System Model[96], OpenFOAM[97], [98] or STAMPS[99].

The most straightforward way to implement AAD is using operator overloading, which allows the flexibility to easily follow complex control flow and different language constructs. Multiple tools exist that efficiently use operator overloading, like Adept[100], dco/c++[101] or Sacado[102]. However, naively implemented operator overloading tools lead to large memory overheads for storing the tape, which is often alleviated with advanced techniques like expression templates[103], using direct derivatives of linear algebra constructs like in the Stan Math library[104] or additional hybrid code generation extensions in case of dco/c++.

The other approach uses source-to-source transformation techniques to generate code to compute the derivatives. This approach can reduce the overhead of tracing the computations at the cost of more complex tooling. Examples of AAD tools with source transformation include ADIC[105], Tapenade[106], OpenAD[107].

To utilize modern hardware efficiently, simulations and applications generally take advantage of parallelism. Computing derivatives and reversing the data flow in a parallel environment brings additional challenges for AD tools. The parallelization of derivative computations is achieved in many ways in AD tools. A fairly simple approach is to compute different tangent mode AD passes that are independent of each other in parallel computing different derivatives of a serial application[108], [109] or with nested parallelism for a parallel application[110], [111]. For reverse mode AD, there is an inherent complication. Because data flow is reversed, thread-safe concurrent reads in the primal code become thread-unsafe concurrent writes in the adjoint code. Handling these race conditions efficiently is a key concern. In [112], it was shown that, with source transformation tools in combination with handwritten adjoint routines, it is possible to get efficient parallel implementations for applications.

Another approach is the use of tool-agnostic extension libraries for handling parallelism, which was introduced with the AMPI library[113], [114] for handling MPI constructs combined with traditional AD tools, which has been shown to successfully integrate with the operator overloading library dco/c++[115]. OpDiLib[116] is another tool-agnostic library for shared memory parallelism with OpenMP or hybrid codes with OpenMP and MPI. The other approach is to integrate special treatment of MPI constructs into the AD library, such as in CoDiPack[117] and TAF [118], [119].

OpenMP parallelism is controlled through preprocessor `#pragmas`, which are hard to follow with conventional operator overloading tools since there is no function to overload like in the case of conventional MPI, and tools had to rely on constructors and destructors.

Multiple tools have recently developed extensions for supporting shared memory parallelism. PARAD[120] extends the operator overloading tool Adept to support OpenMP parallel constructs, and the OpDiLib[116], as mentioned earlier, supports OpenMP through either wrapping the directives into macros or the event-based OMPT API. The source transformation tool Tapenade also gained support for some OpenMP directives through a theoretical model[121] and a more complete support for Fortran with a formal extension FormAD[122]. The Enzyme source transformation tool performs the code generation inside the compiler on the LLVM IR level[123], taking advantage of the optimization passes of the compiler prior to the code generation and using rule-based transformations for OpenMP and MPI[124]. PerforAD showed the capability to generate race-free OpenMP for adjoint-based loop transformations for stencil loops[125].

Similarly to the Stan Math Library, using specialized adjoint functions for linear functions like matrix-matrix multiplication, AAD can be applied to CUDA applications efficiently[126]. A handful of tools provide support for CUDA as well. AutoMat[127] shows possibilities for operator overloading for CUDA kernels but heavily depends on recomputing values and lacks high-level checkpointing strategies like Revolve. Enzyme showed the ability to generate adjoint kernels for race-free CUDA kernels using the same mechanisms in the IR[128].

My research aims to provide a performance portable solution for stencil applications by supporting both many-core CPUs through OpenMP shared memory parallelism and GPUs through CUDA from a single high-level application source code taking advantage of the opportunities of the abstraction of the OPS DSL.

1.11 Structure of the dissertation

My research aims to extend the capabilities of existing domain-specific languages to facilitate their use as a performance-portable and future proof solution in the ever changing HPC landscape. Each chapter of the dissertation discusses one area of my research related to improving the two DSLs of the OP-DSL family, briefly introducing the theory and background of the specific area, then describes my scientific contributions supporting my related theses.

Chapter 2 presents my research focusing on the code generation process of DSLs like OP2, the DSL for unstructured meshes. First, I introduce the use of skeletons to minimize the number of lines generated programmatically. Then, I show how the code generation can leverage a modern compiler framework to increase the robustness of the code generation and detail the design and implementation of OP2-Clang, a code generator based on Clang's libTooling. Section 2.3 demonstrates OP2-Clang's extensibility and modularity. I finish the chapter with an analysis of the performance of the generated code.

In Chapter 3, I introduce my research on scalable solver algorithms for batch-tridiagonal problems used in ADI applications. These solver algorithms are integrated into OPS and are used by applications in Chapter 4. Section 3.1 outlines the motivation for high-performance tridiagonal solvers. Section 3.2 starts with a novel distributed memory algorithm for batch-tridiagonal solvers aiming to improve the MPI scalability of such solvers to enable the use of ADI on large-scale CPU and GPU clusters. Then, Section 3.3 analyzes the scaling properties of the implemented solver algorithms and highlights the trade-offs arising at scale.

Chapter 4 focuses on Adjoint-mode Algorithmic Differentiation of stencil applications. Section 1.9 introduces the background of adjoint mode algorithmic differentiation and highlights the core advantages that OPS can build on. Then, I introduce a model of the adjoint computations based on the abstraction used to describe the original computation in OPS. I describe the scheduling of the reverse pass and the code generation used to parallelize the adjoint loops. I extend this model and library with support for computations outside of OPS. Finally, I provide an abstraction to control the memory overhead of the derivative propagation using the Revolve checkpointing strategy.

Finally, Chapter 5 summarizes the contributions of the dissertation. It shows the important methods and tools used and briefly overviews new scientific results and their potential applications.

2 Source-to-source translation for unstructured-mesh applications

This chapter presents my research involving unstructured mesh computations and their mapping to parallel hardware. As Section 1.4 discuss Domain-Specific Languages (DSLs) often use source-to-source translation layers to produce optimized hardware-specific code from a higher abstraction. However, these translation layers often build on fragile techniques (like basic sting manipulation in Python) or exotic compiler ecosystems which leads to a high barrier for adoption. An other problem is that these approaches lack basic tooling support and require large efforts to provide proper diagnostics. My research aims to solve these problems by proposing the integration of the translation layer into the industrial strength Clang/LLVM compiler toolchain. I introduced OP2 in Section 1.3.1 and its abstraction. My research focuses on improving the code generation used by the DSL in two primary regards.

- I introduced a parallelization skeleton-based approach for code generation, drastically reducing the amount of generated code with the main structure of the loops outlined in the skeleton.
- I improved the robustness of the code generation library, moving it from a text-based Python implementation into Clang's compiler infrastructure, taking advantage of the compiler's representation of the source code.

The rest of this chapter is organized as follows. In Section 2.1, I briefly introduce the motivation driving this work, including the limitations of the current source-to-source translation software. In Section 2.2, I chart the design and development of OP2-Clang. Section 2.3 illustrates the ease of extending the tool charting the case for the SIMD-vectorization and CUDA code generator. Section 2.4 shows the performance of the generated parallel code and compares the results to the code generated by the current OP2 source-to-source translator. Section 2.5 details conclusions and potential applications of the results.

2.1 Motivation

Embedded DSLs such as OP2, provide an API embedded in general-purpose languages such as C/C++/Fortran. They rely on source-to-source translation and code refactorization to translate the higher-level API calls to platform-specific parallel implementations. OP2 targets the solution of unstructured-mesh computations, where it can generate a variety of parallel implementations for execution on architectures such as CPUs, GPUs, distributed memory clusters, and heterogeneous processors making use of a wide range of platform-specific optimizations.

Compiler toolchains supporting the source-to-source translation of code written in mainstream languages such as C/C++ or Fortran currently lack the capabilities to carry out such wide-ranging code transformations. Available toolchains such as the ROSE compiler framework [38], [39], [41] and others, have suffered from a lack of adoption by both the compilers and HPC community. This has been a major factor in limiting the wider adoption of DSLs or active libraries with a non-conventional source-to-source translator where there is a lack of a significant community of developers under open governance to maintain it. The result is often a highly complicated tool with a narrow focus, steep learning curve, inflexibility for easy extension to produce different target code (e.g., new optimizations, new parallelizations for different hardware), and issues with long-term maintenance. In other words, the lack of industrial-strength tooling that can be integrated into a DevOps toolchain without any changes provokes a lack of growth in the numbers of users and developers for DSLs or active libraries, which in turn is the source of missing tools themselves, leading to a typical catch-22 or deadlock situation. The underlying motivation of the research presented in this chapter is to break this deadlock by pioneering a methodology using an industrial-strength compiler toolchain that can be integrated as-is in most DevOps environments.

Clang/LLVM's Tooling library (libTooling) [129] has long been touted as facilitating source-to-source translation capabilities but has only demonstrated its use in simple source refactoring and code generation tasks [130], [131] or in the transformation of a very limited subset of algorithms without hardware-specific optimizations [50]. In this chapter, I introduce OP2-Clang [132], a source-to-source translator based on Clang's LibTooling for OP2. The broader aim is to generate platform-specific parallel codes for unstructured-mesh applications written with OP2. Two options to achieve this are: (1) translating programs written with OP2's C/C++ API to code with C++ parallelized with SIMD, OpenMP, CUDA, and their combinations with MPI, etc., that can then be compiled by conventional platform-specific compilers and (2) compiling programs written with OP2's C/C++ API to LLVM Intermediate Representation (IR). The former case, which is the subject of this research, follows OP2's current application development process and has been shown to deliver significant performance improvements. The latter case, which will be explored in future work, opens up opportunities for low-level, application-driven optimizations that would otherwise be unavailable to the source-to-source OP2 solution or to the same application written as a generic C++ program. The development of OP2-Clang also aims to provide additional benefits, which are challenging to implement in OP2's current source-to-source translation layer written in Python. These include full syntax and semantic analysis of OP2 programs with improved user development tools to diagnose and correct errors. Many of such capabilities come for free when using Clang/LLVM. In this research, I chart the development of OP2-Clang, outlining its design and architecture.

2.1.1 Code generation patterns in OP2

The OP2 API was constructed to make it easy for a parsing phase to extract the relevant information about each loop that will describe which computation and memory access patterns will be used - this is required for code generation aimed at different architectures and parallelizations.

The fact that only a few parameters define the parallelization means that in the case of two

Listing 2.1 Skeleton for OpenMP (excerpt) – direct kernels

```
// elemental kernel function
void skeleton(double * __restrict__ d) {}

void op_par_loop_skeleton(char const *name, op_set set, op_arg arg0) {
  /*-------------- number of arguments --------------*/
  int nargs = 1;
  op_arg args[1] = {arg0};
  /*---------------- Invariant code -----------------*/
  int exec_size = op_mpi_halo_exchanges(set, nargs, args);
  #pragma omp parallel for
  for (int n = 0; n < exec_size; n++) {
    if (n == set->core_size)
      op_mpi_wait_all(nargs, args);
  /*-------------------------------------------------*/
    // set up pointers, call elemental kernel
    skeleton(&((double *)arg0.data)[2 * n]);
  }
}
```

Listing 2.2 Skeleton for OpenMP (excerpt) – indirect kernels

```
// elemental kernel function
void skeleton(double * __restrict__ d) {}

void op_par_loop_skeleton(char const *name, op_set set, op_arg arg0) {
  /*-------------- number of arguments --------------*/
  int nargs = 1; op_arg args[1] = {arg0};
  int ninds = 1; op_arg inds[1] = {0};
  /*---------------- Invariant code -----------------*/
  int set_size  = op_mpi_halo_exchanges(set, nargs, args);
  op_plan *Plan = op_plan_get(name, set, 256, nargs, args, ninds, inds);
  int block_offset = 0;
  for (int col = 0; col < Plan->ncolors; col++) {
    if (col == Plan->ncolors_core) {
      op_mpi_wait_all(nargs, args);
    }
    int nblocks = Plan->ncolblk[col];
    #pragma omp parallel for
    for(int blockIdx = 0; blockIdx < nblocks; blockIdx++) {
      int blockId  = Plan->blkmap[blockIdx + block_offset];
      int nelem    = Plan->nelems[blockId];
      int offset_b = Plan->offset[blockId];
      for(int n = offset_b; n < offset_b + nelem; n++) {
  /*-------------------------------------------------*/
        // Prepare indirect accesses
        int map0idx = arg0.map_data[n * arg0.map->dim + 0];
        // set up pointers, call elemental kernel
        skeleton(&((double *)arg0.data)[2 * map0idx]);
      }
    }
  }
}
```

computational loops, the generated parallel loops have the same lines of code with only small code sections with divergences. The identical chunks of code in the generated parallel loops can be considered invariant to the transformation or boilerplate code that should be generated into every parallel implementation without change. However, given that these sections largely define the structure of the generated code, they can be viewed as an important blueprint of the target code to be generated. This leads us to the idea of using a parallel implementation (with the invariant chunks) of a dummy loop and carrying out the code generation process as a refactoring or modification of this parallel loop. In other words, modify the dummy parallel loop as a skeleton (or template) to generate the required candidate computational loop. For example, Listing 2.1 and Listing 2.2 illustrate partial parallel skeletons we can extract for the generated OpenMP implementation for direct and indirect loops.

In a direct loop, all iterations are independent of each other, and as such, the parallelization of such a loop does not have to worry about data races except for global reductions. However, in indirect loops, at least one `op_dat` is accessed using an indirection, i.e., via an `op_map`. Such indirections occur when the `op_dat` is not declared on the set over which the loop is iterating. In which case an `op_map` that provides the connectivity information between the iteration set and the set on which the `op_dat` is declared over is used to access (read or write depending on the access mode) the data. This essentially leads to indirect access.

With indirect loops, we must ensure that no two threads are simultaneously writing to the same data location. This is handled through the invariant code responsible for ordering the loop iterations in Listing 2.2. In this case, OP2 orchestrates the execution of iterations using a coloring scheme that can take data locality into consideration [133].

2.2 Clang LibTooling for OP2 Code Generation

The idea of modifying the target parallelization skeletons forms the basis for the design of OP2-Clang in this work. The alternative would require the full target source generation with the information given in an `op_par_loop`. With such a technique, the variations to be generated for each parallelization and optimization would have made the code generator prohibitively laborious to develop and even more problematic to extend and maintain. The skeletons simply allow us to reuse code and allow the code generator to concentrate on the parts that need to be customized for each loop, optimization, target architecture, and so on. As such, we use multiple skeletons for each parallelization. In most cases, one skeleton for direct and one for indirect kernels are used, given the considerable differences in direct and indirect loops as given in Listing 2.1 and Listing 2.2. The aim, as mentioned before, is to avoid significant structural transformation.

Clang's Tooling library (libTooling) provides a convenient API to perform a range of varying operations over source code. As such, its capabilities lend very well to the tasks of refactoring and source code modification of a parallelization skeleton in OP2. The starting point of source-to-source translation in OP2-Clang is to make use of Clang to parse and generate an Abstract Syntax Tree (AST) of the code with OP2 API calls. The generated AST is used to collect the required information of the sets, maps, data including their types and sizes, and information in each of the parallel loops that make up the application. The second phase involves transforming the skeletons with the information for each parallel loop. The two phases of code generation

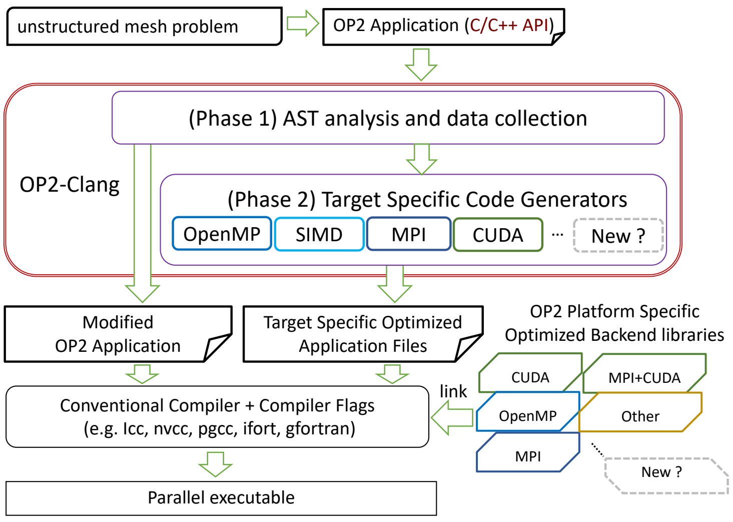

unstructured mesh problem
OP2 Application (C/C++ API)
(Phase 1) AST analysis and data collection
OP2-Clang
(Phase 2) Target Specific Code Generators
OpenMP
SIMD
MPI
CUDA
...
New ?
Modified OP2 Application
Target Specific Optimized Application Files
OP2 Platform Specific Optimized Backend libraries
CUDA
MPI+CUDA
OpenMP
Other
MPI
New ?
link
Conventional Compiler + Compiler Flags (e.g. Icc, nvcc, pgcc, ifort, gfortran)
Parallel executable

Figure 2.1: The high-level architecture of OP2-Clang and its place within OP2

and where they fit into the overall architecture of OP2 are illustrated in Figure 2.1. The output of OP2-Clang will be compiled by conventional compilers, linking with OP2's platform-specific libraries for a given parallelization to produce the final parallel executable. Each parallel version is generated separately.

2.2.1 OP2-Clang Application Processor

The first phase of OP2-Clang is responsible for collecting all data about the parallel loops, or kernel calls, used in the application. This step will also perform semantic checks based on the types of the `op_dat`s. Particularly check whether the declared `op_dat`s match the types declared in the `op_par_loop`. However, such checks are not currently implemented but will be very straightforward, given that all the information is present in the AST. The data collection and model creation happens along the OP2 API calls. As the parser builds the AST with the help of `ASTMatchers` [134], OP2-Clang keeps a record of the kernel calls, global constants, and global variable declarations that are also accessible inside kernels.

During the data collection, this phase also carries out a number of modifications to the application-level files that contain the OP2 API calls. Essentially, replacing the `op_par_loop` calls with the function calls that implement the specific parallel implementations. Of course, these implementations are yet to be generated in the second phase of code generation.

2.2.2 OP2-Clang Target Specific Code Generators

The second phase is responsible for generating the target-specific optimized parallel implementations for each kernel whose details are now collected within OP2-Clang. Given the information of each of the `op_par_loop`s the parallelization (currently one of OpenMP, CUDA, SIMD, or MPI), we are generating code for and whether the loop is a direct or indirect loop, a target skeleton can be selected.

The target code generation, then, is a matter of changing the selected skeleton at appropriate places, using the properties of the candidate `op_par_loop` for which the generator will generate

Listing 2.3 Left: Parallelization skeleton for MPI (excerpt) **Right:** Generated MPI parallelization (excerpt)

```c
// elemental kernel function
void skeleton(double *__restrict__ d){
}




void op_par_loop_skeleton(
    char const *name, op_set set,
    op_arg arg0) {
 /* ----- number of arguments ---- */
 int nargs = 1;
 op_arg args[1] = {arg0};

 /* ------- Invariant code ------- */
 int exec_size=op_mpi_halo_exchanges(
                   set, nargs, args);
 for (int n = 0; n < exec_size; n++){
   if (n == set->core_size)
     op_mpi_wait_all(nargs, args);
 /* ------------------------------ */

   // prepare indirect accesses
   int map0idx =
    arg0.map_data[n*arg0.map->dim+0];



   // set up pointers, call kernel
   skeleton(
    &((double*)arg0.data)[2*map0idx]);

 }
 // invariant code
 ...
}
```

```c
// elemental kernel function
void res(double * __restrict__ edge,
         double * __restrict__ cell0,
         double * __restrict__ cell1){
 *cell0 += *edge; *cell1 += *edge;
}

void op_par_loop_res(char const *name,
           op_set set, op_arg arg0,
           op_arg arg1, op_arg arg2) {
 /* ----- number of arguments ---- */
 int nargs = 3;
 op_arg args[3] = {arg0, arg1, arg2};

 /* ------- Invariant code ------- */
 int exec_size=op_mpi_halo_exchanges(
                   set, nargs, args);
 for (int n = 0; n < exec_size; n++){
   if (n == set->core_size)
     op_mpi_wait_all(nargs, args);
 /* ------------------------------ */

   // prepare indirect accesses
   int map0idx =
    arg1.map_data[n*arg1.map->dim+0];
   int map1idx =
    arg1.map_data[n*arg1.map->dim+1];

   // set up pointers, call kernel
   res(&((double*)arg0.data)[n],
    &((double*)arg1.data)[2*map0idx],
    &((double*)arg1.data)[2*map1idx]);
 }
 // invariant code
 ...
}
```

a parallelization. For example, consider the `op_par_loop` in Listing 1.1. This loop is an indirect loop with three arguments, one of which is a directly accessed `op_dat` and two of which are indirectly accessed `op_dats`. The loop iterates over the set of `edges`, and the elemental computation kernel is given by `res`. Listing 2.3 shows the simplest skeleton in OP2-Clang, the skeleton for generating MPI parallelization, and the specific code that needs to be generated by changing the skeleton for the above loop. The elemental kernel function needs to be set to `res`, the number of arguments set to three while handling the indirections by using the mappings specified for those arguments, and finally, the elemental kernel should be called by passing in the appropriate arguments.

This type of transformation rhymes well with Clang's RefactoringTool[135], [136], which can apply replacements to the source code based on the AST. As such, the process of modifying the skeleton first creates the AST of the skeleton by running it through Clang. Then, the AST is searched for points of interest (i.e., points where the skeleton needs to be modified). The search is again done using `ASTMatchers`, which, in principle, are descriptions of AST nodes of interest. For example, to set the specific elemental kernel function from `skeleton` to `res` in Listing 2.3, OP2-Clang needs to find the function call in the AST (part of which is given in Listing 2.4) using the matcher given in Listing 2.5. The definitions of the matchers are trivial, given that the skeleton is an input to the above process.

Listing 2.4 Elemental function call in the AST of the skeleton

```
FunctionDecl op_par_loop_skeleton
...
`-CallExpr 'void'
  |-DeclRefExpr 'void (double *)' lvalue
    Function 'skeleton'
  `-UnaryOperator 'double *' prefix '&'
    `-ArraySubscriptExpr 'double' lvalue
      ( ... )
```

Listing 2.5 ASTMatcher to match the AST node for the elemental function call

```
StatementMatcher functionCallMatcher =
callExpr(
  callee(functionDecl(hasName("skeleton"), parameterCountIs(1))),
         hasAncestor(
           functionDecl(hasName("op_par_loop_skeleton"))))
  .bind("function_call");
```

To formulate all such modifications to the skeleton, I create a set of matchers and run them through the `OP2RefactoringTool`, which is derived from the base class `RefactoringTool` in LibTooling. When `OP2RefactoringTool` with the specific collection of matchers are run through the AST of the skeleton, the matchers find the AST nodes of interest, create a `MatchResult` object containing all the information for the given match, invokes a callback function with the `MatchResult` where we can identify which matcher found a match with its keys. The identified calling matcher provides the AST node of interest and the corresponding source location. This allows the generation of the specific replacement code in a `Replacement`[137] just as it is shown in Listing 2.6. The replacement is not immediately done but is collected in a map. Once all the AST nodes of interest are matched and the replacement strings collected, together with the source locations, they can be applied to the source code. In this case, the collection of `Replacements` are checked to ascertain if the replacements are independent of each other so that they can be applied to the source without errors. This finalizes and commits the changes to the skeleton.

As it can be seen, the process does not do large structural transformations. All the changes on the skeleton can be formulated as replacements for single lines or small source ranges. The adoption of skeletons leads to another implicit benefit. As the code generation requires the AST of the skeleton built, the skeleton must be valid C/C++ code, which, combined with the fact that we apply relatively small modifications to the code, implies that the generated code can also be guaranteed to be valid C/C++. Any errors can only come from small chunks of generated code, which is easy for the OP2-Clang developer to debug. This is a significant benefit during the development of a new code generator. One of the key difficulties of OP2's current Python-based translator is the lack of support for such debugging tasks to ascertain that valid C/C++ code is generated.

Listing 2.6 Example of creating a `Replacement` to replace the elemental function call. The `Result` object is the `MatchFinder::MatchResult` object created by the match of the ASTMatcher from Listing 2.5.

```
CallExpr *match = Result.Nodes.getNodeAs<CallExpr>("func_call");
SourceLocation begin = match->getLocStart();
SourceLocation end = match->getLocEnd();
SourceRange matchRange(begin, end);
string replacement = "res(&((double *)arg0.data)[n],"
                     "&((double *)arg1.data)[2 * map0idx],"
                     "&((double *)arg1.data)[2 * map1idx]);"
Replacement replacement(*Result.SourceManager,
                        CharSourceRange(matchRange, false),
                        replacement);
```

2.3 Extensibility and Modularity

The underlying aim of OP2 is to create a performance portable application written using the OP2 API. A further objective is to "future-proof" the higher-level science application where only the code generation needs to be extended to translate it to be able to execute on new hardware platforms. As such, the target code generation, using OP2's Python-based translator, currently supports a range of parallelizations with optimizations tailored to the underlying hardware to gain near-optimal performance. The new OP2-Clang source-to-source translator will also need to support generating all these parallelizations and optimizations but also be easy to extend to implement new parallelizations and optimizations. In this section, we present further evidence of OP2-Clang's capabilities for modular and extensible code generation for a number of architectures, carrying out different optimizations.

Since the data collection phase is independent of the target code generation, OP2-Clang only needs to add a new target-specific code generator to support generating code for a new platform or to use a new parallel programming model. As we have seen, a target-specific code generator consists of (1) a parallel skeleton (usually one skeleton for implementing direct loops and one for indirect loops), (2) a list of matchers that identify AST nodes of interest, and (3) a corresponding list of Replacements that specify the changes to the code.

While the skeletons already modularize and enable the reuse of code, the matchers and the corresponding `Replacements` can also be reused. Thus, when developing a new code generator, we reuse existing `ASTMatchers` and `Replacements` as required. Only the matchers and `Replacements` that do not exist need to be created from scratch.

While Clang's `ASTMatchers` is extensive, there are some AST nodes that don't have the specialized matchers we require. For example, there were currently no matchers to match nodes representing the OpenMP constructs. I had to extend the list of available matchers with a single matcher to match the `omp parallel for` pragma in order to be able to perform our translation

Listing 2.7 The modified version of the elemental function `res`, to use strided memory accesses wit SoA data layout.

```
__constant__ intdirect_res_stride_OP2CONSTANT;
__constant__ intopDat1_res_stride_OP2CONSTANT;
__device__ void res_calc_gpu(const double * __restrict edge,
                             double * __restrict cell0,
                             double * __restrict cell1){
  cell0[0 * opDat1_res_stride_OP2CONSTANT] +=
       edge[0 * direct_res_stride_OP2CONSTANT];
  cell1[0 * opDat1_res_stride_OP2CONSTANT] +=
       edge[0 * direct_res_stride_OP2CONSTANT];
}
```

conveniently. In this section, I look at several other challenges that had to be overcome when supporting the various code generators for OP2.

2.3.1 CUDA

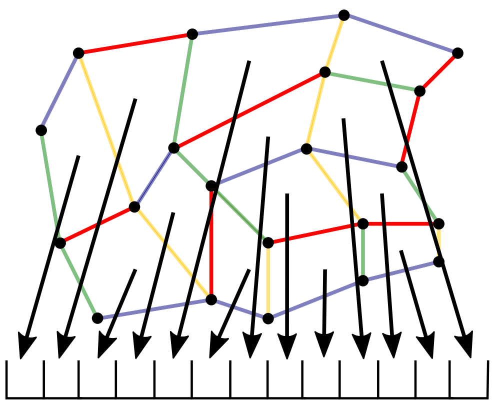
The CUDA parallelization presented a number of challenges in code generation. Again, much of the code generation using a skeleton followed a similar process to that of the OpenMP parallelization. However, the skeleton was larger and required considerably more replacements. Nevertheless, the steps taken to do the replacements were the same. For CUDA, it was necessary to accurately set the device pointers and create a CUDA kernel call that encapsulates the elemental function. OP2 handles the data movement between the device and the host in the backend by copying the data to device arrays and updating host arrays if necessary. Therefore, this aspect does not affect the code generation.

The main challenge with CUDA was implementing a number of optimizations that significantly impact the performance on GPUs, unlike the previous parallelizations that utilize CPUs. First among these is memory access; the memory access pattern of CPUs is not optimal for GPUs. For example, to gain coalesced memory accesses on GPUs, OP2 can restructure the data arrays to make the neighboring threads read data next to each other in memory. On CPUs with large caches, it is beneficial to organize data in an Array of Structures (AoS) layout, which maximizes data reuse. However, on GPUs, the threads are performing the same operation on consecutive set elements at the same time. Organizing the data such that the data needed by consecutive threads are next to each other is more beneficial. In this case, we can read one cache line and use all of the data in it. This data layout is called a Structure of Arrays (SoA) layout. To use a SoA layout, OP2-Clang needs to change the indexing inside the elemental function to use a strided accesses pattern. The generated elemental function for executing `res` in CUDA is illustrated in Listing 2.7.

Another way to improve the memory access patterns in CUDA is to modify the coloring strategy used for indirect kernels. The coloring in the OpenMP parallelization is done such that no two threads with the same color write to the same data locations. Then, iterations with the same color can be run in parallel. Applying this strategy to CUDA means that thread blocks need to be colored, and no two thread blocks will write to the same data as shown in Figure 2.2a for an example kernel reading data on the edges and writing data on the cells. However, for

Listing 2.8 CUDA kernel with global coloring (excerpt)

```
// CUDA kernel function
__global__ void op_cuda_res(const double *__restrict arg0,
           double *ind_arg0, double *ind_arg1,
           const int *__restrict opDat0Map, int start,
           int end, const int * __restrict col_reord,
           int set_size) {
  int tid = threadIdx.x + blockIdx.x * blockDim.x;
  if (tid + start >= end) return;
  int n = col_reord[tid + start];
  int map0idx = opDat0Map[n + set_size * 0];
  int map1idx = opDat0Map[n + set_size * 1];
  res_calc_gpu(arg0 + n, ind_arg0 + map0idx * 1,
               ind_arg1 + map1idx * 1);
}
```


(a) Global coloring approach. In each kernel launch, the kernels work on edges of the same color. Arrows corresponding to block red.

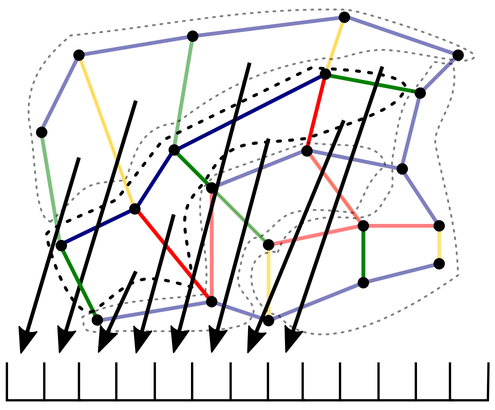

(b) Hierarchical coloring approach. The thread blocks are circled with dashed lines, and the edge colors show the color inside the block.

Figure 2.2: Schematic figure of different coloring strategies used in OP2. The arrows represent the individual pieces of data loaded indirectly when executing the block.

CUDA, a further level of coloring could improve the performance [J1]. In this case, the threads within a thread block are also colored to avoid data races. This two-level coloring is called hierarchical coloring. Hierarchical coloring has been shown to improve data locality and data reuse inside CUDA blocks considerably.

Figure 2.2b shows the effect of the hierarchical coloring on the amount of memory loaded by the block to the shared memory. This coloring approach can take advantage of the memory locality coming from the natural order of the mesh elements that form the CUDA thread blocks. To further improve the memory locality or, in other words, reduce the amount of memory loaded by each block (and increase the reuse of the loaded memory), one can apply renumbering techniques on the mesh itself. The renumbering of the mesh can increase the memory locality, but in return, it also increases the number of colors inside the block, which increases the number of synchronizations required to execute the block. This trade-off is explored in more detail in [J1], and the results can be applied to the generated code, but optimizing the mesh is out of the scope of this work on the code generation. The difference in the CUDA kernel function between the

Listing 2.9 CUDA kernel with hierarchical coloring (excerpt)

```
// CUDA kernel function
void op_cuda_res(const double *__restrict arg0, double *ind_arg0,
     double *ind_arg1, constint *__restrict opDat0Map,
     int block_offset, int *blkmap, int *offset, int *nelems,
     int *ncolors, int *colors, int nblocks, int set_size) {
  __shared__ int ncolor, nelem, offset_b;

  if (blockIdx.x >= nblocks) return;
  double arg1_l[1] = {0.0}, arg2_l[1] = {0.0};
  if (threadIdx.x == 0) {
    int blockId = blkmap[blockIdx.x + block_offset];
    nelem = nelems[blockId];
    offset_b = offset[blockId];
    ncolor = ncolors[blockId];
  }
  __synchthreads();

  int col2 = -1, n = threadIdx.x;
  if (n < nelem) {
    res_calc_gpu(arg0 + offset_b + n, arg1_l, arg2_l);
    col2 = colors[n + offset_b];
  }
  for (int col = 0; col < ncolor; col++) {
    if (col2 == col) {
      int map0idx = opDat0Map[n+offset_b+set_size*0];
      int map1idx = opDat0Map[n+offset_b+set_size*1];
      ind_arg0[0 + map0idx * 1] += arg1_l[0];
      ind_arg1[0 + map1idx * 1] += arg2_l[0];
    }
    __synchthreads();
  }
}
```

two coloring strategies is shown in Listing 2.8 and Listing 2.9. The variations to the code to be generated can simply be captured again with a different skeleton, in this case, a skeleton that does the hierarchical coloring. However, the required Replacements, including the data layout transformations (AoS to SoA), can be reused for both one-level coloring and hierarchical coloring skeletons.

2.3.2 SIMD vectorization

A more involved code generation task is required for SIMD vectorization on CPUs. For vectorization, OP2 attempts to parallelize over the iteration set of the loop [26]. The idea is to generate code that will be automatically vectorized when compiled using a conventional C/C++ compiler such as `icpc`. Listing 2.10 illustrates the code that needs to be generated by OP2-Clang to achieve vectorization for our example loop `res`. There are two key differences here, compared to non-vectorizable code as in Listing 2.3: (1) the use of gather/scatters when indirect increments are applied and (2) the use of a modified elemental function in the vectorized loop. The first is motivated due to the multiple iterations (equivalent to the SIMD vector length of the CPU) that are carried out simultaneously. In this case, we need to be careful when indirect writes are performed. In each iteration, we perform a gather of the required data into local arrays, then perform the computation on the local copies, and then we perform a scatter to write back the updated values. The gathers and the execution of the kernel are vectorized with `#pragma omp simd`, the scatter cannot be vectorized due to data races and so is executed serially. Finally, the remainder of the iteration set needs to be completed. Much of the code in Listing 2.10 can be generated as discussed previously. However, now we must also modify the internals of the elemental function `res` to produce a vectorizable elemental kernel `res_vec`. As illustrated, the function signature needs to be changed, which in turn requires modifications to the data accesses inside the function body (i.e., the computational kernel).

In order to perform these transformations, we introduced a further layer of refactoring, which parses the elemental function and transforms it into the vectorized version. Since the elemental function consists of the kernel that each iteration of the loop performs, the scope of these transformations is limited. Even the indexing of the arrays is done in the generated code that calls the elemental function. To perform the necessary changes to the elemental function, the function itself is passed through Clang to obtain its AST, and matchers are used to identify the AST nodes in the function signature and replace them with the correct array subscript (e.g., `*edge` is changed to `edge[*][SIMD_VEC]`). Again `ASTMatchers` are used to identify AST nodes within the elemental kernel, replacing them with the variable with array subscripts (`edge[0]` is changed to `edge[0][i]`). Simple dereferences are replaced with `[0][simd_vec]` indexing. The match and replacements here are only different because we are now modifying the elemental kernel itself and not a skeleton.

2.4 Evaluation and Performance

In contrast to the OP2-Clang translator, OP2's current Python-based translator only parses the application source to identify OP2 API calls. However, no AST is created as a result, but simply the specifications and arguments in each of the `op_par_loop` calls are collected and stored in Python lists. When it comes to generating code, the full source of what is to be generated is produced using the information gathered in these lists for each parallelization. No text replacements are done as in the OP2-Clang translator. However, as the invariant code for a given parallelization can be generated without change, only the specific changes for a given `op_par_loop` need to be produced. Again, the code generation stage does not use an AST. As such, changing code within elemental kernels (as in the SIMD vectorization case) is significantly more difficult and cannot easily handle how users write their elemental kernels. All of the above makes the Python translator error-prone and difficult to extend and maintain. In this section, I present some results from evaluating the OP2-Clang translator on two OP2 applications by comparing the code generated from it to the performance of the code generated through OP2's current Python-based translator.

Both of the applications used in these tests have loops with indirections and indirect increments, various global reductions, and the use of global constants.

Listing 2.10 Vectorized loop for `res` (excerpt).

```
// vectorized elemental function
inline void res_vec(const double edge[*][SIMD_VEC],
                    double cell0[*][SIMD_VEC],
                    double cell1[*][SIMD_VEC], int i) {
  cell0[0][i] += edge[0][i]; cell1[0][i] += edge[0][i];
}

#pragma novector
for(int n=0; n<(exec_size/SIMD_VEC)*SIMD_VEC; n+=SIMD_VEC) {
  double arg0_p[1][VEC]; double arg1_p[1][VEC]; double arg2_p[1][VEC];
  // gather data to local variables
  #pragma omp simd
  for ( int i = 0; i < SIMD_VEC; i++ ) {
    arg0_p[0][i] = (ptr0)[idx0_2 + 0];
    arg1_p[0][i] = 0.0; arg2_p[0][i] = 0.0;
  }
  // vectorized elemental function call
  #pragma omp simd
  for ( int i = 0; i < SIMD_VEC; i++ ) {
    res_vec(arg0_p, arg1_p, arg2_p, i);
  }
  // Scatter indirect increments
  for ( int i = 0; i < SIMD_VEC; i++ ) {
    int map0idx = arg1.map_data[(n + i) * arg1.map->dim + 0];
    int map1idx = arg1.map_data[(n + i) * arg1.map->dim + 1];
    ((double *)arg1.data)[2 * map0idx] += arg1_p[i];
    ((double *)arg2.data)[2 * map1idx] += arg2_p[i];
  }
}
// remainder loop
for ( int n = 0; n < exec_size; n++ ) {
  int map0idx = arg1.map_data[n * arg1.map->dim + 0];
  int map1idx = arg1.map_data[n * arg1.map->dim + 1];
  res(&((double *)arg0.data)[n], &((double *)arg1.data)[2 * map0idx],
    &((double *)arg1.data)[2 * map1idx]);
}
```

2.4.1 The Airfoil Application

The first application, Airfoil, is a benchmark application representative of large industrial CFD applications utilized by users of OP2. It is a non-linear 2D inviscid airfoil code that uses an unstructured grid and a finite-volume discretization to solve the 2D Euler equations using scalar numerical dissipation[138]. The algorithm iterates towards the steady state solution in each iteration using a control volume approach, meaning the change in a cell's mass is equal to the net flux along the four edges of the cell, which requires indirect connections between cells and edges. Airfoil is implemented using OP2, where two versions exist, one implemented with OP2's C/C++ API and the other using OP2's Fortran API [139], [140].

The application consists of five parallel loops: **save_soln**, **adt_calc**, **res_calc**, **bres_calc** and **update** [12]. The **save_soln** loop iterates through cells and is a simple loop accessing two arrays directly. It copies every four state variables of cells from the first array to the second one. The **adt_calc** kernel also iterates on cells, and it computes the local area/timestep for every single cell. For the computation, it reads values from nodes indirectly and writes in a direct way. There are some computationally expensive operations (such as square roots) performed in this kernel. The **res_calc** loop is the most complex loop with both indirect reads and writes; it iterates through edges and computes the flux through them. It is called 2000 times during the total execution of the application and performs about 100 floating-point operations per mesh edge. The **bres_calc** loop is similar to **res_calc** but computes the flux for boundary edges. Finally, **update** is a direct kernel that includes a global reduction, which computes a root mean square error over the cells and updates the state variables.

The mesh used in our experiments consists of over 2.8 million node cells and about 5.8 million edges. In the most computationally intensive loop from the five, about 100 floating-point operations are performed per mesh edge. This makes the code memory bandwidth bound and a good candidate to showcase the effect of the reduced memory movement used in hierarchical coloring.

2.4.2 The Volna Application

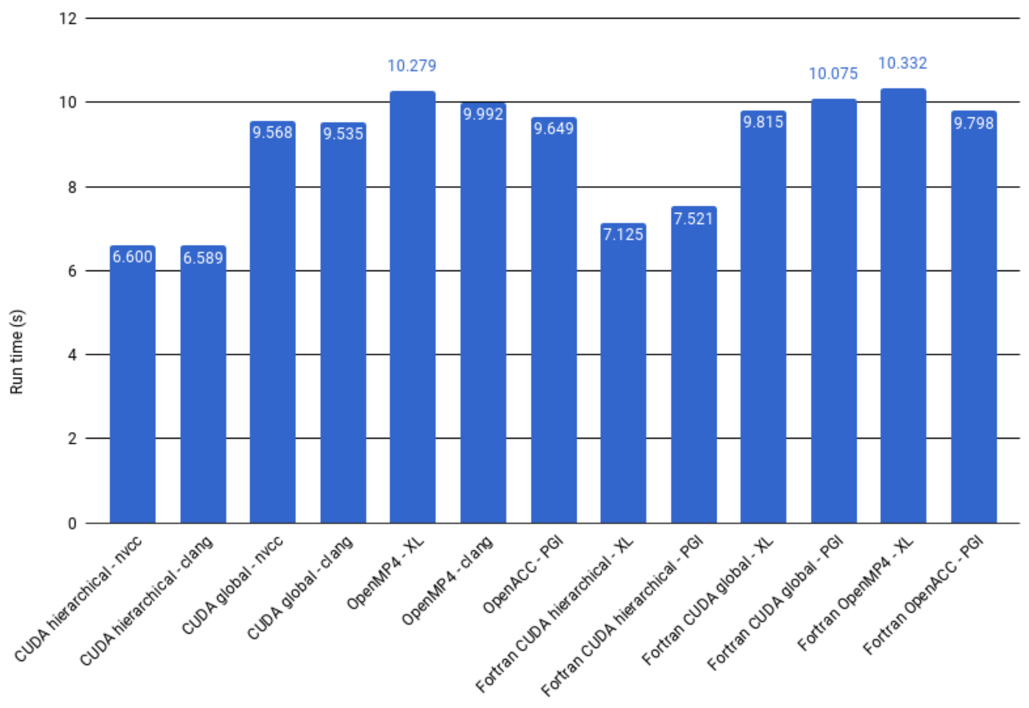
The second application, Volna, is a shallow water simulation capable of handling the complete life-cycle of a tsunami (generation, propagation, and run-up along the coast) [24], [141]. The simulation algorithm works on $2.5D$ unstructured triangular meshes and uses the finite volume method. Volna is written in C/C++ and converted to use the OP2 library[140]. For Volna, the top three kernels where the most time is spent: **computeFluxes**, **SpaceDiscretization** and **NumericalFluxes**. In the **computeFluxes** kernel, there are indirect reads and direct writes, in **NumericalFluxes**, there are indirect reads with direct writes and a global reduction for calculating the minimum timestep and in **SpaceDiscretization** there are indirect reads and indirect increments.

Tests are executed in single precision on a mesh containing 2.4 million triangular cells and about 3.5 million edges, simulating a tsunami run-up to the US Pacific coast.

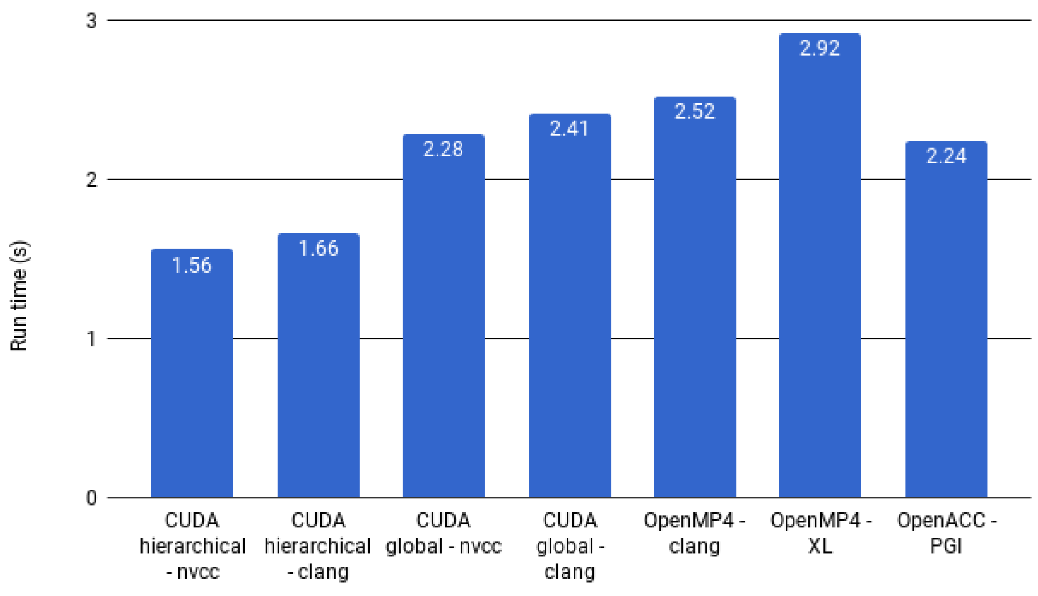
2.4.3 Performance

	SIMD	OpenMP	CUDA Global (AoS)	CUDA Global (SoA)	CUDA Hier. (AoS)	CUDA Hier. (SoA)
Airfoil	363.92 s (-0.2%)	70.417 s (1.2%)	12.77 s (-0.6%)	9.58 s (-0.4%)	9.85 s (0.2%)	7.30 s (1.8%)
Volna	95.39 s (0.3%)	14.84 s (-0.2%)	3.00 s (0.5%)	2.33 s (0.2%)	2.32 s (1.2%)	1.97 s (1.1%)

Table 2.1: Performance of Airfoil and Volna on the Intel Xeon E5-1660 CPU (for OpenMP and SIMD) and on an NVIDIA P100 GPU with OP2-Clang. CUDA results with two different colorings (global and hierarchical) and two data layouts (AoS and SoA) presented. The values in parenthesis are the percentage difference in run time compared to the sources generated with OP2's current Python-based source-to-source translator (negative values mean OP2-Clang has better performance).



(a) Measured run times of versions for the Airfoil application.



(b) Measured run times of versions for the Volna application.

Figure 2.3: Absolute performance comparison of the Airfoil and Volna application on a P100 GPU using different coloring and parallelization approaches.

The generated code was compiled and executed on a single Intel Xeon CPU E5-1660 node (total of 8 cores) for OpenMP and SIMD vectorization (using Intel compilers suite 17.0.3.) and a single NVIDIA P100 GPU with CUDA 9.0. Table 2.1 shows the performance results and percentage difference of runtime compared to OP2's current Python-based translator. In all cases, the performance difference is less than 2%. This figure was the same when comparing the run times of each kernel. These results, therefore, give an initial indication that identical code was generated by OP2-Clang.

The run times of different versions of Airfoil on a P100 GPU are shown in Figure 2.3a. The hierarchical coloring is used in `res_calc` and `bres_calc`, because these have indirect increments. The versions using the hierarchical coloring scheme have the best performance due to the huge performance gains in `res_calc` thanks to data reuse. The main difference between versions with the same coloring strategy is in the run times of the `res_calc` and `adt_calc` kernels, where most of the computation is performed. The Fortran versions suffer from high register pressure, leading to lower occupancy in general, and directive-based approaches have the same problem, especially in `adt_calc` [C1].

For Volna, the **SpaceDiscretization** kernel has a huge impact on runtime (half of the time is spent in this kernel when using global coloring), and so the hierarchical coloring leads to significant overall performance gain as shown on Figure 2.3b (the measurements are in single precision because Volna requires only single precision to get correct results).

2.5 Conclusions

In this chapter, I introduced OP2-Clang, a source-to-source translator based on LibTooling, for OP2. OP2-Clang is capable of parsing a higher-level declarative program written in OP2's C/C++ API and generating parallel code based on SIMD, OpenMP, CUDA, and their combinations with MPI. The wide range of transformations required for generating code for each parallelization in OP2 and the variation in specific optimizations for each are significant, going well beyond what has been previously demonstrated with LibTooling. We presented the use of parallelization skeletons to reuse code and demonstrated the use of LibTooling's `ASTMathchers` and `Replacements` to modify a skeleton to generate the necessary parallel code. Multiple levels of refactoring using LibTooling's `RefactoringTool` enables the application of specific optimizations in a flexible, maintainable, and extensible manner. Challenges in developing OP2-Clang were presented, discussing the generation of MPI, OpenMP, SIMD vectorized code, and CUDA code for CPUs and GPUs. Performance from the OP2-Clang generated code showed near-identical performance to the code generated by OP2's current source-to-source translator (based on Python). We believe that the lessons learned from OP2-Clang can be readily applied in developing similar source-to-source translators, particularly for DSLs.

3 Scalable Batch-Tridiagonal solver algorithms

In this chapter, I present my research involving linear solvers for ADI applications. Section 1.7 highlights the limitations of the current solutions availible for ADI applications. A large 3D ADI application requires solution of a large number of independent tridiagonal equation systems in each spatial dimension for every time iteration. It is not feasible to restrict the domain decompositions along any dimension or require data transposition for certain dimensions on large clusters. However, most of the state-of-the-art libraries use similar restrictions or provide limited scalability as the number of nodes increase. My goal is to provide exact scalable solver algorithms for batch-tridiagonal systems for large-scale HPC clusters for ADI applications. I explore solver algorithms and the algorithmic trade-offs required at increasing machine scale. My work does not focus on improving the performance of CPU or GPU kernels solving partial systems, as this was done in prior work.

In Section 1.6 I introduced the core algorithms for the solution of tridiagonal systems. This chapter is divided into two parts. The first introduces the extension of the tridiagonal solver algorithms to distributed memory-based systems and shows the implications to performance at increasing scale. In the second part, the best implementations are used to solve a number of large-scale problems, analyzing performance on CPU and GPU clusters.

3.1 Motivation

Tridiagonal systems of equations arise in numerous fields, particularly as part of the numerical approximation of multidimensional Partial Differential Equation (PDEs). They frequently appear in CFD, Computational Electro-Magnetics (CEM), computational finance, and image processing. In some cases in computational finance, the use of the ADI time discretization is preferred leading to the need for the solution of multiple tridiagonal systems of equations in each dimension [59], [142]–[144]. In CFD, tridiagonal systems form the core component for using implicit techniques [145] with application in solving incompressible fluid flow problems [146] and design of turbomachinery [147] among others. Since there are large numbers of independent systems to be solved in multiple dimensions, they offer significant opportunities for exploiting the massive parallelism available on modern multi-core CPU and many-core GPU devices. With the advent of such hardware, recent work [65] re-examined the choice between different tridiagonal solution algorithms (Thomas [60], Parallel Cyclic Reduction (PCR) [62] and Hybrid [65]) and showed that high performance is achievable on both shared memory environments like CPUs and GPUs with platform-specific optimizations. However, many real-world problems require such algorithms to work efficiently over multiple CPU/GPU devices due to the need for compute and memory resources beyond a single node.

A good example is high-fidelity simulations such as the ones performed with the Xcom-

	Communication	Number of messages	Message size
Thomas-Thomas(AG)	All-to-All	1	$2 \times N_p \times N_{sys}$
Thomas-Thomas(GS)	All-to-All	2	$\frac{2 \times N_p \times N_{sys}}{N_p}$
Thomas-PCR	One-to-One	$2 \times log_2 N_p + 2$	N_{sys}
Thomas-Jacobi	One-to-One	$2 \times J + 2$	N_{sys}

Table 3.1: Communication steps needed to solve the reduced system for each algorithm. N_p is the number of processes that share the same set of tridiagonal systems N_{sys} is the number of independent systems the processes share. J is the number of Jacobi iterations required. The message size is shown in terms of elements, each element requires to send the corresponding a_i, c_i, d_i coefficients ($b_i = 1$).

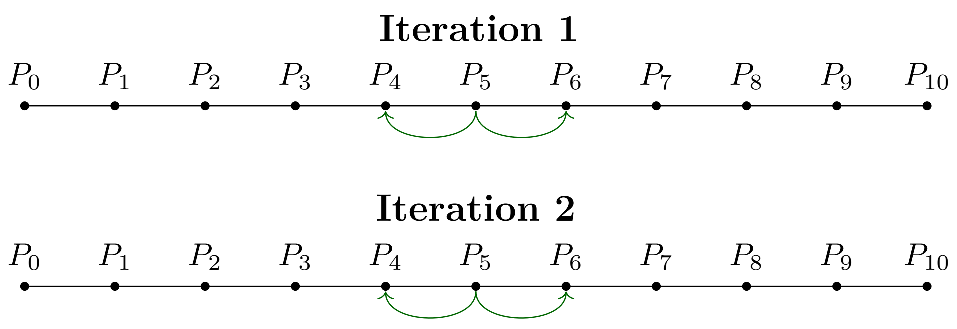
pact3d [148] framework, requiring the solution of up to 150 batches of tridiagonal systems at each time step to compute derivatives and interpolations using implicit high-order finite-difference schemes. For a production problem with 1024^3 mesh nodes, which represents, for instance, a wind farm of several kilometers squared with a mesh node every 2 meters, this would require a cluster with more than 80 GPUs to solve 1024^2 systems, each with a length of 1024 for a single batched solve (of which there are 150 per timestep). Looking at exascale systems, such simulations will be based on 100-1000 billion mesh nodes and performed with 10-100 million cores for hundreds of thousands of time steps.

Such problems mean that tridiagonal solver algorithms over distributed memory for these multi-core/many-core devices still require careful investigation in terms of performance and especially scalability. This is imperative in the current age of exascale systems in HPC, where the software capabilities of exploiting such systems crucially depend on the scalability of numerical simulation applications and their underpinning algorithms. In my research, I investigate the state-of-the-art multi-core/many-core algorithms for tridiagonal solvers for distributed-memory systems and re-examine the algorithmic trade-offs required at increasing machine scale to achieve good performance. Based on this research I introduce a new, highly scalable algorithm and implementation that extends the single-node shared memory work of László et al [65] to distributed memory CPU and GPU clusters that are essential for solving modern state-of-the-art problems.

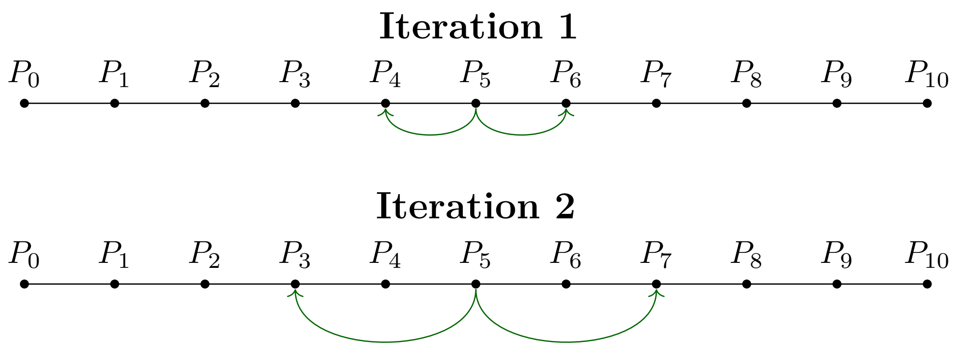
3.2 Distributed Memory Algorithms

The new distributed memory tridiagonal solver builds on the hybrid Thomas-PCR algorithm detailed in the Section 1.6. I implemented multiple variations of this hybrid algorithm, but the overall structure of the distributed tridiagonal solver can be summarized as follows. Each subsystem of size M belongs to a separate MPI process, which performs the hybrid Thomas-* forward pass. This produces a reduced system with two rows per MPI process. The solution to the reduced system is implemented in a number of ways, resulting in different performance characteristics over distributed memory systems. Once the reduced system is solved, the backward pass of the hybrid Thomas-* is performed on each MPI process.

The reduced system can be solved using several strategies in terms of memory movement, as well. The reduced system can be gathered into a single MPI process, which then solves it and scatters the results back to the other MPI processes. This *gather-scatter* (GS) implementation


Iteration 1
P_0 P_1 P_2 P_3 P_4 P_5 P_6 P_7 P_8 P_9 P_{10}
Iteration 2
P_0 P_1 P_2 P_3 P_4 P_5 P_6 P_7 P_8 P_9 P_{10}

(a) Constant communication pattern of Jacobi iterations.


Iteration 1
P_0 P_1 P_2 P_3 P_4 P_5 P_6 P_7 P_8 P_9 P_{10}
Iteration 2
P_0 P_1 P_2 P_3 P_4 P_5 P_6 P_7 P_8 P_9 P_{10}

(b) Communications in PCR, distance doubles in every iteration.

Figure 3.1: MPI communications using Jacobi and PCR on the reduced system. Each arrow highlights a message sent with the values owned by the process, and the P5 node also receives messages from the same processes.

can also be slightly modified to obtain an *allgather* (AG) implementation where the reduced system is gathered onto all MPI processes and then solved on each process. AG removes the need for the scattering of the results. Both GS and AG require excessive global communications, which naturally leads to poor scaling.

The PCR or Jacobi methods can be used to avoid global collectives. *PCR* would follow the same algorithm as described previously but with the addition of point-to-point MPI communications during each iteration of the algorithm. Therefore, it will carry out MPI communications with processes successively further away for the later iterations of the PCR algorithm. Figure 3.1b shows the pattern for the communication for the first two iterations. A further alternative is to use the iterative *Jacobi* method on the reduced system, similar to the TridiagLU implementation, to obtain an approximate solution to the reduced system. Again, there is the option to provide an estimated number of iterations or to check for convergence. Using the Jacobi iterations for the reduced system has the advantage that it only requires MPI processes to communicate with their neighbors. Figure 3.1 compares this communication pattern to the communications in PCR. To summarize the different solutions Table 3.1 shows the number and size of the messages required for each solver algorithm. While the message size for the two iterative solution is the same, and each iteration requires communication with two additional nodes the key difference is the number of messages and the locality of those messages. While Jacobi might require more iterations after the messages in PCR are sent to other physical nodes in the cluster, communications for the Jacobi iterations become cheaper compared to the PCR iterations. However, if required

Algorithm 6 `forward_sweep`(a, b, c, d)

```
1:  $d_1^* \leftarrow d_1/b_1$
2:  $c_1^* \leftarrow c_1/b_1$
3:  $a_1^* \leftarrow a_1/b_1$
4:  $b_0^* \leftarrow b_0 - c_0 a_1^*$
5:  $d_0^* \leftarrow d_0 - c_0 d_1^*$
6:  $c_0^* \leftarrow -c_0 c_1^*$
7:  for $i = 2, 3, ..., M-2$ do
8:      $r \leftarrow 1/(b_i - a_i c_{i-1}^*)$
9:      $d_i^* \leftarrow r(d_i - a_i d_{i-1}^*)$
10:     $a_i^* \leftarrow -r a_i a_{i-1}^*$
11:     $c_i^* \leftarrow r c_i$
12:     $b_0^* \leftarrow b_0^* - c_0^* a_i^*$
13:     $d_0^* \leftarrow d_0^* - c_0^* d_i^*$
14:     $c_0^* \leftarrow -c_0^* c_i^*$
15: end for
16: $r \leftarrow 1/(b_{M-1} - a_{M-1} c_{M-2}^*)$
17: $d_{M-1}^* \leftarrow r(d_{M-1} - a_{M-1} d_{M-2}^*)$
18: $a_{M-1}^* \leftarrow -r a_{M-1} a_{M-2}^*$
19: $c_{M-1}^* \leftarrow r c_{M-1}$
20: $d_0^* \leftarrow d_0^*/b_0^*$
21: $c_0^* \leftarrow c_0^*/b_0^*$
22: $a_0^* \leftarrow a_0/b_0^*$
23: return $a^*, c^*, d^*$
```

to check for convergence, then its near neighbor communication advantage gets nullified as a global collective communication is needed for each iteration (or every n number of iterations). Considering the above approaches, the key advantage of PCR for the reduced system is that it avoids the need for collective communications while at the same time providing an exact solution.

An improvement to the forward pass of the hybrid Thomas algorithm (Algorithm 3) is to combine it with the forward of TridiagLU [76]. Algorithm 6 normalizes each row and forms the a^* column fromFigure 1.6 and c_0^* for a subdomain resulting in a reduced system with one row per subdomain. This relaxes the need to express each unknown in terms of u_0 and u_{M-1}, instead each $u_i, i \in 1, ...M-1$ will be expressed with u_0 and u_{i+1}:

$$a_i^* u_0 + u_i + c_i^* u_{i+1} = d_i^*, \quad i = 1, 2, \ldots, M-1.$$

Although this introduces dependencies inside a partition, the backward substitution pass still requires only a single sweep without extra memory movements. Both the original and modified algorithms are trivially scalable since there is no communication involved. The reduced computational cost of the forward pass and the smaller reduced system size (one row per MPI process instead of two) leads to smaller messages and better overall performance. Hence, the Tridsolver library uses Algorithm 6 on GPUs when the reduced system is solved with the Jacobi or PCR methods.

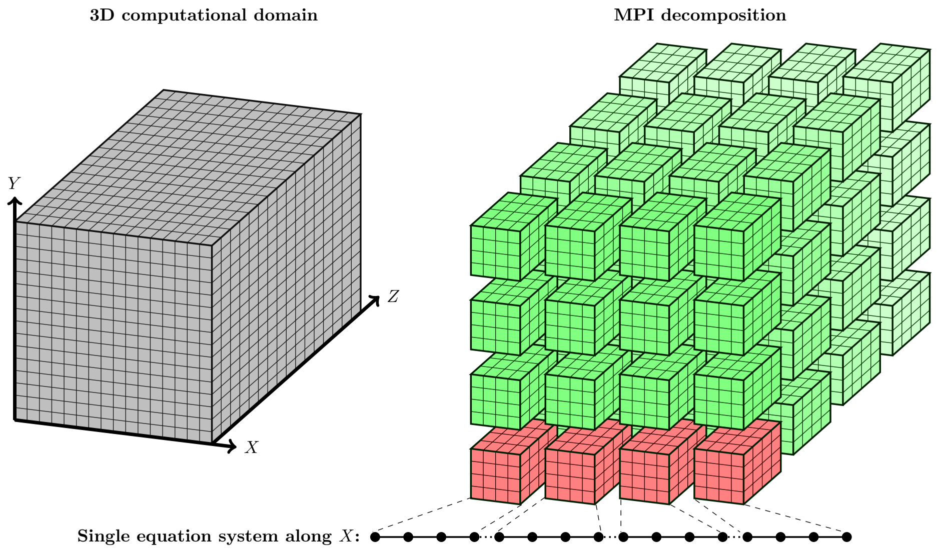


Figure 3.2: MPI decomposition of a 3D computational domain along all three spatial dimension. Each small grid represents the computational domain owned be a single MPI process. The nodes highlighted by red share tridiagonal systems along the X axis.

3.2.1 Tridiagonal systems in 3D applications

A tridiagonal system, by its nature, represents a one-dimensional problem, however, applications of interest are commonly 2 or 3-dimensional. These applications would use these higher dimensional structured grids as the computational domains and distribute these grids evenly along all spatial dimensions of the computational domain. Figure 3.2 shows such a decomposition on a 3D domain.

Tridiagonal systems are formed in these applications by solving along one of the coordinate axes - and there will be as many *independent* systems as there are discretization points along the other axes. For example, the ADI [59] method, preferred in computational finance, works by repeatedly solving tridiagonal systems along different axes. If we take a problem along the X axis N_yN_z (where N_y and N_z is the number of discretization points owned by the process in Y and Z dimension respectively) independent systems is shared among the processes highlighted by red in Figure 3.2. In ADI, the a_i, b_i, c_i, d_i coefficients are calculated for each grid point in a way that matches the underlying data structure of the application; data for the diagonals are stored contiguously in either a row-major (Z is contiguous, Y and X are strided) or more commonly, as Figure 3.3 shows, a column-major (X is contiguous, Y and Z are strided) format. This poses a challenge for algorithms that then solve multiple tridiagonal systems simultaneously; coefficients for an individual system will be laid out differently, depending on the direction of the solve. If we solve along the X direction (and use a column-major format), for example, then coefficients for the same system are contiguous in memory, followed by the coefficients for the next system, etc. If, however, we solve along the Z direction, then coefficients for the same row of different systems are laid out contiguously, followed by the next row, etc.

The TridiagLU library can only handle a data layout corresponding to a Z-solve as described above, and so in a 3D application, one would have to appropriately transpose data for X and Y solves - an operation notoriously difficult to do efficiently due to inefficient memory access patterns. On the other hand, the tridiagonal solver library developed in this work (which we

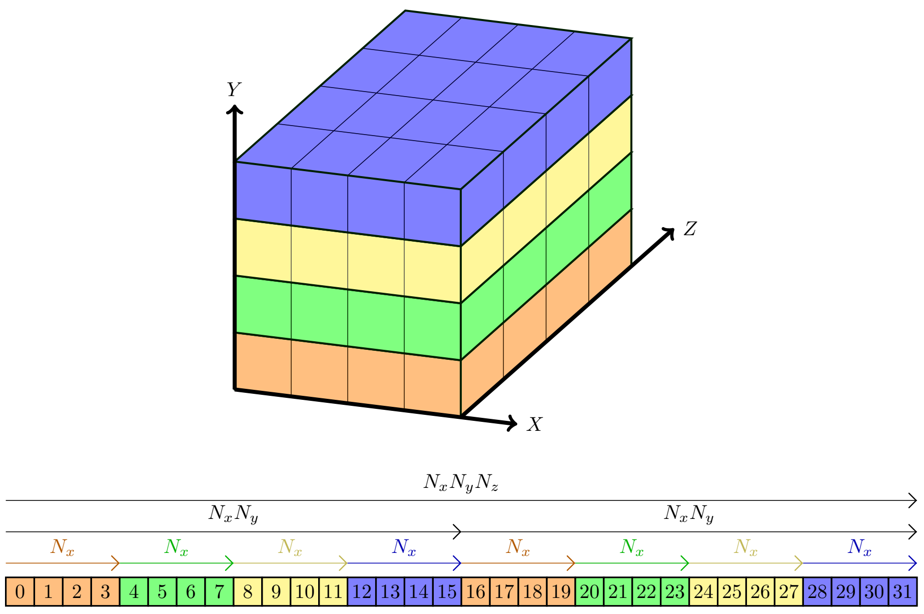


Figure 3.3: Example memory layout of the coefficients stored in column-major format within a MPI process for a 3D domain. Values in X stored contiguously, Y is stored with stride N_x and Z is stored with stride $N_x * N_y$.

henceforth call the *Tridsolver* library), was designed from the outset to handle higher-dimensional applications carrying out 1D solves in different directions - of course as with the approaches of László et al [65] the implementations are still impacted by memory access patterns.

A number of optimizations help improve performance on modern systems. On CPUs the Y and Z dimension solves (see next section) can be vectorized by splitting the tridiagonal systems into strips of consecutive memory and adding compiler pragma `omp simd` on the appropriate loops. On the GPU, a key issue is the uncoalesced memory accesses in the X dimension. Figure 3.3 shows that the coefficients corresponding to a single system are stored contiguously in memory. Each GPU thread works on a single system. Loading the data for the coefficients of each system in a natural way lead uncoalesced data accesses (each thread reads data with stride N_x). Local transposition using vector shuffles [65] provides a solution where threads of a warp cooperate to read a 32×16 or 32×8 (depending on single or double precision mathematics) block of the YZ plane at once. This corresponds to either 16 or 8 elements of 32 neighboring tridiagonal systems. After loading this block of data, the elements of the tridiagonal system are not necessarily held by the thread solving that system. The *__shfl_xor_sync()* intrinsic is then used to swap the elements to the correct threads. A similar operation is performed in reverse when storing intermediate values and the solution of the tridiagonal systems. Figure 3.4 shows both the original method of loading a block of memory, where each CUDA thread only loads its own tridiagonal system, and the new cooperative memory strategy used to achieve coalesced memory accesses.

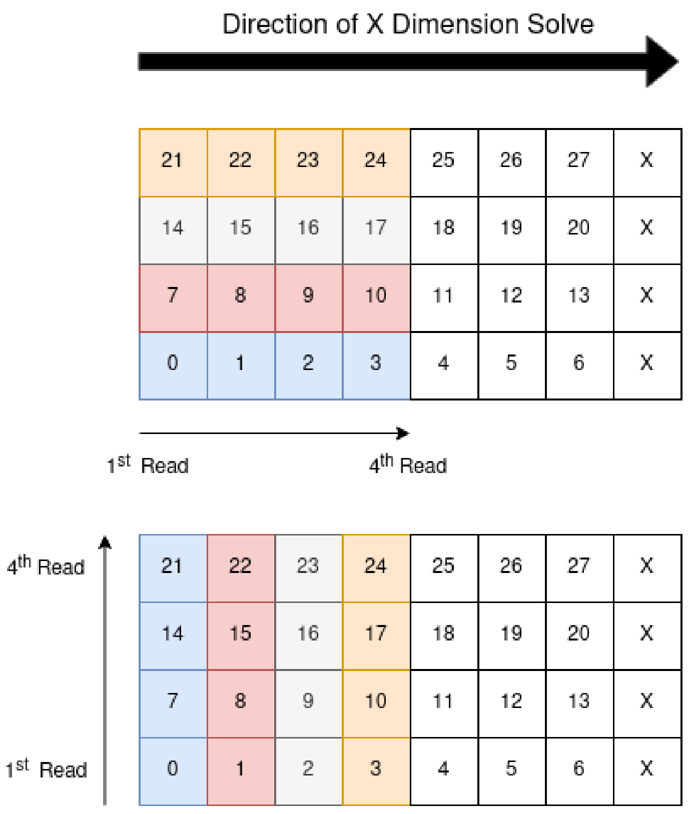


Figure 3.4: Memory access patterns during X solves for CUDA threads. The X dimension is stored contiguously as the indices highlight and the rows are representing independent systems. Each thread loads a single block at a time. **Above:** the original memory reads where each CUDA thread (shown in a different color) only loads the values for its own tridiagonal system, leading to uncoalesced reads (first read instruction loads indices: $0, 7, 14, 21$). **Below:** the data is loaded using coalesced memory access. This pattern results in faster loads, but the threads get data from different systems. Extra vector shuffles are required for each thread to get the corresponding coefficients.

3.3 Evaluation and Performance

To study the performance and scalability of the Tridsolver library, I designed benchmarks (published with the repository) for two of the UK's HPC systems: ARCHER2[1], a CrayEX system with AMD Rome CPUs (2×64 cores per node) and 256 GB of RAM, and Cirrus[2], a HPE/SGI system with 36 GPU nodes, each with 4×NVIDIA V100 16GB GPUs, interconnected with NVLink, and FDR Infiniband between nodes.

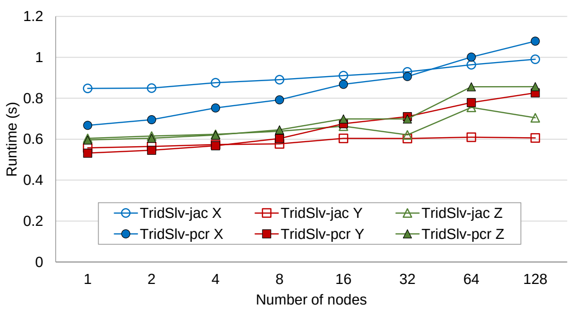
All measurements showed in this section is a result of average of 10 runs. As a baseline, I compared against the TridiagLU library on the CPU - which only supports distributed memory parallelism with MPI. For 3D problems, TridiagLU assumes the same coefficients from different systems are packed together, corresponding to a Z solve. I include an extra copy of the a, b, c coefficient arrays in our timing as these are overwritten by the solve algorithm, but the original values are required by our applications. For the solution of the reduced system, I evaluated both

[1]https://www.archer2.ac.uk/
[2]https://www.cirrus.ac.uk/

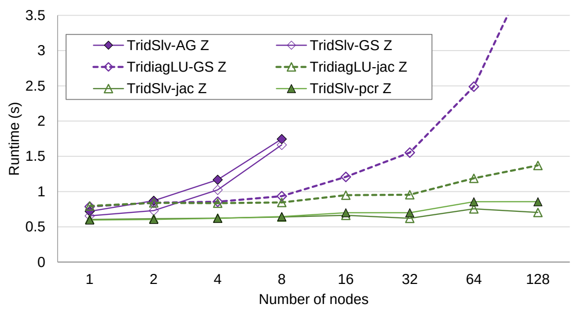
the exact solver approach with Gather-Scatter and the approximate iterative approach (Jacobi), which includes Allreduce calls to determine whether the solution has converged.

I evaluated the performance of our library (marked with TridSlv) utilizing, for the reduced system solve, both exact solution approaches (with AG, GS, PCR) as well as the approximate iterative approach (Jacobi), including convergence checks. For the solution of the reduced system, I evaluated solutions in all directions, and directly compared to TridiagLU in Z. To avoid differences in convergence at increasing scale, I fixed the number of Jacobi iterations at 10 as done by Ghosh et al [76].

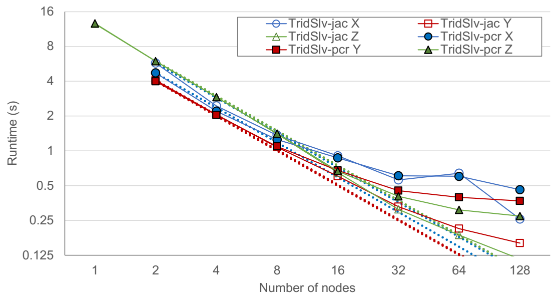
3.3.1 Weak Scaling on CPUs - ARCHER2



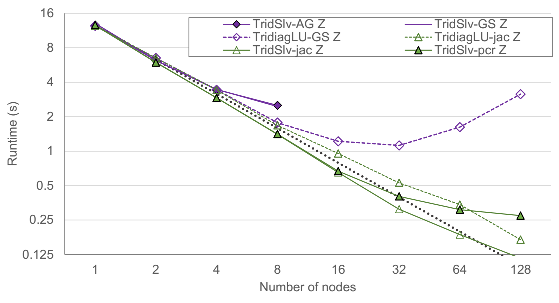
(a) Tridsolver (TridSlv) weak scaling in X, Y and Z dimensions



(b) Tridsolver (TridSlv) vs TridiagLU weak scaling in Z dimension



(c) Tridsolver (TridSlv) strong scaling in X, Y and Z dimensions



(d) Tridsolver (TridSlv) vs TridiagLU strong scaling in Z dimension

Figure 3.5: ARCHER2 scaling: (a),(b) - Weak-scaling, 512^3 grid points per node. (c),(d) - Strong-scaling, 8192 points in the direction of solve, and 512 in others. AG - AllGather, GS - Gather-Scatter. The dotted lines in the strong scaling plots show ideal scaling performance.

For weak scaling, where problem size increases with machine size, I picked 512^3 grid points per ARCHER2 node, a typical problem size used by frameworks such as Xcompact3D. Currently, ARCHER2 only has 1024 nodes in 4 cabinets, and considering that tridiagonal solves in various directions are completely independent, I tested weak scalability only along one "line" of nodes: for X solve $N \times 1 \times 1$ nodes and $(N * 512) \times 512 \times 512$ grid points, for Y solve $1 \times \ N \times 1$ nodes and $512 \times (N * 512) \times 512$ grid points, and for Z solve $1 \times 1 \times \ N$ nodes and $512 \times 512 \times (N * 512)$ grid points. With pure MPI, we have 128 processes per node, which we distribute $4 \times 4 \times 8$ along the X, Y, and Z directions respectively.

Weak scaling performance along different directions varies significantly (see Figure 3.5a) - the X solve is the least amenable to parallelization and vectorization, with the Thomas algorithm

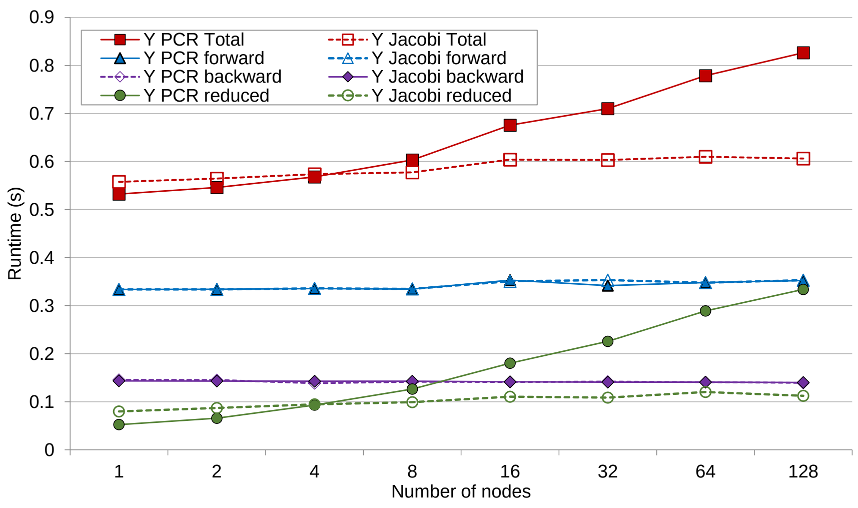
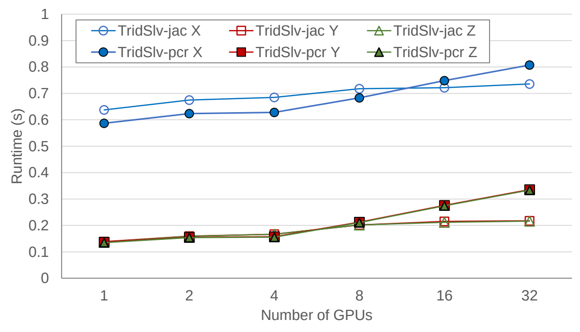


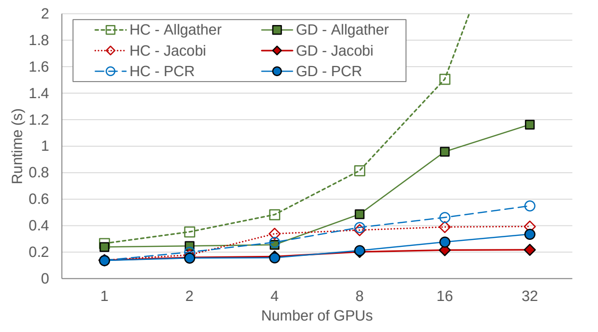
Figure 3.6: TridSlv (Y-dim) weak scaling runtime breakdown, ARCHER2: 512^3 per node

being up to 1.7× slower than the Y solve due to the diagonals for each system is contiguous in memory when solving along the X dimension. The Y and Z solves lend themselves to trivial parallelization. However, as the algorithm steps from row to row, the corresponding coefficients are separated by larger strides (more so in the case of Z) leading to degraded TLB performance, as documented before [65]. The performance on a single node heavily depends on the possible memory bandwidth. On a single ARCHER2 node, the hybrid Thomas forward step achieves 270.2 GB/s for X and Y solve, and 225 GB/s for Z solve the backward achieves 307.7 GB/s, 275.8 GB/s, and 268.5 GB/s for X, Y, and Z solves, respectively. Although the theoretical maximum for the AMD Rome CPU is 204.8 GB/s per socket, the Triad (simple addition kernel) kernel in the BabelStream [149] benchmark achieves 288 GB/s as well. Comparing different solvers, as expected, the Tridsolver Allgather (AG) and Gather-Scatter (GS) approaches (Figure 3.5b) have poor scaling efficiency (60%). This scaling behaviour comes down to two factors, the first is the all-to-all nature of the communication and the second is the fact that the amount of data that participates in the communication scales with the number of processes as Table 3.1 shows. While TridiagLU GS scales somewhat better (48-94%) due to the distributed nature of reduced system solves, scaling efficiency remains low due to high communication costs. In contrast, the Jacobi approximate solver (jac) has excellent scalability (90-98%) due to its low-volume neighbor-to-neighbor communication patterns. The Tridsolver with PCR for reduced system solve comes very close to Jacobi in terms of scaling efficiency - only falling behind at larger node counts. This is due to PCR having overall worse (long-distance) communication patterns (see Figure 3.1b), but as Table 3.1 shows number of messages scale logarithmically with the number of processes along the solve dimension.

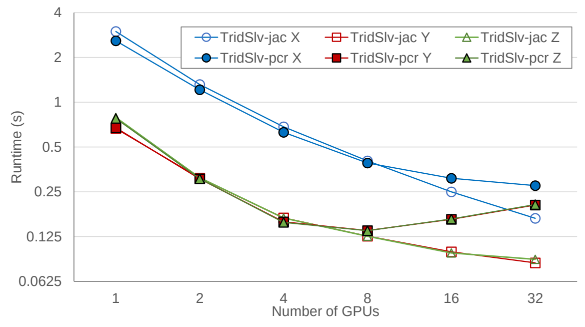
Investigating performance in more detail, we see that since forward and backward steps involve no communication, they scale trivially: Figure 3.6 shows that during Y solve, total runtime starts at 0.52 seconds and increases to 0.62 for Jacobi and 0.83 for PCR at 128 nodes. The Forward step takes 0.34 seconds ($< 58\%$), and the backward takes 0.15 seconds ($< 27\%$). The reduced system solve step takes $0.10 - 0.14$ seconds for 10 iterations of the Jacobi solver, with very little increase ($92 - 96\%$) in time when scaling. In contrast, the same step takes $0.06 - 0.33$ seconds with PCR, increasing steadily with a $70 - 74\%$ scaling efficiency.



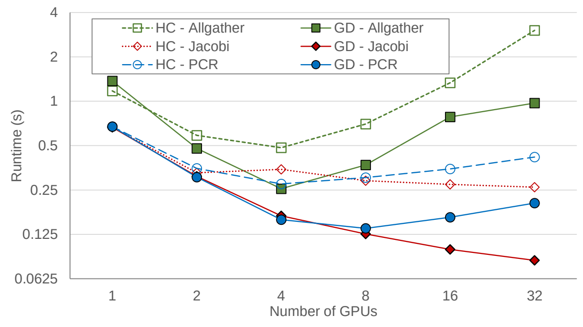
(a) Tridsolver (TridSlv) weak scaling in X, Y, and Z-dims



(b) Tridsolver (TridSlv), weak scaling, Host Copies vs GPU-Direct



(c) Tridsolver (TridSlv), strong scaling in X, Y, and Z-dims



(d) Tridsolver (TridSlv), strong scaling, Host Copies vs GPU-Direct

Figure 3.7: Cirrus scaling (MPI+CUDA) :(a),(b) - Weak-scaling, 512^3 points per GPU. (c),(d) - Strong-scaling, 2048 points in the direction of solve, 512 points in others. HC - host copy, GD- GPU direct.

3.3.2 Strong scaling on CPUs - ARCHER2

For strong scaling, where a large single global problem is solved at increasing machine scale, I used a grid size of 8192 in the direction of the solve and 512 in other directions, allowing us to scale from 1 node to 128 nodes. I assigned $4 \times 4 \times 8$ processes per node in the X, Y, and Z directions, respectively. Figure 3.5c shows the results in different directions and Figure 3.5d compares different solvers. Here, the logarithmic scale for the y axis reduces the visibility of the difference between the X solve and other solves. Nevertheless, it is consistent with the slowdown observed when weak scaling. Even superlinear scaling (102-108%) can be observed on up to 8 nodes, with both Jacobi and PCR, owing to the continuously reducing number of TLB and LLC misses. However, at larger scales, communication costs dominate; using Jacobi for the reduced solve, efficiency drops to 80% above 32 nodes, and using PCR to 82-56% above 32 nodes. At 128 nodes for the Z solve, the reduced system solve phase accounts for 60% of total time with Jacobi and 85% with PCR.

Comparing the scalability of different algorithms in Figure 3.5d, we see that the Tridsolver AG and GS variants slow down early and actually run out of memory due to the large size of the reduced system. The PCR solver shows competitive scaling compared to the approximate Jacobi method - it is within a factor of 2 and scales further than the TridiagLU library.

3.3.3 Weak Scaling on GPUs - Cirrus

For weak scaling on GPUs, I kept the problem size per GPU at 512^3 in order to compare with the CPU results and scaled the problem in the direction of the solve, only performing MPI decomposition in that direction. I did not compare against other libraries as I am not aware of other MPI-enabled GPU tridiagonal solver libraries. Tridsolver implementations support any MPI distribution by copying data to the host (I used MPT 2.22) and then making transfers between CPUs, as well as GPU Direct-enabled MPI distributions (I used OpenMPI 4.1.0), where transfers take place directly between GPUs. The best results were achieved with GPU Direct.

Looking at results from Cirrus in Figure 3.7a, I compare the performance when using Jacobi and PCR variants to solve the reduced system in different directions (note that Y and Z solves are virtually indistinguishable on the plot). As for the CPU, the performance of X solves is degraded by poor memory access patterns. On a single GPU, X solve in Tridsolver achieves 458 GB/s bandwidth, while the Y and Z solves achieve 731 GB/s and 739 GB/s, respectively (the Triad kernel in the BabelStream [149] benchmark achieves 821 GB/s on a single V100 GPU). I compared Tridsolver on a single V100 GPU to the batch tridiagonal solver functions in the cuSPARSE. Currently, cuSPARSE has two batch-tridiagonal solver functions: `cusparse<t>gtsv2StridedBatch()` uses the same memory layout as the X solve in Tridsolver and achieves 525.5 GB/s, while `cusparse<t>gtsvInterleacedBatch()` comparable to the Z solve and achieves 725.6 GB/s. On Cirrus, there is a marked decline in parallel efficiency beyond 4 GPUs. Up to 4 GPUs, communications are performed via the high-speed NVLink interconnect, but beyond that, there is inter-node communication through a slower Infiniband connection. Studying performance breakdowns in more detail, we see that in the case of the Y and Z solves, total time takes between 0.13 – 0.21 (Jacobi) or 0.13 – 0.33 (PCR) seconds. The computational part takes 0.13 seconds (forward and backward steps). For up to 4 GPUs, the cost of communications is less than 8% and 4% of total runtime for Jacobi and PCR, respectively. However, beyond 4 GPUs, this increases to 22-27% for Jacobi and 27-53% for PCR.

Figure 3.7b compares the performance of the baseline MPI implementation using explicit host copies (HC) with using GPU Direct (GD) - I also show the scalability of the Allgather (AG) version. AG clearly shows the impact of increasing communication volume as the number of GPUs increase, leading to dramatic slowdowns. On Jacobi and PCR, the GD version is up to $1.69\times$ faster. Note that on a single GPU, the PCR and Jacobi versions are $1.8\times$ faster than the AG version. Since on a single node, there is no need for a reduced system solve, the performance improvement is purely due to the adoption of the newly improved Thomas forward and backward pass.

Overall, we see that a single GPU is $4.6\times$ faster than a single ARCHER2 node running with a pure MPI parallelization in the Y and Z directions. This difference arises from the overhead of the MPI communication on the CPU node and the bandwidth limitations of the two hardware. At 32 GPUs/nodes, this is reduced to $3\times$ for Jacobi and $2.1\times$ for PCR due to the comparatively worse communications scaling on the Cirrus GPU cluster.

3.3.4 Strong Scaling on GPUs - Cirrus

The largest problem that can fit in a single GPU has 2048 points in the direction of solve and 512 in others - which then can be strong-scaled up to 32 GPUs. Results in Figure 3.7c detail again the Y and Z solves only showing marginal differences in performance. As before, we see a drop in scaling efficiency beyond 4 GPUs, which are in a single node interconnected with NVLink; over 93% up to 4 GPUs, then 55-66% for Jacobi and 39-57% for PCR. As observed during weak scaling, communications become more of a bottleneck for the PCR solver: at 32 GPUs, 86% of total time is communications, compared to Jacobi's 72%. The differences between HC and GD versions are even more prominent in strong scaling (see Figure 3.7d). GD is up to 3.25× faster.

3.4 Conclusion

In this chapter, I investigated the state-of-the-art in multi-core/many-core algorithms for tridiagonal solvers for distributed-memory systems and re-examined the algorithmic trade-offs for obtaining better scaling and runtime performance at increasing machine scale. The exploration led to the development of an improved distributed-memory solver with scalable performance for a large number of MPI nodes based on the hybrid Thomas-PCR algorithm, giving exact solutions to the problem by extending and augmenting a previous single-node library to execute over clusters of CPUs and GPUs. Further developments led to the implementation of a new, improved Thomas-PCR forward pass and integrating iterative techniques based on a Jacobi solver, which provided approximate solutions that can be used as an option for the solution of the reduced system resulting on the boundaries of MPI partitions.

Performance evaluation on a CrayEX system showed superior performance on realistic problem sizes, specifically for ADI applications. The new solver with the Jacobi solver for the reduced system obtained 90-98% scaling efficiency. However, solving the reduced system with the PCR algorithm provided competitive performance. It achieved almost perfect scaling and tested up to 16 ARCHER2 nodes along the solve dimension, with the added advantage of providing an exact solution.

Execution on a GPU cluster demonstrated that the Jacobi and PCR solvers (for the reduced system solve) scaled with 93% efficiency up to 4 GPUs due to the high bandwidth single node interconnect. However, efficiency reduced to 55-66% for Jacobi and 39-57% for PCR beyond this point. Further optimizations with a modified Thomas-PCR forward pass algorithm improved performance with a speedup of 1.8×.

The new tridiagonal solver library is integrated into the OPS domain-specific language [11] for the solution of structured-mesh problems. This extends OPS's capabilities with implicit solutions on top of its existing explicit solvers used in frameworks such as OpenSBLI [35].

4 Adjoint mode Algorithmic Differentiation with OPS

In this chapter, I present my research using domain-specific languages for computing derivative information of scientific applications. Derivatives are crucial for various engineering and scientific applications such as optimization, machine learning, and inverse problems. While finite difference approximations can estimate derivatives, they are computationally expensive and inaccurate. Algorithmic (Automatic) Differentiation (AD) provides an efficient and exact method to compute derivatives by treating computer programs as mathematical functions and applying the chain rule of calculus.

My work focuses on adjoint mode AD for structured mesh stencil applications. I introduce an extension to the Oxford Parallel Library for Structured mesh solvers (OPS) domain-specific language to compute adjoints using OpenMP and CUDA parallelism. Using a DSL provides multiple advantages for the applications. The introduction of AAD to OPS can provide a performance-portable implementation of the derivative calculation. In the modern landscape of HPC, where more and more specialized hardware arises to meet the computational needs of scientific computations, performance-portability and hardware-agnostic abstractions are the only way to provide future-proof solutions for applications. OPS allows developers to express mesh algorithms from a high-level code targeting multiple hardware from the same source with the potential of getting support for future hardware through OPS.

My research aims to provide a performance portable solution for stencil applications by supporting both many-core CPUs through OpenMP shared memory parallelism and GPUs through CUDA from a single high-level application source code. The user defines the application in the abstraction of OPS, provides the loop bodies and their adjoints as functions, and the library generates all the parallel implementations for the application. To enable AD for an OPS application, the original application code requires minimal changes (such as seeding and accessing derivatives) and the adjoints of the loop bodies. Many of the tools mentioned above depend on large tapes to follow the control flow of the applications. In OPS, the code transformation and optimization step for the parallel implementations happens at compile time with additional augmentation to build the tape. At runtime, OPS builds a highly compact tape of the entire simulation where each entry in the tape represents a parallel loop or an external function (see Section 4.1.3).

In Section 4.1, I present a model for reverse mode differentiation of complex stencil applications using OpenMP and CUDA from the same source expressed in the OPS domain-specific language. I introduce an AD tape to the OPS DSL with elements handled on the computational kernel level, allowing more compact storage thanks to the OPS abstraction. I show that this tape makes the implementation of extensions, such as optimal AD checkpointing via Revolve[150] or arbitrary "external" (to OPS) adjoint functions, such as linear solvers, reasonably straightforward. In Section 4.2, I introduce the mapping of the high-level description of computational loops to the parallel implementation of the adjoint loops. I show that extensive transformations and optimizations on the parallel implementation of the adjoint loops are feasible with the information provided by the DSL. Finally, in Section 4.3, the experimental results for three applications implemented in OPS are shown: a simple Poisson equation solver, the Cloverleaf hydrodynamics mini-application, and a Convection-Diffusion Equation (CDE) solver code. I present detailed performance of the OpenMP and CUDA implementations. I show performance results for two implementations to compute adjoints for the Poisson application, one with traditional implementation and one with the application expressed as a fix point iteration. Furthermore, I show the performance of Revolve as the checkpointing algorithm for Cloverleaf and a Convection-Diffusion Equation (CDE) application.

4.1 Reverse mode Algorithmic Differentiation in OPS

This chapter focuses on adjoint mode differentiation of stencil computations in the Oxford Parallel Library for Structured mesh solvers (OPS) [11], [13] domain-specific language.

I introduced OPS in Section 1.3.2 and described the main cornerstones of the abstraction and the API.

OPS supports multiple parallelization models and targets from the same high-level application code using code transformation techniques to provide efficient and optimized implementations for all supported targets. This approach naturally creates performance-portable code bases for the applications. Extending a DSL like OPS with AD support enables a new class of applications to take full advantage of the performance-portability and optimizations provided by the DSL. The OPS library itself can generate code based on the additional information and restrictions of the domain, which enables diagnostics and optimizations that would require extensive code analysis or are just impossible for general-purpose tools. The ability to follow data movement and dependencies throughout the application is not only critical for some of the optimizations OPS can provide, but it also plays a crucial role in adopting reverse mode AD in OPS. The code generation is the key to future-proof and make the applications performance-portable. In addition the code generation also grants us an entry point to insert the proper functionalities to build a tape in OPS. OPS also reduces the adjoint AD memory overhead of storing the control flow by using a high-level representation of the computational steps.

Our extension for OPS uses reverse-mode AD and does not support forward-mode AD; from this point on, the discussion focuses on reverse-mode AD.

OPS follows the control flow through the `ops_par_loop` calls (such as in Listing 1.2). These calls refer to computational kernels, listing all accessed datasets and the access patterns. This makes them the perfect candidate to become the building block of the control flow graph in OPS.

Note that the ownership of all datasets is handed to the library and can only be accessed through OPS APIs, which allows OPS to keep track of the state of all datasets and when it is required to move the data, for example, between CPUs and GPUs. The ability to follow data movement and dependencies throughout the application has already shown to be a critical factor

Listing 4.1 The user given adjoint loop body for the kernel from Listing 1.2.

```cpp
// Adjoint user kernel
void stencil_adjoint(
    const ACC<double> &u, ACC<double> &u_a1s,
    const ACC<double> &f, ACC<double> &f_a1s,
    ACC<double> &u2, ACC<double> &u2_a1s) {
  double div = (2.0 * (dx * dx + dy * dy));
  u_a1s(-1, 0) += u2_a1s(0, 0) * dy * dy / div;
  u_a1s(1, 0) += u2_a1s(0, 0) * dy * dy / div;
  u_a1s(0, -1) += u2_a1s(0, 0) * dx * dx / div;
  u_a1s(0, 1) += u2_a1s(0, 0) * dx * dx / div;
  f_a1s(0, 0) += u2_a1s(0, 0) * dx * dx * dy * dy / div;
  u2_a1s(0, 0) = 0;
}
```

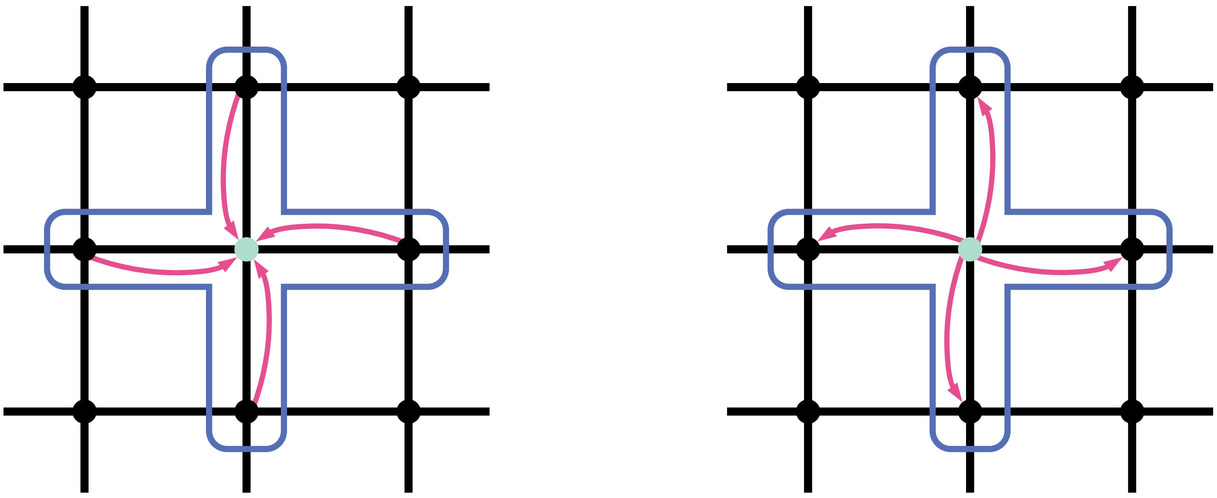


Figure 4.1: **Left:** The gather stencil used during the forward pass in the parallel loop in Listing 1.2. Data is read on the four neighboring points and in the center and only writes at the center. **Right:** The stencil and access pattern used during the reverse pass on the adjoint memory with the loop body in Listing 4.1. The data read on the five points, but writes happen for the neighbors.

in allowing OPS to perform complex optimizations like loop tiling and lazy execution[11] and also plays a crucial role in adopting reverse mode AD in OPS.

Since OPS has all the required information on the data dependencies on a loop level, we can build a DAG on a computational loop level instead of on an expression level, resulting in a smaller memory footprint. OPS will handle the execution of the adjoint loops by caching intermediate states during the primal evaluation and re-loading these states in the adjoint loops in reverse order.

The only component required to do adjoint mode AD is the adjoint functions of the kernel bodies. OPS already generates the primal kernel from the kernel function and the data presented by the stencils. If OPS has a reference to the adjoint of the kernel body (such as in Listing 4.1) function, it can use the same information to generate parallel loops for the adjoint of the kernel.

In an OPS loop, each dataset is accessed through a stencil with an assigned access pattern: read, write, or increment. All forward loops in OPS must use gather stencils only, meaning that read can appear with any stencil with any number of points, but increment and write stencils

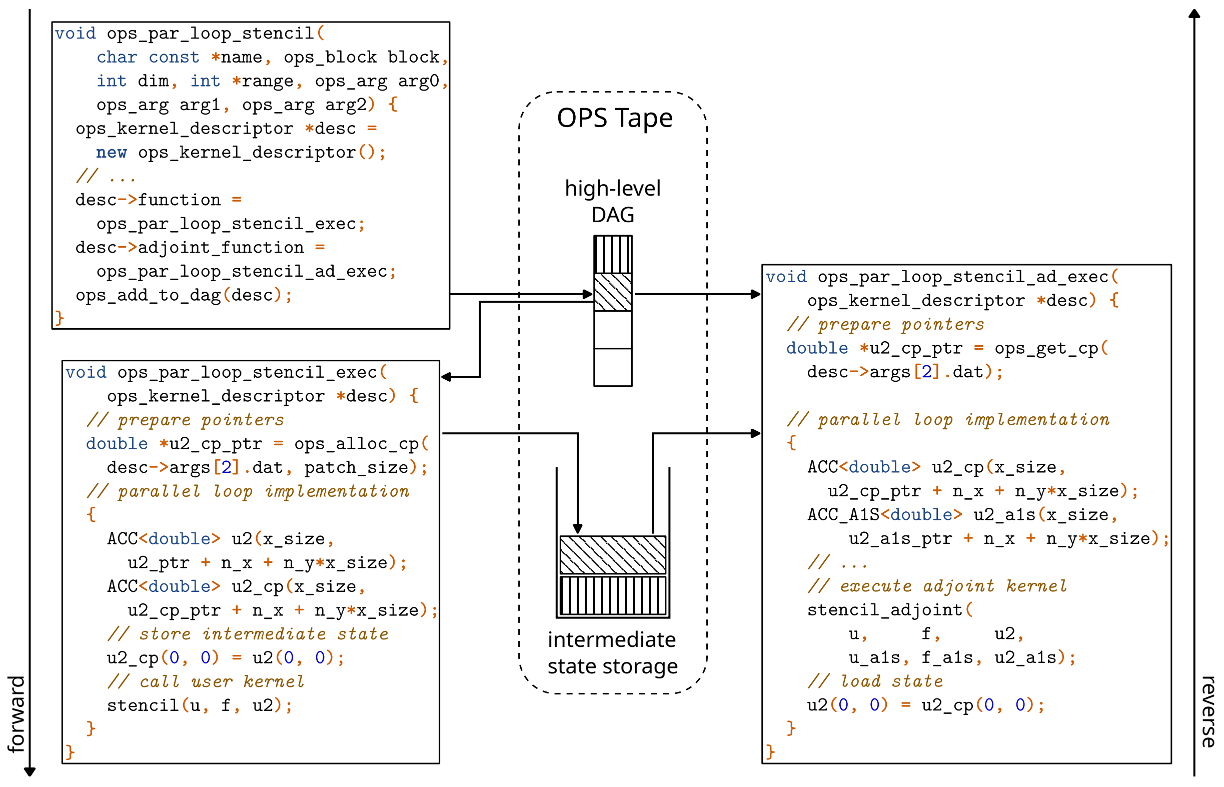
Listing 4.2 Example parallel loop using active grid invariant scalar data.

```
// User kernel
void kernel(const ACC<double> &a, const double* scalar,
            ACC<double> &anext) {
  anext(0, 0) += *scalar * (a(1, 0) - a(0, 0) + a(-1, 0));
}
// Adjoint User kernel
void kernel_adjoint(
    const ACC<double> &a, ACC_A1S<double> &a_a1s,
    const double* scalar,         double* scalar_a1s,
    ACC<double>   &anext, ACC_A1S<double> &anext_a1s) {
  a_a1s( 1, 0) +=      *scalar * anext_a1s(0, 0);
  a_a1s( 0, 0) += -1 * *scalar * anext_a1s(0, 0);
  a_a1s(-1, 0) +=      *scalar * anext_a1s(0, 0);
  *scalar_a1s +=
        (a(1, 0) - a(0, 0) + a(-1, 0)) * anext_a1s(0, 0);
}
// ...
// Declaring a dataset on a block
double scalar = 1.0;
ops_scalar scl = ops_decl_scalar("scl", &scalar, 1);
// ...
// Execute a given loop on the block
ops_par_loop(kernel, "kernel", block, 2, iter_range,
   ops_arg_dat(a,      1, S2D_3PT, "double", OPS_READ),
   ops_arg_scalar(scl, 1,          "double", OPS_READ),
   ops_arg_dat(anext,  1, S2D_00,  "double", OPS_INC));
```

must have one single point access with zero offsets. In Reverse mode AD in the adjoint loops, the data flow will be reversed from gathering stencils, and we will get scatter operations on the adjoint data. Due to the reversal, the write and increment stencils on the datasets will turn into one-point read stencils on the adjoint data, and read stencils will turn into increment stencils with multiple points. Figure 4.1 shows the change in the access pattern between the primal and adjoint loop for a five-point stencil. The above has two implications: first, the primal loops are race-free, the parallelization is trivial, and second, the adjoint loops will have data races on the adjoint data. However, OPS has all the information on where these data races occur from the read stencils of the primal loops.

OPS has two other loop parameter types: grid invariant constants and global reductions. I keep the global constants as passive data, which will not have derivatives, and introduce a new `ops_scalar` type for global read parameters with adjoint data. Reads on `ops_scalars` will turn into global reductions in the adjoint loops. Finally, global reductions become global reads on the adjoint data in the adjoint loops.

Listing 4.2 shows a kernel with an active scalar value. To use `ops_scalars` in the kernel, it must be passed with a `ops_arg_scalar` wrapper with `OPS_READ` access. This argument will notify OPS to generate the proper code for the adjoints as well. Note that the adjoint loop has a global reduction in the derivative of the scalar, which will naturally have a significant effect on the performance during the execution of the adjoint loop. On the other hand, the opposite happens for reductions in primal loops, which can be removed entirely from adjoint loops and produce a global read on the adjoint memory.



```
void ops_par_loop_stencil(
    char const *name, ops_block block,
    int dim, int *range, ops_arg arg0,
    ops_arg arg1, ops_arg arg2) {
  ops_kernel_descriptor *desc =
    new ops_kernel_descriptor();
  // ...
  desc->function =
    ops_par_loop_stencil_exec;
  desc->adjoint_function =
    ops_par_loop_stencil_ad_exec;
  ops_add_to_dag(desc);
}
```

```
void ops_par_loop_stencil_exec(
    ops_kernel_descriptor *desc) {
  // prepare pointers
  double *u2_cp_ptr = ops_alloc_cp(
    desc->args[2].dat, patch_size);
  // parallel loop implementation
  {
    ACC<double> u2(x_size,
      u2_ptr + n_x + n_y*x_size);
    ACC<double> u2_cp(x_size,
      u2_cp_ptr + n_x + n_y*x_size);
    // store intermediate state
    u2_cp(0, 0) = u2(0, 0);
    // call user kernel
    stencil(u, f, u2);
  }
}
```

```
void ops_par_loop_stencil_ad_exec(
    ops_kernel_descriptor *desc) {
  // prepare pointers
  double *u2_cp_ptr = ops_get_cp(
    desc->args[2].dat);

  // parallel loop implementation
  {
    ACC<double> u2_cp(x_size,
      u2_cp_ptr + n_x + n_y*x_size);
    ACC_A1S<double> u2_a1s(x_size,
        u2_a1s_ptr + n_x + n_y*x_size);
    // ...
    // execute adjoint kernel
    stencil_adjoint(
        u,     f,     u2,
        u_a1s, f_a1s, u2_a1s);
    // load state
    u2(0, 0) = u2_cp(0, 0);
  }
}
```

Figure 4.2: Interactions of forward and corresponding reverse loops with the high-level tape in OPS. Each active forward loop pushes a small descriptor into the DAG, and during execution, it will save the overwritten data to a stack-like storage. In the reverse pass, OPS will call the generated adjoint for the loop, which will propagate the derivative information and load the saved state.

4.1.1 Adjoint function of the loop body

OPS handles the code generation of the adjoint loops similarly to the primal, which requires a function pointer to the loop body. OPS depends on the function provided by the user as an adjoint of a loop body with the `_adjoint` suffix. To help provide these functions, I implemented a helper script to generate adjoints of user kernels using Tapenade[106], which can help to generate adjoints for relatively simple loop bodies and can give a good starting point to write the adjoints for more complex functions. This tool acts as a wrapper for calling Tapenade on the user kernels, mapping the OPS API to a format for which Tapenade can generate adjoints. The tool treats `ops_dat` and `ops_scalar` parameters as active at the moment. After generating the adjoint functions, OPS uses them as the user-provided adjoints for the loops to generate the backend-specific parallel loops. In our experiments, I used this tool to generate the adjoint implementations for the 87 distinct kernels in Cloverleaf - a few required further modifications (such as removing stack objects that would not work for CUDA) to these implementations to get our final versions of the adjoint kernels.

4.1.2 Getting derivatives in OPS

Listing 4.3 shows an example AD workflow in OPS. The code computes the adjoints with respect to a given output in four main steps in OPS. Initialize datasets and grid invariant

Listing 4.3 Typical steps for an Adjoint workflow in OPS.

```
// OPS initialisation
std::unique_ptr<OPS_instance> instance =
  std::make_unique<OPS_instance>(argc, argv, 1);
// Mesh
ops_block block = instance->decl_block(2, "The Block");
// Preparing data for ops
ops_scalar scl = ops_decl_scalar("scl", &scalar, 1);
ops_dat a_in = block->decl_dat(1, size, base, pad_p, pad_m,
                             input_a, "double", "a_in");
ops_dat a  = block->decl_dat(1, size, base, pad_p, pad_m,
                             nullptr, "double", "a");
ops_dat a2 = block->decl_dat(1, size, base, pad_p, pad_m,
                             nullptr, "double", "a2");
// ...
// Run the code to be differentiated
ops_par_loop(copy, "copy", block, 2, iter_range,
   ops_arg_dat(a_in,   1, S2D_00, "double", OPS_READ),
   ops_arg_dat(a,      1, S2D_00, "double", OPS_WRITE));
for (int i = 0; i < niter; ++i) {
  ops_par_loop(kernel, "kernel", block, 2, iter_range,
     ops_arg_dat(a,      1, S2D_3PT, "double", OPS_READ),
     ops_arg_scalar(scl, 1,          "double", OPS_READ),
     ops_arg_dat(a2,     1, S2D_00,  "double", OPS_INC));

  std::swap(a2, a);
}

// Initialise Adjoints
int memspace = OPS_HOST;
auto *a_a1s = reinterpret_cast<double *>(
    a->derivative_get_raw_pointer(S2D_00, &memspace));
// Seed the output
a_a1s[output_idx] = 1.;
a->release_raw_derivative(OPS_WRITE, memspace);
// Interpret Adjoints
ops_interpret_adjoints(instance.get());

// Accessing derivatives
int disp[OPS_MAX_DIM];
int size[OPS_MAX_DIM];
ops_dat_get_extents(a_in, 0, disp, size);
size_t array_size = size[0] * size[1];
std::vector<double> output(array_size);
a_in->fetch_derivative(output.data(), OPS_HOST);

```

values (ops_scalars) before the primal, followed by actually computing the primal the same way as a traditional OPS application. Then, seed the output variable's derivative and finally run the reverse pass. During the primal execution, OPS will save all events (loops, reductions, etc.) relevant for derivative propagation into its own tape and save the intermediate states if necessary. OPS needs to store only a small descriptor of the parallel loop in memory to create a representation of the computation. Based on this descriptor, OPS will execute the loop and store the overwritten data. Figure 4.2 shows the actions performed on a parallel loop and its corresponding section in the reverse pass. After the primal execution, the adjoint of the output needs to be initialized, and finally, the `ops_interpret_adjoints` call will run the adjoint of each computational step in reverse order, accumulating derivative information. This final step will allocate and initialize all required adjoint variables that are not seeded before the function call. For each event in the DAG, OPS will execute the corresponding adjoint function, such as calling the adjoint loop as shown in Figure 4.2. Listing 4.3 also shows the two main ways to access derivatives of datasets. The API is similar to the data access API in OPS. The user can either get access to the raw pointer underneath the `ops_dat` variable or can create a copy in some memory space.

In order to maximize performance and constrain memory use, it is helpful to identify which variables are taking part in the propagation of derivative information. Existing tools use different conventions - for example, most operator overloading tools embed this information into their types. Some source transformation tools treat all pointers to floating point values as active, and all variables copied by value as passive constants or depend on an explicit listing of active input and outputs. In OPS, all datasets declared with `ops_decl_dat` are considered active, the user can define passive datasets with the `ops_decl_dat_passive` function to avoid the allocation of the adjoint variables, and similarly, the user can create passive loops with `ops_par_loop_passive` function for loops that are not part of the derivative propagation. For active loops, OPS can detect active variables from the signature of the adjoint function of the loop body. The example kernel in Listing 4.2 shows a loop where all inputs are active, and their derivative is present in the argument list of the adjoint. By simply leaving out the derivative of an input from the signature of the adjoint of a given kernel function, OPS will treat the relevant data in that kernel as passive.

4.1.3 Computations outside of OPS

In some cases, pure stencil computations cannot give sufficient performance, or methods that cannot be represented in the abstraction of OPS are required. An important case here is direct linear solvers, which are used to implement High-Order Compact discretization schemes or modern direction splitting time steppers such as Douglas, Modified Craig-Sneyd, or Hundsdorfer-Verwer [151]. The standard solution in a non-AD context is to ask for access to the raw data behind the OPS datasets, perform the computations outside of OPS, and return the data to OPS. I refer to such calculations which take place outside of the OPS abstraction as *external calculations*. OPS provides an external adjoint API to support this feature in an AD-aware manner. OPS takes a function pointer to the primal of the external computations, a function to the adjoint of the external function, and an explicit list of datasets accessed in the external function, similar to an

OPS loop. Inside this function, the user can execute arbitrary computations with the traditional APIs and even OPS loops as long as all data ownership is returned to OPS before the end of the function. It is crucial that the function not modify any global program state not owned by OPS since this could result in incorrect values should checkpointing be used. For such cases, OPS provides `ops_execute_external_function` version which always saves the affected OPS data to avoid the re-execution of external functions at the cost of the additional memory overhead. Listing 4.4 shows an external function from the CDE application accessing four datasets that form a batch of tridiagonal systems, solve the systems, and return ownership to OPS. The `ad` function contains the adjoint of the external function, which consists of a tridiagonal solve of the transposed system and an OPS loop on the derivatives.

4.1.4 Controlling memory requirements

Adjoint mode AD executes the adjoint of each instruction of the primal in reverse order, for which it requires the intermediate states of the variables. There are two ways to produce these intermediate states that are combined in most tools: saving the states during the primal and reloading from memory and recomputing. OPS, by default, saves all data that a loop writes at the beginning of every loop. During the adjoints, OPS will load these, restoring all variables to the state before the primal loop. With this strategy, OPS does not need to rerun primal loops, which means faster overall run-time, but this introduces significant memory requirements to checkpoint all these intermediate states. Existing tools use checkpointing strategies to manage the trade-off between extra computational steps due to recomputing and the high memory requirement of adjoint mode AD. OPS implements Revolve [150], originally developed for time marching schemes.

OPS enables users to define checkpoints, manually specifying which datasets to save. OPS will store the information of all possible checkpoints in the DAG and use Revolve to manage rerunning loops and saving intermediate states. Listing 4.5 shows the Revolve API for OPS. At the beginning of the application, OPS requires the user to set the parameters for Revolve with `ops_ad_set_checkpoint_count`, and then Revolve decides which checkpoints should be active in the chain. At the beginning of each application-level iteration, the user provides a list of datasets to build a checkpoint with `ops_ad_manual_checkpoint_datlist`. The list should contain datasets that are required to reset their states to this point to compute the current iteration, or in other words, all datasets that are read and will be overwritten later in the application. Every time OPS encounters a `ops_ad_manual_checkpoint_datlist` call, it will save the list of datasets provided, and if this checkpoint should be active, then saves a copy of each dataset to memory and compute the index of the next active checkpoint. In the reverse pass, OPS will interpret the adjoint of one iteration at a time. For each iteration, it will recompute iterations from the last active checkpoint and cache the intermediate states for the last iteration to prepare it for the reverse pass. If OPS passes an active checkpoint, OPS frees the saved data, and when OPS recomputes sections of the DAG and has available slots for checkpoints, OPS will save checkpoints based on the Revolve algorithm.

Listing 4.4 An external function solving a batch tridiagonal system formed by the datasets. The `primal` function will be called by the `ops_execute_external_function_recomp` function during the forward pass, while the `ad` function provides the adjoint for the external function.

```
auto primal = [=]() {
  // get access to used ops_dats
  double *a_raw = reinterpret_cast<double *>(
      ops_dat_get_raw_pointer(a, 0, S2D_00, &mem_space));
  // ...
  tridDmtsvStridedBatch(&trid_params, a_raw, b_raw, c_raw, d_raw,
                        ndim, solvedim, dims, stride);
  ops_dat_release_raw_data(a, 0, OPS_READ); // ...
};
auto ad = [=]() {
  // get access to used ops_dats
  double *a_raw = reinterpret_cast<double *>(
      ops_dat_get_raw_pointer(a, 0, S2D_00, &mem_space));
  double *d_a1s = reinterpret_cast<T *>(
        ops_dat_derivative_get_raw_pointer(d, S2D_00, &mem_space));
  // ...
  tridDmtsvStridedBatch(&trid_params, c_raw - 1, b_raw, a_raw + 1, d_a1s,
                        ndim, solvedim, dims, stride);
  // Release datasets
  ops_dat_release_raw_derivative(d, OPS_RW, mem_space);
  ops_dat_release_raw_data(a, 0, OPS_READ);
  // ...
  // Update a1_M =  -a1_d * x^T
  ops_dat a1_a = ops_get_derivative_as_ops_dat(a); // ...
  ops_par_loop_passive(update_a1_M, "update_a1_M", theBlock, 2, range,
                       ops_arg_idx(),
                       ops_arg_gbl(&nx, 1, "int", OPS_READ),
                       ops_arg_dat(d, 1, S2D_3PT_X, "double", OPS_READ),
                       ops_arg_dat(a1_d, 1, S2D_00, "double", OPS_READ),
                       ops_arg_dat(a1_a, 1, S2D_00, "double", OPS_INC),
                       ops_arg_dat(a1_b, 1, S2D_00, "double", OPS_INC),
                       ops_arg_dat(a1_c, 1, S2D_00, "double", OPS_INC));
};
ops_execute_external_function_recomp(primal, ad,
     ops_arg_dat(a, 1, S2D_00, "double", OPS_READ),
     ops_arg_dat(b, 1, S2D_00, "double", OPS_READ),
     ops_arg_dat(c, 1, S2D_00, "double", OPS_READ),
     ops_arg_dat(d, 1, S2D_00, "double", OPS_RW));
```

Listing 4.5 Example use of Revolve for the loop chain from Listing 4.3 with 10 checkpoints and a fix chain length of `niter`. The `ops_ad_set_checkpoint_count` call initiates Revolve, and the `ops_ad_manual_checkpoint_datlist` function marks the checkpoints from which OPS can start recomputing steps. The function takes all the `ops_dats` that are required to recompute the iteration.

```
// Initialise Revolve
ops_ad_set_checkpoint_count(instance.get(), 10, niter);
// Register a Revolve checkpoint
ops_ad_manual_checkpoint_datlist(a_in);
// Run the code to be differentiated
ops_par_loop(copy, "copy", block, 2, iter_range,
   ops_arg_dat(a_in,   1, S2D_00, "double", OPS_READ),
   ops_arg_dat(a,      1, S2D_00, "double", OPS_WRITE));
for (int i = 0; i < niter; ++i) {
  // Register a Revolve checkpoint
  ops_ad_manual_checkpoint_datlist(a, a2);
  ops_par_loop(kernel, "kernel", block, 2, iter_range,
     ops_arg_dat(a,      1, S2D_3PT, "double", OPS_READ),
     ops_arg_scalar(scl, 1,          "double", OPS_READ),
     ops_arg_dat(a2,     1, S2D_00,  "double", OPS_INC));

  ops_par_loop(copy, "copy", block, 2, iter_range,
     ops_arg_dat(a2,     1, S2D_00, "double", OPS_READ),
     ops_arg_dat(a,      1, S2D_00, "double", OPS_WRITE));
}
```

4.1.5 Retaping and reverse accumulation

OPS provides access to its tape object to give a finer level of control if needed. Through the tape object, OPS provides support for retaping, where the same tape is used multiple times to compute different rows in the Jacobian. OPS allows to create and compose a hierarchy of tapes that allows advanced techniques like reverse accumulation for fixed point iterations [152] that can drastically reduce the size of the tape and cached states.

Listing 4.6 shows an example of reverse accumulation. OPS can create a local tape in the adjoint, record computations, and reuse the computational graph to form a convergent iteration from the adjoint. Since all iterations in the primal are performed in the external function (or performed as passive loops), these loops are not present in the original tape, and OPS will not save any intermediate state for them. Instead, in the adjoint of the external function, the new tape will record one iteration and its intermediate states. However, OPS must reset the adjoint and primal states between each iteration, which involves runtime overheads.

4.2 Orchestration

As discussed in Section 4.1, all primal computational loops in OPS will write data at the center of the stencil or perform a reduction on global variables. OPS will save the state of the reductions for all loops in order to avoid recomputing reductions. For datasets, the parallel OPS loop already ensures that no race conditions exist for the writes; thus, creating copies of the states is

Listing 4.6 Example code for executing a fixed point iteration in the primal with a fixed point iteration as an adjoint. In the adjoint of an external function, the user can create local tapes to use. Similarly to other external functions, the user can use passive loops for error calculation, reseeding, or other purposes.

```
auto primal = [&]() {
  double error = std::numeric_limits<double>::max();
  while (error > tolerance) {
      ops_par_loop(step, "step", block, 2, range,
          // ...
          ops_arg_dat(a, 1, S_00, "double", OPS_INC),
          ops_arg_reduce(err, 1, "double", OPS_MAX));
      err->get_result(&error);
  }
};
auto ad = [&]() {
  ops_dat derivative = a->get_derivative_as_ops_dat();
  ops_reduction maxerr =
        ops_decl_reduction_handle(sizeof(double), "double", "maxerr");
  ops_ad_tape *tape = ops_create_tape(OPS_instance::getOPSInstance());
  // record one iteration
  ops_par_loop(step, "step", block, 2, range,
      // ...
      ops_arg_dat(a, 1, S_00, "double", OPS_INC),
      ops_arg_reduce(tmperr, 1, "double", OPS_MAX));
  double error = std::numeric_limits<double>::max();
  while (error > tolerance) {
      // interpret adjoints
      tape->interpret_adjoint(OPS_instance::getOPSInstance());
      // check error
      ops_par_loop_passive(maxerr, "maxerr", block, 2, range,
          // ...
          ops_arg_dat(derivative, 1, S_00, "double", OPS_READ),
          ops_arg_reduce(maxerr, 1, "double", OPS_MAX));
      maxerr->get_result(&error);
      // zero adjoints in tape
      tape->zero_adjoints_except({a});
      // reseed others like scalars
      // ...
  }
  ops_remove_tape(OPS_instance::getOPSInstance(), tape);
};
ops_execute_external_function(primal, ad,
    // ...
    ops_arg_reduce(err, 1, "double", OPS_MAX),
    ops_arg_dat(a, 1, S_00, "double", OPS_INC));
```

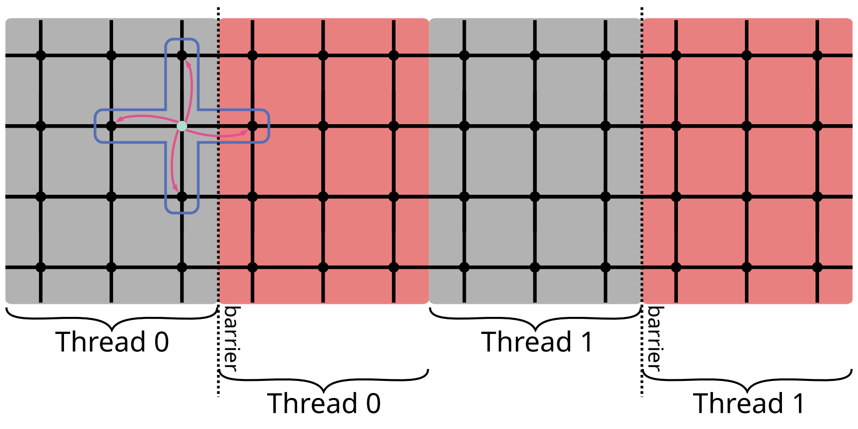


Figure 4.3: Example scheduling applied by OPS with OpenMP for a three-point wide stencil. Stripes with the same color are executed parallel, each thread executes a single chunk.

performed inside the loops before calling the user kernel. In the adjoint loops, the OPS will load these copies for each grid point at the beginning of the loop body and redirect any subsequent writes for these datasets to local storage.

The adjoint kernels will reverse the data flow, introducing data races on the adjoint data of read-only datasets. There are four main data access patterns to consider: reductions, global reads (`ops_scalar`), data access on grid points, and data access on lower dimensional data. The first two will switch sides in the reverse pass on the adjoint data, the adjoint of reductions will be read-only data, and the loop will perform reductions on the adjoint for the `ops_scalar` arguments. In the other two cases, the data races will arise along read stencils accessing multiple points or lower dimensional data. In the implementation, I chose to handle the data races from lower dimensional data separately, which I will discuss in Section 4.2.1.

To ensure correct results with data races present during the adjoints, a common approach is to use atomic operations on increments for data that is not thread-local. The other main possibility is to use the colored execution of grid points and run only points with the same color in parallel. The deciding factor between the two solutions is the cost of atomic operations and the cost of creating a coloring and running the loops in multiple chunks. From the stencils in the primal loops, OPS has information on the pattern in which data races could arise during the adjoint of loops and can generate adjoint loops in different configurations.

On CPUs, the cost of atomics is higher, and running the loop in two chunks with a barrier between the two chunks is relatively fast. Therefore, OPS uses the latter approach. Given the stencils' width in a parallel loop, OPS can divide the grid into stripes along the highest dimension. Figure 4.3 shows an example of the coloring. OPS chooses the stripes such that each stripe has a width equal to at least the width of the stencils. Then, the stripes are executed in two stages, and each stage computes at least the width of the stencil iterations, making both stages race-free (no overlaps of the written data within a stage).

On GPUs, the overhead of introducing atomic operations is much smaller compared to the performance loss of the worse memory access patterns that would be introduced by colored execution. Therefore, adjoint loops in CUDA use atomic operations on the adjoints of datasets with a single kernel launch. This keeps the coalesced memory accesses in the loops, but the performance of the writes on the adjoint data depends on the stencil itself.

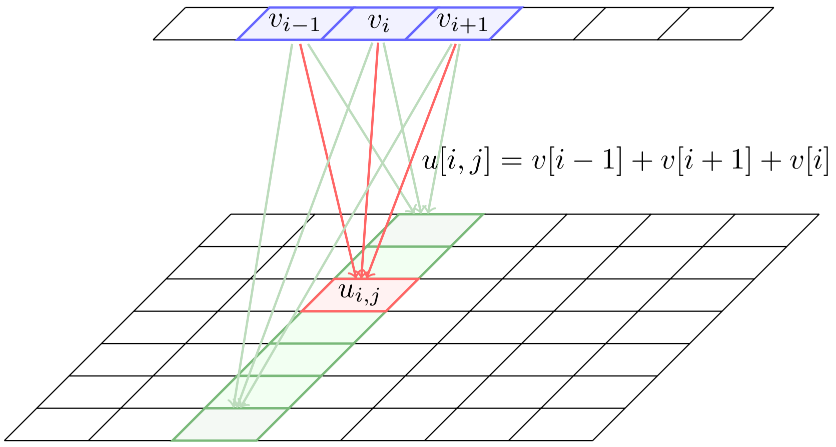


Figure 4.4: Data access for a three-point read stencil on a 1D dataset v in a 2D loop. The arrows represent the accessed values to compute the new u values. The loop reads the same three points for each iteration, sharing the same i index highlighted in green. In the adjoint loop, each iteration highlighted in green will increment the derivative for the same three values in v. In addition, the neighboring columns will have overlapping increments as well.

Listing 4.7 Example two dimensional loop with data access to lower dimensional data.

```
for (int j = 1; j < size_y - 1; ++j) {
  for (int i = 1; i < size_x - 1; ++i) {
    u[j * size_x  + i] = x[i - 1] + x[i] + x[i + 1] + v[j - 1] + v[j] + v[j + 1];
  }
}
```

4.2.1 Handling low-dimension datasets

As mentioned above, we handle access to lower dimensional datasets separately to mitigate the effect of the shared data access of large amounts of threads. Reads in the primal to a low-dimension dataset will result in a large number of writes in the adjoint loop to the adjoint of the low-dimension dataset. Essentially, the loops perform sums along the dimensions where the lower dimensional dataset is invariant. For example, in a two-dimensional loop such as in Listing 4.7, the loop would access two one-dimensional datasets x and v, where x is invariant along the Y axis, and v is invariant along the X axis with access patterns shown in Figure 4.4.

The loop is parallelized on CPUs in the outermost (non-contiguous) dimension. If a dataset is not invariant in this dimension, such as v in the example, then the original parallelization already takes care of the races on the dataset. However, in the case of x, where the dataset is invariant in the outermost dimension, all threads would write the same values of the adjoint, which would lead to further data races. To avoid a data race for these datasets, OPS will allocate a temporary buffer for each thread to collect the increments from the given thread during the adjoint loop. Then, in a separate step, OPS will sum the increments from each thread for the adjoint of the lower dimensional datasets.

On the GPU, atomic operations are sufficient for handling the data races in theory. However, in the presence of lower dimensional datasets, the performance of the atomic operations drastically drops due to the high number of conflicts. If possible, OPS will use two-dimensional CUDA

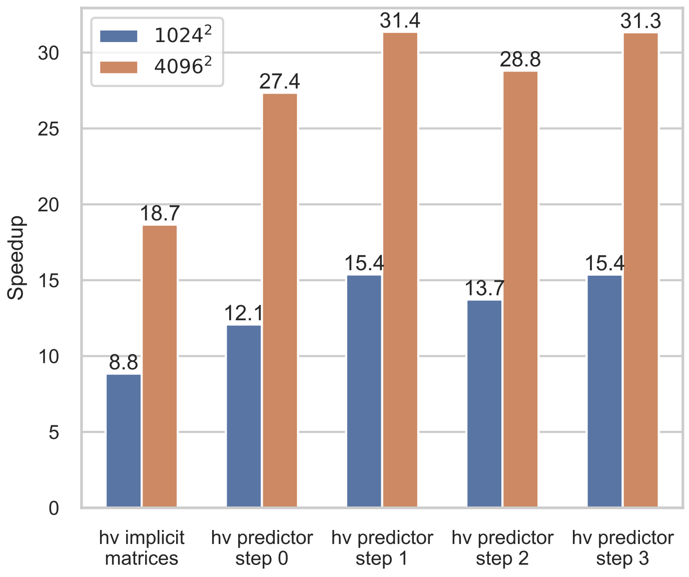


Figure 4.5: Speedup of our access-pattern-aware adjoint reductions and accumulations, compared to using only atomics on kernels from the CDE application, which have lower-dimensional data accesses with a mesh size of 1024^2 and 4096^2 on an NVidia A100.

thread blocks with the shape of $32 \times n_{warps}$ to reduce the effect of writes to the same location within an OPS block. This shape keeps the coalesced memory accesses of the primal on the higher dimensional datasets. However, threads within a warp will write the same values for $v[j]$ and $v[j \pm 1]$ from Listing 4.7 (a dataset that is invariant in X), so when the warp executes an atomic write, the memory address collisions within the warp will result in the write being serialized. In the case of x (a dataset that is invariant Y), threads within a warp will only have address collisions if non-zero stencils are present, but address collisions arise among the warps inside the block. To reduce the effects of these data races, OPS will accumulate the increments of each thread in registers and then handle races in a separate step at the end of the kernel. There are four important cases that OPS handles. First, for datasets like v that are always shared by the warp, OPS will perform a warp level reduction so that only one thread per warp will perform an atomic write to the global memory. Second, for datasets like x that are always shared among the warps of the block, the increments are written with atomics directly by each thread at the end since the threads in a single warp do not have races in the center of the stencil. The final two cases arise in three- or higher-dimensional loops. If a dataset is invariant in both the X and Y dimensions, all threads within a block will share all accessed data, and OPS will perform a block-level reduction for all written adjoint values. Finally, datasets that are invariant only in the third or higher dimensions will behave like ordinary datasets from the perspective of the block.

Figure 4.5 shows the actual speedup this optimization achieved on the adjoints of kernels with lower dimensional datasets in one of the test applications. These kernels use the access pattern shown in Listing 4.7 on two separate datasets. The CDE application is a 2D stencil code on a non-uniform mesh[153]. Computing the finite difference estimates requires knowing the mesh coordinates to the left and right in the X dimension and to the top and bottom in the Y dimension, in addition to knowing the current mesh point. These X and Y mesh coordinates are stored in two 1D datasets. Warps have coalesced reads to the X mesh coordinates similarly to x

in Listing 4.7, whereas each read from the Y mesh coordinates results in a single value broadcast across the warp as for v. The three-point access to the two datasets can be replaced with access to 12 separate datasets per mesh point but with a one-point stencil instead of three, which reduces the data races during the adjoint kernel. This optimization increases the performance for both the primal and adjoint kernels. In the presence of one-point stencils only, the reductions achieve 3.7 – 6.9x and 7.1 – 15.4x speedups on the 1024^2 and 4096^2 mesh, respectively.

4.3 Evaluation

To show the performance of AAD in OPS, I tested it on three distinct applications. I tested a Poisson mini application from the examples in OPS, the Cloverleaf[154] CFD application, and a 2D parabolic Convection-Diffusion Equation (CDE) solver from computational finance[153]. In my measurements, *passive* denotes the time it takes to run the original application, *forward* denotes the time it takes to run the application and record the tape, and *adjoint* denotes the time of computing derivatives for all inputs.

The Poisson application solves the equation

$$\nabla^2 u = f$$

with Jacobi iterations with 2D finite differences, where the update iteration can be written as:

$$u_{i,j}^{(n+1)} = \frac{(u_{i-1,j}^{(n)} + u_{i+1,j}^{(n)})dy^2 + (u_{i,j-1}^{(n)} + u_{i,j+1}^{(n)})dx^2 - dx^2 dy^2 f_{i,j}}{2(dx^2 + dy^2)}$$

I used a 4096^2 mesh with 100 iterations for evaluating performance. These measurements show the performance of recording the tape and accumulating derivative information with OPS. I introduced two versions for handling the derivative propagation; the first is to record all iterations and compute the adjoints, and the second is to take advantage of the fact that the code can be expressed as a fixed point iteration. Hence, I can compute the derivative with a fixed point iteration with the tape for a single iteration.

CloverLeaf[154] is a mini-app that solves the compressible Euler equations on a Cartesian grid using an explicit, second-order accurate method. Cloverleaf is a mini-app from the UK Mini App Consortium and is widely used for performance benchmarks for stencil computations. I used the OPS implementation for Cloverleaf on two example problems, one with a 960^2 problem size and 2955 time steps and a bigger 3840^2 problem with 87 time steps.

The CDE application is a convection-diffusion-reaction code from computational finance[153]. It computes the price of a European call option driven by a Heston stochastic volatility model using the Method of Lines. The CDE application uses a finite volume scheme with central finite differences and first-order upwinding at the $v = 0$ boundary. It uses the Hundsdorfer–Verwer ADI time stepping scheme and requires the solution of batches of tridiagonal systems, which the tridsolver[J2] library performs. I use this application to highlight two aspects: the effect of low-dimensional datasets on performance and external adjoint nodes in OPS.

4.3.1 Measurement Setup

I ran the OpenMP measurements on CentOS Linux release 7.9 using a single socket of an Intel(R) Xeon(R) Gold 6226R CPU at 2.9 GHz without hyper-threading and 376 GB RAM. Using the Empirical Roofline Toolkit[155] OpenMP kernels reach 224.5 GFLOP/s on a single sockets without hyper-threading. The CUDA measurements were executed on a CentOS 8 using an AMD EPYC 7F72 24-core Processor and an NVidia A100 GPU with 40 GB RAM with a 5.25 TFLOP/s peak double precision compute throughput. All codes are compiled using GCC 11.2 and CUDA 12.0. I do not use Revolve for the Poisson application but save all intermediate states for the 100 iterations. When I do not specify otherwise, I report runtimes of Cloverleaf 985 using 29 Revolve checkpoints for both the 960^2 and 3840^2 test-cases, and 10 checkpoints for CDE. All measurements were repeated ten times, and the mean value is shown as the final result. For each application, I measured the original application and the AD version with the combined forward and reverse pass.

4.3.2 Results

Intel(R) Xeon(R) Gold 6226R 16 threads using OpenMP	Revolve CPs	Runtime Passive (s)	Peak memory passive (GiB)	Runtime Adjoint (s)	Peak memory adjoint (GiB)	Adjoint factor	Number of sensitivities
Poisson 4096^2, 100 iter	-	0.677	0.50	2.974	14.88	4.39	34 M
Poisson Iterative 4096^2, 100 iter	-	0.677	0.50	2.514	4.63	3.72	34 M
Cloverleaf 960^2, 2955 iter	895	60.076	0.18	450.526	29.01	7.03	4 M
Cloverleaf 960^2, 2955 iter	100	60.076	0.18	470.058	3.96	7.82	4 M
Cloverleaf 3840^2, 87 iter	29	32.791	2.76	202.890	28.46	6.19	59 M
Cloverleaf 3840^2, 87 iter	10	32.791	2.76	218.522	18.43	6.66	59 M
CDE 1024^2, 100 iter	10	1.399	0.19	13.521	1.07	9.67	1 M
CDE 4096^2, 100 iter	10	34.259	3.01	261.440	12.52	7.63	17 M
NVidia A100							
Poisson 4096^2, 100 iter	-	0.032	0.50	0.195	14.88	6.05	34 M
Poisson Iterative 4096^2, 100 iter	-	0.032	0.50	0.171	4.63	5.31	34 M
Cloverleaf 960^2, 2955 iter	895	7.639	0.18	40.631	29.01	5.32	4 M
Cloverleaf 960^2, 2955 iter	100	7.639	0.18	42.825	3.96	5.61	4 M
Cloverleaf 3840^2, 87 iter	29	1.799	2.76	11.885	28.46	6.61	59 M
Cloverleaf 3840^2, 87 iter	10	1.799	2.76	12.722	18.43	7.07	59 M
CDE 1024^2, 100 iter	10	0.453	0.19	2.078	1.07	4.59	1 M
CDE 4096^2, 100 iter	10	3.731	3.01	20.157	12.52	5.40	17 M

Table 4.1: Runtime and memory comparison between the original applications and the computing of the adjoints measured on an Intel(R) Xeon(R) Gold 6226R with 16 threads using OpenMP (Top) and on an NVidia A100 using CUDA (Bottom). **Note:** For the Poisson code, Iterative stands for computing the adjoints as a fixed point iteration with storing tape for the last iteration only.

For all applications, I evaluated the performance of the original OPS application as well as OPS with adjoint calculations. I evaluated each application using the OpenMP and CUDA backend and code generation path of OPS from the same source. As with many stencil applications, the Poisson and Cloverleaf applications are memory bandwidth-bounded codes, where the achieved bandwidth is a commonly used metric for performance on a given hardware. The CDE application uses lower dimensional datasets in most computational kernels, which leads to compute-bound loops for most of the loops in the application.

Table 4.1 shows all the absolute runtimes and peak memory usage of all the measured codes as well as the number of sensitivities computed by the evaluation of the adjoint model.

The simplest application is Poisson, which performs the computations from Listing 4.3, where a single computational loop applies the stencil from Listing 1.2 in each iteration. The first version applies the adjoint calculation directly, caching all intermediate states of the datasets for the

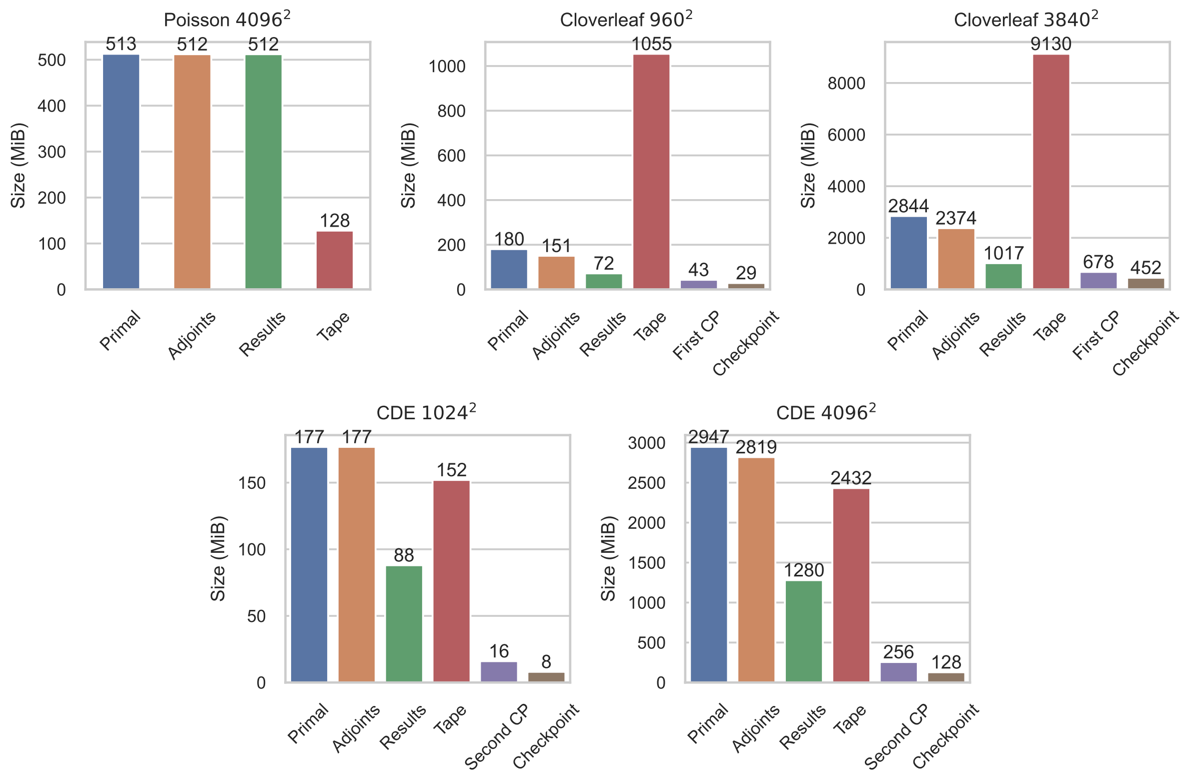


Figure 4.6: Memory requirements for the applications. *Primal* is the memory usage of the original application, while *Adjoint* shows the adjoint memory allocated for datasets. The checkpoint sizes note the size of the Revolve checkpoints. *First CP* for Cloverleaf and *Second CP* for CDE marks a single checkpoint with more data due to datasets that are only written at the beginning. All other checkpoints use the uniform size shown as *Checkpoint*. *Tape* marks the required saved states for interpreting one time step. Finally, *Results* marks extra copies for saving the datasets at the end of the forward pass.

whole application during the forward pass. The *Iterative* version performs the 100 iterations without caching any state in an external adjoint function. It creates a new tape for the fixed point iteration during the adjoint evaluation, records a much smaller tape for a single iteration, and performs 100 adjoint evaluations on that tape. Figure 4.6 shows the components of the application's memory usage. The *Iterative* approach reduces the required memory footprint due to caching only one iteration (and a loop after the fixed-point iteration computing the error) and reusing that tape instead of using 100x of the memory storing the tape for 100 iterations. The trade-off is that this approach introduces overheads to the adjoint propagation corresponding to resetting the primal and adjoint memory between adjoint evaluation on the secondary tape. Nevertheless, these overheads have a lower impact on the total runtime than caching all intermediate states during the forward pass in our case.

In Cloverleaf, we have significantly more kernels per time iteration (over 150), which leads to much more cached memory, as Figure 4.6 shows. Creating a tape containing data for every time step with these sizes is infeasible. Therefore, our implementation for Cloverleaf has to utilize Revolve.

Finally, in addition to Revolve, the CDE application uses the external function API with

Application	Testcase	BW (GiB/s)	Peak (%)
Poisson	4096^2, 100 iter	55.54	92.28
Poisson Active	4096^2, 100 iter	54.82	91.03
Poisson Iterative	4096^2, 100 iter	57.07	94.77
CloverLeaf	960^2, 2955 iter	50.33	83.58
CloverLeaf Active	960^2, 2955 iter	46.02	76.43
CloverLeaf	3840^2, 87 iter	59.42	98.67
CloverLeaf Active	3840^2, 87 iter	58.22	96.68
BabelStream Triad	1.0 GiB	60.22	100

(a) Bandwidth values measured on a single socket of an Intel(R) Xeon(R) Gold 6226R without hyper-threading using OpenMP.

Application	Testcase	BW (GiB/s)	Peak (%)
Poisson	4096^2, 100 iter	1195.93	92.87
Poisson Active	4096^2, 100 iter	980.11	76.11
Poisson Iterative	4096^2, 100 iter	968.66	75.22
CloverLeaf	960^2, 2955 iter	600.02	46.60
CloverLeaf Active	960^2, 2955 iter	634.70	49.29
CloverLeaf	3840^2, 87 iter	1124.07	89.13
CloverLeaf Active	3840^2, 87 iter	1024.86	79.59
BabelStream Triad	1.0 GiB	1287.70	100

(b) Bandwidth values measured on the NVidia A100 GPU using CUDA.

Table 4.2: Bandwidth values for the Poisson and Cloverleaf applications. The Peak (%) values show the relative bandwidth compared to the Triad kernel in BabelStream.

	OpenMP, GFLOPS/s				CUDA, TFLOP/s			
	1024^2, 100 iter		4096^2, 100 iter		1024^2, 100 iter		4096^2, 100 iter	
	primal	adjoint	primal	adjoint	primal	adjoint	primal	adjoint
hv implicit matrices	40.90	41.14	41.29	47.13	1.71	1.39	1.80	1.34
hv predictor step 0	41.50	24.58	38.40	27.62	2.85	1.58	3.00	1.67
hv predictor step 1	36.95	36.90	37.01	41.92	2.69	1.95	3.01	1.80
hv predictor step 2	35.89	24.66	32.63	27.90	2.72	1.68	2.90	1.75
hv predictor step 3	36.99	35.95	37.01	41.31	2.69	1.95	3.01	1.80
Peak throughput	224.5				5.25			

Table 4.3: Double precision throughput of compute bound kernels in the CDE application. The first four columns were measured on a single socket of an Intel(R) Xeon(R) Gold 6226R with the peak throughput measured with the Empirical Roofline Toolkit[155]. The last four columns were measured on the NVidia A100 GPU.

tridiagonal solver calls. We used two problem sizes to test a 1024^2 and a 4096^2 mesh with 100 time steps in each case. In both cases, the tridiagonal solver calls dominate the forward pass on GPUs (around 90% of total time), while on CPUs, the 1024^2 problem spends 23% and the 4096^2 46% of the forward pass solving tridiagonal systems.

I used the Triad kernel from BabelStream [149] to measure the achievable bandwidth for our test platforms. This kernel achieved 60.22 GiB/s on a single NUMA node of an Intel(R) Xeon(R) Gold 6226R and 1287.70 GiB/s on the NVidia A100 GPU. Available memory bandwidth is the main performance bottleneck for the Poisson and Cloverleaf applications. Table 4.2a shows the achieved bandwidth for the whole application run on CPU. For all test cases, the Adjoint versions achieve slightly lower bandwidth values due to the heavy use of atomics, with the biggest difference being 4.3 GiB/s for the 960^2 mesh in CloverLeaf which results in achieving 76.4% of the bandwidth of BabelStream. The highest bandwidth measurement for adjoints reaches 96.7% of BabelStream. Using fixed point iterations to compute the derivatives for the Poisson code achieves comparable bandwidth to the original application and lower runtime due to lower overheads during the forward pass.

Table 4.2b shows similar results on the NVidia A100 GPU. The 960^2 mesh is too small to saturate the GPU and achieves worse bandwidth. Similarly to the OpenMP versions, the adjoint implementations achieve lower bandwidth in general, but in the case of CUDA, the impact of

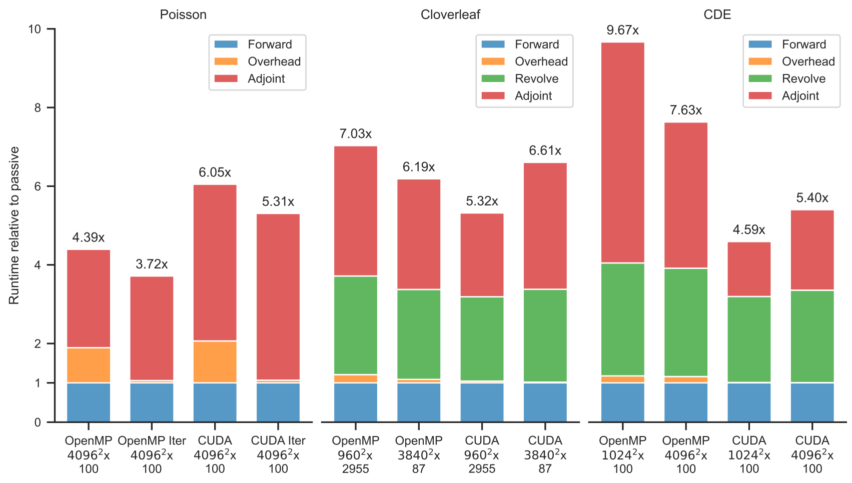


Figure 4.7: The overhead of AD on the benchmark applications, compared to a single evaluation of the passive, original application, showed by **Forward**. The values show that evaluating the adjoints takes N times of evaluating the original application. All bars show the values relative to the original applications. The **Overhead** values show the additional cost of collecting the DAG, saving intermediate states, and storing Revolve checkpoints during the forward pass, **Revolve** values represent the additional time spent on replaying sections of the applications to restore states for the adjoints loops and the **Adjoint** part shows the time spent in the actual adjoint loops.

atomics leads to a bigger difference than the passive versions.

Table 4.3 shows the achieved compute throughput for the compute-bound kernels from the CDE applications. The adjoint loops achieved similar compute throughput compared to the primal loops using OpenMP. However, on GPUs, the increased cost of the atomic leads to a larger difference in performance. As Table 4.3 shows, the adjoint loops achieve 55-80% of the throughput of the corresponding primal loops while reaching up to 37% of the theoretical double precision throughput of the A100.

Note that the runtimes in Table 4.1 and the bandwidth values were measured using the same high-level OPS source code without any modifications. By eliminating the cost of developing, debugging, and maintaining separate code bases for distinct parallel platforms, a domain-specific language like OPS enables the developer to focus on the application itself rather than the hardware-specific details and allows the application to run on the available hardware.

Figure 4.7 shows the relation between the runtime of the original application and the adjoint evaluation. The total values at the end of each bar show the relative cost of calculating the adjoints (including evaluating the original code). These values mean the time to evaluate the derivatives is equivalent to running the original code N times. The blue bars represent the original runtime, and the orange *Overhead* parts represent the additional time the active code spends in the forward pass. This time is spent on tasks like caching intermediate states, building the tape, and handling Revolve checkpoints. The green parts for Cloverleaf and CDE represent the time spent on re-running sections of the forward pass and handling moving the Revolve checkpoints, and finally, the red part marks the time spent on actually evaluating the adjoints.

In the case of the Poisson code, we can clearly see the effect of caching intermediate states. For all loops for each grid point, OPS will save the value of the written dataset, doubling the

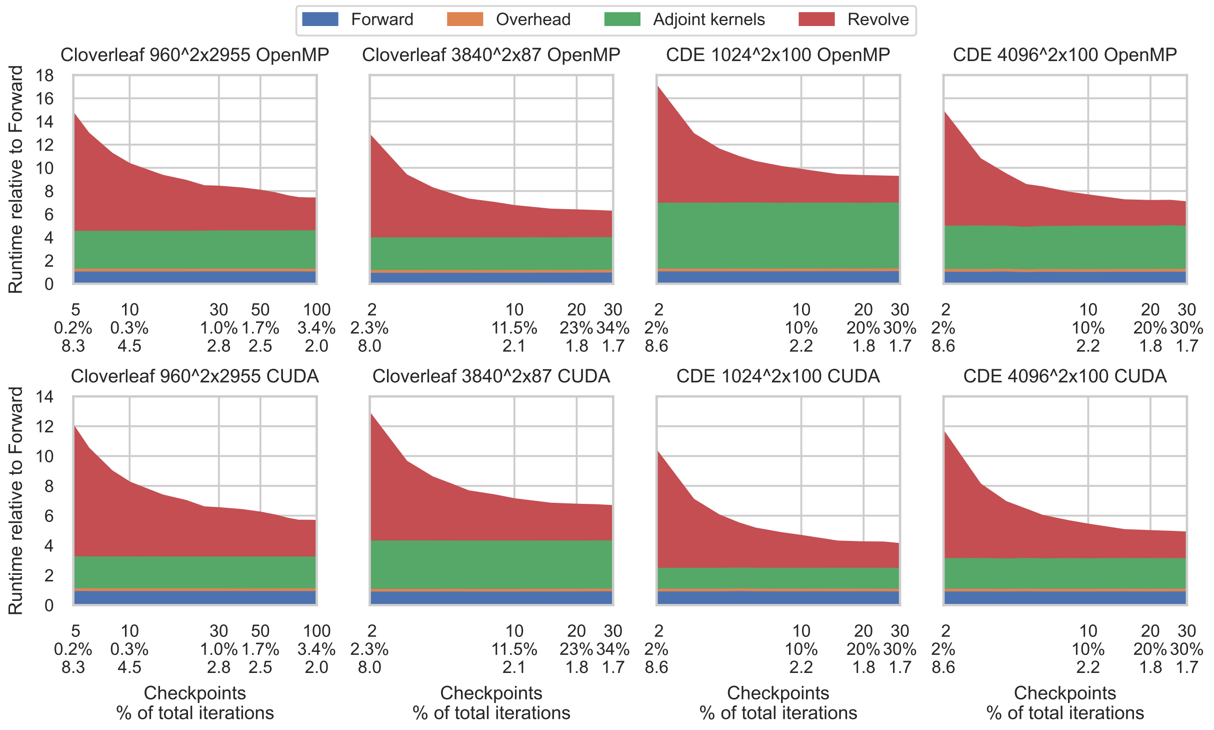


Figure 4.8: Runtime relative to a single evaluation of the original application (**Forward**) of evaluating the adjoints for the applications (including the original applications) with Revolve. On the X axis number of available Revolve checkpoints is increased. At each tick on the X three values are shown. The number Revolve checkpoints used, ratio of checkpoints and the total number of iterations, and the average number of forward steps taken at each iteration is shown. The increased number of checkpoints reduces the number of additional forward steps required to restore intermediate states.

write in each kernel, thus, the time required for the forward pass for OpenMP. The overhead is slightly bigger in the case of CUDA due to the higher cost of the memory operations related to the cached states. Nevertheless, the *Iterative* version clearly shows its advantage in reducing the overhead for the forward pass since it only requires caching before and after the fixed point iterations. Of course, the *Iterative* spends more time evaluating the adjoints as a consequence since it needs to reset the tape used for the fixed point iteration, increasing the cost of the adjoint evaluation from $2.5x$ to $2.7x$ and from $4.0x$ to $4.2x$ for OpenMP and CUDA respectively. These overheads are still compensated in the case of Poisson with the reduction in the *Overhead* cost, but this heavily depends on the application and the quality of the adjoint kernels as well.

The *Overhead* for Cloverleaf mainly comes from caching the final iteration and handling Revolve checkpoints. For each problem, I chose the maximum saved Revolve checkpoint as $\frac{N_{iter}}{3}$, which shows the increased relative overhead for the 960^2 problem. However, the actual difference is amortized due to grouping the allocations for the checkpoints. The chosen checkpoint counts lead to spending $2.1 - 2.5x$ the time of the original application performing re-runs of forward kernels. In terms of actual adjoint kernels relative to the original application, OPS spends $3.3x$ (960^2) and $2.8x$ (4096^2) time on CPUs and $2.12x$ (960^2) and $3.2x$ (4096^2) time on GPUs.

For the CDE application, I fixed the checkpoint count to 10 for the 100 iteration, which leads to small overheads on the forward pass and $2.05 - 2.87x$ time spent on re-computing forward kernels (including forward kernels with caching). There is a large difference in the relative runtime of the adjoint computation between the OpenMP ($1024^2 - 5.62x, 4096^2 - 3.72x$) and the CUDA ($1024^2 - 1.4x, 4096^2 - 2.05x$) implementation. The main cause of the difference is in the time spent on the tridiagonal solver calls - the original application spends 92% for the small and 88% for the big mesh of the runtime in the tridiagonal solver calls in CUDA for the while for the OpenMP version this ratio is only 23% and 46% respectively. As I showed, the adjoint for the tridiagonal solve is another tridiagonal solve and a small loop on the derivatives leading to a small overhead on the adjoints. In terms of adjoint loop performance on the small mesh, both the OpenMP and CUDA versions compute the adjoints in $6.6 - 6.7x$ while we see a difference on the bigger mesh ($5.5x$ for OpenMP and $8.2x$ for CUDA) where the overhead from the reductions on the low dim datasets showing their effects with the increased number of blocks present.

Figure 4.7 clearly shows the effect of Revolve on the total runtime of the derivative propagation, and Figure 4.6 shows the effect on the memory consumption. The Poisson application shows the effect of naively implementing AD with caching all intermediate states. The original application uses 513 MiB memory (4 datasets with a size of 128 MiB each plus some additional memory in OPS). In addition, the active version allocates the adjoints for all these datasets and also saves an additional copy of the primal results. The value shown in Figure 4.6 for the tape corresponds to one time iteration where the loop writes one dataset. Hence, OPS will store 1 dataset worth of data for the time iteration, adding 128 MiB per iteration, a total of 12.5 GiB memory for the 100 iterations, and a runtime overhead on the forward pass, but potentially faster time to derivatives. This highlights the main benefit of the iterative approach since this version only caches data for one iteration, drastically decreasing the memory footprint of computing adjoints, resulting in a constant memory requirement for computing the adjoints instead of constantly increasing with the iteration count and the complexity of the computations. For a reasonably complex application, the size of the tape required for a single iteration drastically increases, making it infeasible to tape all iterations at once, and many applications are not fixed point iterations, ruling out reverse accumulation. For such applications, Revolve provides a solution to control the memory footprint at the cost of additional computing steps. Figure 4.6 also shows how the memory requirements change with the number of Revolve checkpoints used in Cloverleaf and CDE. The effect is twofold: OPS will cache only one iteration at a time (reducing the peak memory for this category by a factor of N_{iter} and fixing it to the *Tape* value shown in the figure) and increasing the memory usage by saving the Revolve checkpoints to memory. Both applications have one-time writes on a few datasets, leading to a bigger initial checkpoint, but all other checkpoints are the same size, increasing the peak memory footprint linearly. Without the active Revolve checkpoints, OPS can compute the adjoints using only $5.4 - 8.1x$ more memory for Cloverleaf and $3.2 - 3.4x$ more memory for CDE compared to the original application. Counting the 10 Revolve checkpoints used for CDE OPS computes the adjoints under $3.7x$ more memory.

In terms of runtime, Revolve will, in total, increase the runtime with the additional re-compute steps and the cost of actually saving the checkpoints. To get a clearer picture of the effect on the runtime, Figure 4.8 shows both Cloverleaf and CDE runtimes at different maximum active checkpoint counts. The theoretical minimum runtime overhead would arise if the application stores all Revolve checkpoints during the forward pass. Then, each iteration would be replayed once while saving all states. On the other end, using only a few Revolve checkpoints while having smaller memory requirements leads to significantly more re-compute steps. This highlights again the trade-off between memory and total runtime. While with each additional checkpoints increases the peak memory usage linearly with the size of a checkpoint of size showed in Figure 4.6. Figure 4.8 provides three value on the X axis: the number of active checkpoints used, the percentage of the checkpoints for the iterations can be saved with this amount of active checkpoints, and finally the average number of forward steps taken from each checkpoint[150]. Note that while the memory requirement increase linearly this average recompute number quickly closes to 2. In sync with this number, for both applications, the overhead of the re-compute steps quickly reduces re-computing the forward steps only a handful of times, and then allowing additional checkpoints to store to memory have diminishing returns. In addition, during re-computing loops, OPS can reduce the cost of execution by skipping reductions, which results are already saved during the first execution and would significantly impact loop performance.

4.4 Conclusion

In this chapter, I presented an extension to the OPS DSL to support adjoint mode algorithmic differentiation. OPS allows the expression of structured mesh applications from a high-level source and handles parallelism, data movement, and optimizations to different hardware backends. The extension provides performance portable derivatives for such applications on CPUs and GPUs. The AD extension leverages the domain-specific abstraction of OPS to perform optimizations that would be impossible with a more generic tool.

I introduced a model to describe data dependencies and data races in the adjoint of the computational loops described in the OPS abstraction. From the description of the original stencils, OPS calculates the scatter stencils used in the adjoint loops, which define the patterns where data races arise.

I used this model to map the adjoint loops to OpenMP and CUDA parallel implementations. OPS is free to generate any scheduling to avoid data races with synchronization or can generate code that uses atomic operations. I show that OPS can specialize the kernels to accommodate reductions on lower-dimensional datasets efficiently on both CPUs and GPUs. My approach builds a compact AD tape in OPS by handling derivatives at the loop level instead of expressions. Storing information only at the loop level drastically decreases the size of the tape compared to operator overloading-based tools. In addition, the high level of abstraction makes implementing extensions to the tape, like support for external adjoint functions, straightforward. I demonstrate the use of the external function API using linear solvers in an ADI application. Then, to give control over the memory overhead, I extended the tape with the Revolve checkpointing strategy. Revolve requires the ability to restore the state of the application at specific points and, from these points, recompute the original loops. I show that with code generation, OPS can optimize the recompute steps by skipping reductions, which would increase the run time of the kernels otherwise.

Experimental results on two memory bandwidth bound applications show that the adjoint propagation achieves similar bandwidth as the original primal application, achieving over 90%

of the peak bandwidth on test cases that do not fit in cache on CPUs and only dropping 7% compared to the original on smaller test cases. Similarly, on GPUs, active versions achieve $75 - 80\%$ of the peak bandwidth on test cases that can saturate the GPU. Compared to the original applications, the OpenMP versions on all three test applications computed adjoint under $10x$ overhead and on GPUs under $4.6 - 6.6x$.

I have integrated Revolve into OPS and showed that we can control the memory requirements of the adjoint evaluation as expected. This enables large simulations to run in adjoint mode, which would require too much memory otherwise by trading memory requirements for runtime with re-computing sections of the forward pass.

The adjoint model in OPS provides an efficient way to compute derivatives for performance-critical structured mesh applications. Using a DSL such as OPS unlocks the derivative computations on both hardware from the same high-level source, giving the $10\times$ speedup achieved on the A100 GPU compared to a single socket of the Intel(R) Xeon(R) Gold 6226R for these applications without any additional development cost.

5 Summary of the Dissertation

In this chapter, I provide a summary of the main scientific contributions of this dissertation and briefly discuss the possible application areas of the results.

5.1 Methods and tools

The first part of my research is based on the OP2 domain-specific language [12], which provides a high-level abstraction for the solution of unstructured-mesh applications defining an API to describe computational kernels with all the necessary information for orchestrating parallelism. I implemented the new source-to-source generator based on the refactoring tool support of Clang's LibTooling library. The correctness and performance of the generated code were tested on two applications written using the OP2 abstraction: a benchmark simulating the airflow around the wing of an aircraft called Airfoil and a tsunami simulation software called Volna [141].

The second part of my dissertation focuses on the distributed solution of batch-tridiagonal systems with special attention to applications using the Alternating-Direction Implicit method [59]. The base of this work is the single-node solver library Tridsolver [65]. I combined the exact iterative PCR algorithm [62] with the distributed communication strategies of the TridiagLU library [76].

Finally, the third part of my research focuses on the other domain-specific language of the OP-DSL family OPS [11] and computing sensitivity information for outputs. The structure and abstraction of OPS are similar to OP2's, but OPS targets structured meshes with stencil loops. I applied reverse (adjoint) mode algorithmic differentiation (AAD) to applications written in OPS. I used three applications to analyze the performance of AAD with OPS: a 2D code solving the Poisson equation using Jacobi iterations applying AAD directly and using fixed point iterations [152], the CloverLeaf[154] is a mini-app that solves the compressible Euler equations on a Cartesian grid using an explicit, second-order accurate method and a code for 2D convection-diffusion equation code from computational finance [153].

The applications and library extensions were implemented in C++ in combination with the CUDA language extension for targeting GPUs. While the code generator for OP2 was implemented in C++, the code generator for the AAD support of OPS was implemented in Python. I used the NCCL and MPI libraries for message passing for distributed memory parallelism, and for shared memory parallelism, I used OpenMP and CUDA.

For performance measurements, a range of hardware architectures and platforms were used. For benchmarking the scaling performance of the Tridsolver library, I used two of the UK's HPC systems: ARCHER2[1], a CrayEX system with AMD Rome CPUs (2×64 cores per node) and 256

[1]https://www.archer2.ac.uk/

GB of RAM, and Cirrus[2], a HPE/SGI system with 36 GPU nodes, each with 4×NVIDIA V100 16GB GPUs, interconnected with NVLink, and FDR Infiniband between nodes. For evaluating the performance of AAD in OPS, I ran the OpenMP measurements on a single socket of an Intel(R) Xeon(R) Gold 6226R CPU at 2.9 GHz without hyper-threading and 376 GB RAM and the CUDA measurements were executed using an AMD EPYC 7F72 24-core Processor and an NVidia A100 GPU with 40 GB RAM. In most cases, the runtimes reported are the results of averaging 10 repeated runs.

5.2 New Scientific Results

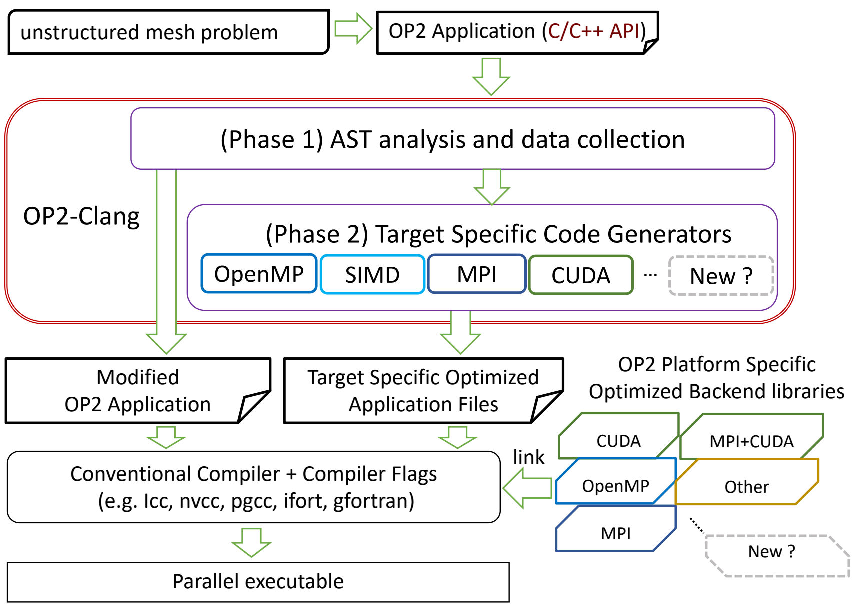
Thesis I. *I designed an automatic translation toolchain that uses a parallelization skeleton based approach. My solution improves the stability and robustness of source-to-source translation in the OP2 DSL, generating code for CPU clusters and GPUs, and improving memory locality. The performance of the generated code is demonstrated on a set of representative applications, and I performed a comparative analysis of the effects of various programming languages and compilers on the efficiency of parallel loops.*

Publications related to this thesis group are: [J1], [C1], [C2].

The OP2 API was constructed to make it easy for a parsing phase to extract the relevant information about each loop that will describe which computation and memory access patterns will be used - this is required for code generation aimed at different architectures and parallelization. The `op_arg_dat` provides all the details of how an `op_dat`'s data is accessed in the loop. With this information, the `op_par_loop` call contains all the necessary information about the computational loop to perform the parallelization. It is clear that due to the abstraction, the parallelization depends only on a handful of parameters, such as the existence of indirectly accessed data or reductions in the loop, plus the data access modes that lend to optimizations.

The fact that only a few parameters define the parallelization means that in the case of two computational loops, the generated parallel loops have the same lines of code with only small code sections with divergences. The identical chunks of code in the generated parallel loops are an important blueprint of the target code to be generated. This leads us to the idea of using a parallel implementation (with the invariant chunks) of a dummy loop and carrying out the code generation process as a refactoring or modification of this parallel loop. Figure 2.2 illustrates partial parallel skeletons we can extract for the generated OpenMP implementation for indirect loops. The code generator can use this dummy parallel loop as a skeleton (or template) and modify it to generate the required candidate computational loop. One can imagine similar skeletons for all target parallelizations. This approach can reduce the cost of introducing new targets since it requires only the implementation of a dummy loop to use as a skeleton instead of implementing code generation paths for the whole kernel. At the same time, since implementing a loop is much less error-prone than writing code to generate it, this approach reduces the risk of introducing bugs in the invariant bits of code in the parallel loops.

[2]https://www.cirrus.ac.uk/



unstructured mesh problem
OP2 Application (C/C++ API)
(Phase 1) AST analysis and data collection
OP2-Clang
(Phase 2) Target Specific Code Generators
OpenMP
SIMD
MPI
CUDA
...
New ?
Modified OP2 Application
Target Specific Optimized Application Files
OP2 Platform Specific Optimized Backend libraries
CUDA
MPI+CUDA
OpenMP
Other
MPI
New ?
link
Conventional Compiler + Compiler Flags (e.g. Icc, nvcc, pgcc, ifort, gfortran)
Parallel executable

Figure 5.1: The high-level architecture of OP2-Clang and its place within OP2

Based on the parallelization skeletons, the code generation for a parallel loop can be considered as a refactoring step. Clang's LibTooling library provides great support for code refactoring tasks by matching specific parts of the code's Abstract Syntax Tree (AST) and modifying the source code behind the matched nodes. To complete the whole process of source-to-source transformation in OP2, the code generator requires two steps. The first collects data about the parallel loops to be generated and replaces the original `op_par_loop` calls with calls to the generated functions, and the second is generating code for the target hardware as shown in Figure 5.1. The first phase parses all the arguments in the `op_par_loop` calls to collect all the information that is required to fill in the loop-specific code in the parallelization skeleton. The second phase will choose the appropriate skeleton, build the AST, and perform a set of refactoring steps, such as changing the function signature, to generate the final specialized loop implementation. This approach provides two advantages over the conventional Python code generator. The first is that the code generator can easily reuse bits of refactoring steps between target structures, making extensions for new targets easier. The second is that by using the compiler infrastructure, the new code generator can provide more sophisticated semantic checks over the generated code at the time of the translation.

Thesis II. *I designed a set of novel high-performance, scalable, distributed memory algorithms for the solution of batch-tridiagonal systems of equations, targeting large-scale heterogeneous supercomputers based on modern multi-core and many-core processor architectures. My algorithms can compute both the approximate and the exact solution of individual systems, and seamlessly integrates with Alternating-Direction Implicit methods commonly used in the solution of large-scale high-dimensional Partial Differential Equations. I published my implementation as an extension to the open-source Tridsolver library.*

Publications related to this thesis group are: [J2].

The state-of-the-art distributed memory algorithms for tridiagonal systems divide the system into subsystems and form a smaller decoupled tridiagonal system connecting the partitions (reduced system). Then, commonly, this reduced system is either gathered into a single process to solve, and then the solution is scattered among the processes or solved via iterative solver algorithms like Jacobi iterations. The former scales poorly due to the all-to-all communication patterns, and the latter, while using only point-to-point communications, produces approximate solutions.

In ADI, the coefficients are calculated for each grid point in a way that matches the underlying data structure of the application. MPI nodes are defined along all dimensions, and data for the diagonals are stored contiguously in either a row-major (Z is contiguous, Y and X are strided) or, more commonly, a column-major (X is contiguous, Y and Z are strided) format. This poses a challenge for algorithms that then solve multiple tridiagonal systems simultaneously; the different directions will use different memory layouts, which in turn require different optimizations. Moreover, improving on the state-of-the-art, our library supports all of the three different memory layouts possible for 3 or higher-dimensional problems.

By extending the Thomas-PCR hybrid algorithm to distributed memory environments, I designed a tridiagonal solver algorithm that gives exact solutions for batch tridiagonal problems while retaining the scaling properties of the approximate algorithms. The overall structure of the distributed tridiagonal solver can be summarized as follows. Each subsystem of size M belongs to a separate MPI process, which performs the hybrid Thomas-PCR forward pass. This produces a reduced system with two rows per MPI process. The solution to the reduced system is implemented using the distributed PCR algorithm. This algorithm uses only point-to-point communications, which is a crucial criterion for scalability. Once the reduced system is solved, the backward pass of the hybrid Thomas-PCR is performed on each MPI process.

	Communication	Number of messages	Message size
Thomas-Thomas(AG)	All-to-All	1	$2 \times N_p \times N_{sys}$
Thomas-Thomas(GS)	All-to-All	2	$\frac{2 \times N_p \times N_{sys}}{N_p}$
Thomas-PCR	One-to-One	$2 \times log_2 N_p + 2$	N_{sys}
Thomas-Jacobi	One-to-One	$2 \times J + 2$	N_{sys}

Table 5.1: Communication steps needed to solve the reduced system for each algorithm. N_p is the number of processes that share the same set of tridiagonal systems N_{sys} is the number of independent systems the processes share. J is the number of Jacobi iterations required. The message size is shown in terms of elements, each element requires to send the corresponding a_i, c_i, d_i coefficients ($b_i = 1$).

Table 5.1 shows a comparison between the three major solving strategies for the reduced system and the trade-offs for using them, where N_{proc} marks the number of MPI processes and N_{sys} marks the batch size. To create a scalable critical to avoid all-to-all communications, which would lead to message size correlating with the number of processes. For the algorithms using point-to-point communications, the number of messages and the distance between the communicating nodes affect the overhead of the communication. We can see that the PCR algorithm will use bigger messages between nodes that are further away from each other, but in return, it does not require any global communication and produces exact solutions.

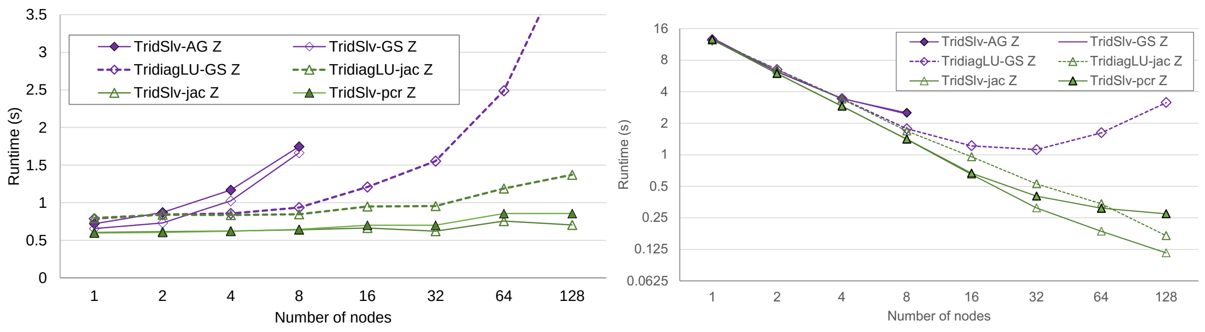


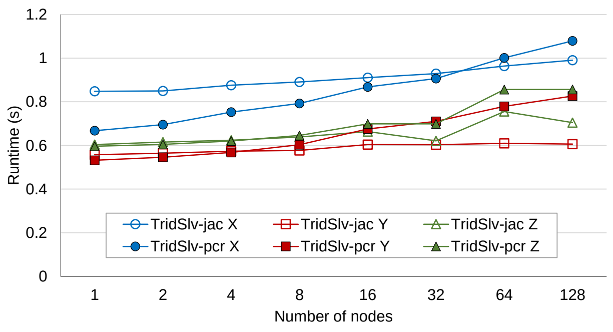
Figure 5.2: Comparison of the Tridsolver library to TridiagLU. **Left:** weak-scaling 512^3 grid points per node, **Right:** Strong-scaling, 8192 points in the direction of solve, and 512 in others. AG - AllGather, GS - Gather-Scatter

Two of the three steps in the hybrid algorithm scales trivially. The only part that contributes to the scaling properties is the distributed solver for the reduced system. Figure 5.2 shows that algorithms relying on global communication collectives take over the runtime after a certain point. For point-to-point communications, the message size and the distance of the nodes that are required to communicate are the two factors defining the overhead. In the results shown in Figure 5.2, I used a problem-specific upper bound on the number of Jacobi iterations instead of using global reduce calls to compute the error; hence, the Jacobi iteration used a fixed number of communications, and each node communicated only with neighboring nodes, but such an upper bound can't be defined for the general case. On the other hand, PCR has one additional communication step at each data point on the figure but does not need any problem-specific heuristics to compute the solution. The increasing cost of the communication clearly shows in the case of strong scaling after 16 MPI nodes, where the cost of the far messages (leaving local memory of ARCHER2 nodes) dominates the total runtime while still beating global communication patterns significantly.

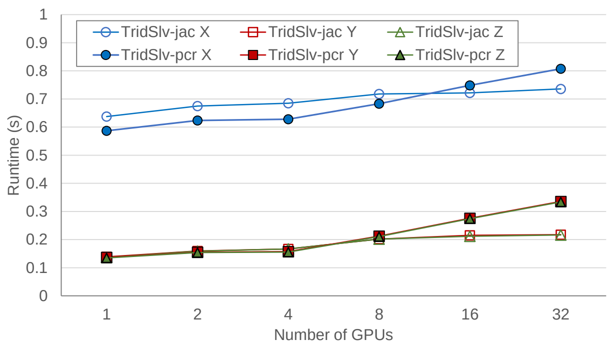
Figure 5.3 shows the scaling performance of the Tridsolver library for solver calls in all directions of a 3D application on ARCHER2 and on the Cirrus system. On CPUs, the PCR version achieves 70% scaling efficiency up to 128 nodes, while on GPUs, the cost of the communication outside a single Cirrus node (4 GPUs) is significantly higher due to slower interconnect, which has a great impact on the scaling of the PCR solver.

Thesis III. *I proposed an advanced, abstract computational model for reverse mode algorithmic differentiation of complex stencil applications and integrated it into the OPS domain-specific language. The model enables OPS to generate multi-core CPU and massively parallel GPU implementations for the adjoint loops, leveraging the metadata provided by the DSL. The model uses a new mapping of the algorithm to novel execution patterns for the adjoint loops. Furthermore, the extension enables OPS to follow the computational steps at a loop level by integrating an AD tape tailored for the OPS DSL, creating a streamlined storage mechanism thanks to the OPS abstractions.*

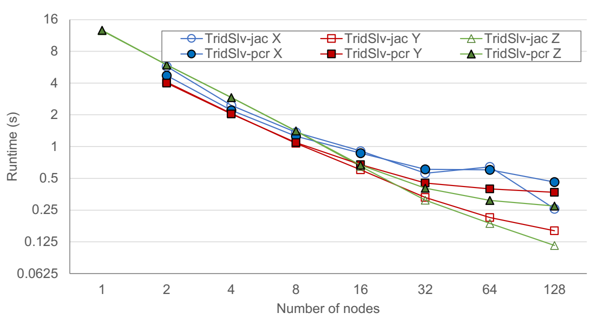
Publications related to this thesis group are: [J3], [C3].



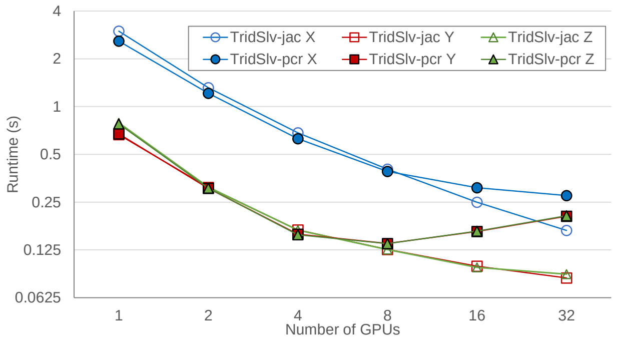
(a) Tridsolver weak scaling on ARCHER2



(b) Tridsolver weak scaling on Cirrus



(c) Tridsolver strong scaling on ARCHER2



(d) Tridsolver strong scaling on Cirrus

Figure 5.3: ARCHER2 scaling (MPI+OpenMP): (a),(c) Cirrus scaling (MPI+CUDA):(b),(d) - All weak-scaling using 512^3 points per node. Strong scaling on ARCHER2 uses 8192 points in the direction of solve while Cirrus measurements use 2048 points and 512 points in others.

Subthesis III.1. *I created a computational model based on the OPS abstraction for structured-mesh stencil applications that describes computational patterns, data, and control flow and described how adjoint mode Algorithmic Differentiation can be performed with this model. I extended the OPS abstraction to handle AD active datatypes and code regions with custom adjoint functions such as linear solvers.*

Computing sensitivities efficiently is crucial in many areas, and getting efficient parallel implementations of the gradient propagation in reverse mode AAD is especially challenging. The two key challenges of reverse mode AD in a parallel environment are the data races introduced by the reversal of the access patterns and following the control flow at runtime. The OPS API uses a description of loops and the data access inside the loops to generate efficient parallel implementations. Using this description, I created a model that describes the access patterns, describing the potential data races for the loops executing the adjoints of the loops, enabling the generation of parallel implementations. Building on the loop chain registered at run-time, this model enables OPS to compute the gradient through reverse-mode AD.

Subthesis III.2. *I designed and implemented a mapping of the high-level model to optimized, low-level parallel computational kernels supporting both multi-core CPUs and many-core GPUs with architecture-specific optimizations.*

In an OPS loop, each dataset is accessed through a stencil with an assigned access pattern:

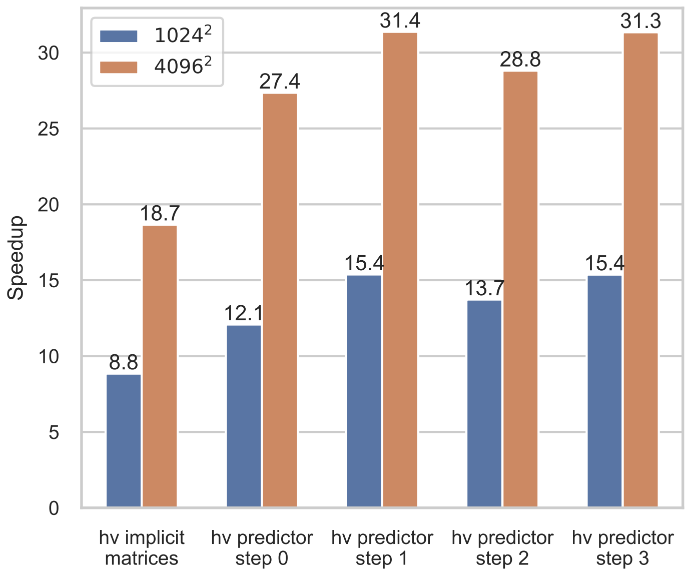


Figure 5.4: Speedup of the version using the access pattern aware reductions over atomic operations in kernels for the CDE application with a mesh size of 1024^2 and 4096^2.

read, write, or increment. All loops in OPS must use gather stencils only, meaning that read can appear with any stencil with any number of points, but increment and write stencils must have one single point access with zero offsets. In reverse mode AD in the adjoint loops, the data flow will be reversed from gathering stencils, and we will get scatter operations on the adjoint data. Due to the reversal, the write and increment stencils on the datasets will turn into one-point read stencils on the adjoint data and read stencils will turn into increment stencils with multiple points. The above has two implications: first, the primal loops are race-free, the parallelization is trivial, and second, the adjoint loops will have data races on the adjoint data. However, the location and structure of these data races are defined by the read stencils of the primal loops. The above has two implications: first, the primal loops are race-free, the parallelization is trivial, and second, the adjoint loops will have data races on the adjoint data. However, these data races are well defined by the read stencils of the primal loops. I devised and implemented an execution pattern that avoids race conditions on CPUs executing the loop in two sweeps with synchronization between. This approach avoids the cost of atomic operations. On GPUs, the cost of atomic operations is lower; hence, the adjoint kernels use atomic operations on the derivatives.

Another important access pattern in terms of performance is reading to lower dimensional data (data that is invariant in some dimension). These accesses will result in a large amount of writes on the same derivative values in the backward pass. I introduced a specialized code generation path for lower-dimensional datasets in CUDA adjoint kernels using reductions in only the required dimensions. Figure 5.4 shows the speedup gain on an NVidia A100 using this optimization.

In some cases, pure stencil computations cannot give sufficient performance or methods that cannot be represented in the abstraction of OPS are required. The standard solution in a non-AD context is to ask for access to the raw data behind the OPS datasets, perform the computations outside of OPS, and return the data to OPS. Following this philosophy, I designed an external

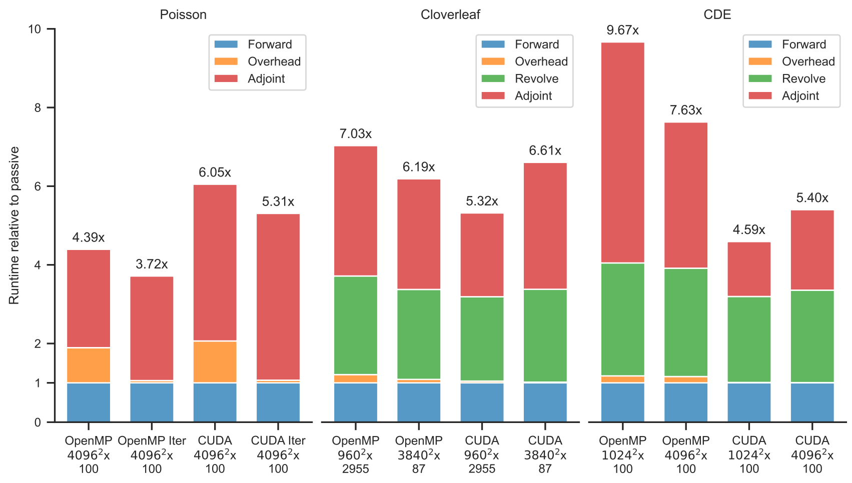


Figure 5.5: The overhead of AD on the benchmark applications, compared to a single evaluation of the passive, original application. The values show that evaluating the gradient takes N times of evaluating the original application. The **Overhead** values show the additional cost of collecting the DAG, saving intermediate states, and storing revolve checkpoints during the forward pass, **Revolve** values represent the additional time spent on replaying sections of the applications to restore states for the adjoints loops and the **Adjoint** part shows the time spent in the actual adjoint loops.

adjoint interface for OPS to express computations that require specialized adjoint functions. The API will ask for a description of the accesses in the external function and two function pointers, one for the primal and one for the adjoint function. This API enables the use of any computation that cannot be expressed as stencil loops or computations that can use specialized adjoint functions. I used two examples to demonstrate the use of this API. The CDE application uses external adjoint functions for linear solvers, where in each iteration, the ADI method requires multiple batch-tridiagonal solves. Here, the adjoint of the linear solver could be expressed as another solver call with an additional small parallel loop. The second example is the use of reverse accumulation for attractive fixed points in the Poisson code. This example demonstrates the use of multiple tapes and multiple evaluations on the same tape using the tape of one forward iteration as a fixed point iteration on the gradient. The performance of these versions is shown in Figure 5.5.

Subthesis III.3. *I extended the OPS abstraction to integrate the model and the mapping, automatically orchestrating the forward and adjoint computations, including control over the memory overhead of differentiation and enabling the efficient parallelization of the gradient computation. I demonstrated the utility and performance of this extension on industrially representative applications.*

Reverse mode AD tools can only follow the control flow at most an expression level, leading to high memory usage. Building on the OPS abstraction, I introduced a tape data structure that keeps track of the parallel loop descriptors and stores overwritten data for the loops. The high-level tape drastically reduces the memory overhead of storing control flow information. However, storing all overwritten data for a large number of iterations in an application would lead to huge tape. To address this issue, I extended the tape with the Revolve[150] checkpointing strategy, providing the user with fine control over the memory usage of the adjoint computation using loop re-execution to recompute the intermediate states.

I evaluated the performance of the gradient computation on three industrially representative applications. Figure 5.5 shows the relative runtime of the whole gradient computation(including the evaluation of the original function) compared to a single evaluation of the original application.

5.3 Potential applications and benefits

My work on the OP2-Clang tool was directly applicable as the source-to-source translation layer of the OP2 DSL. The use of a compiler-based tool could increase the robustness and diagnostic abilities of the code generation and make integration into industrial build systems easier at the same time.

Results carried out in the context of batch-tridiagonal solver libraries can be used in large-scale scientific applications using the ADI method. This research was performed partially in support of the UK's ExCALIBUR project, which aims to deliver the next generation of high-performance simulation software. As part of the project, a discussion of the use of the library in the xCompact3D library [148] is ongoing. This library is an industrial strength library for simulating turbulent flows and is used for simulations such as airflow around an entire wind farm.

Adjoint mode Algorithmic Differentiation is often used in computational fluid dynamics and computational finance. The results show how domain-specific languages or abstractions, in particular OPS, can drastically reduce memory overhead and improve the runtime of computing derivatives compared to general-purpose tools. The high-level OPS application code provides support for both CPUs and GPUs, which makes OPS one of the first performance portable adjoint mode AD libraries. Furthermore, with OPS's support for AD completed, we plan to enable this functionality in higher-level libraries building upon OPS, such as the Navier-Stokes solver OpenSBLI library, which focuses on shocks and boundary layer interactions.

List of author publications

List of journal publications

[J1] A. A. Sulyok, **G. D. Balogh**, I. Z. Reguly, and G. R. Mudalige, "Locality optimized unstructured mesh algorithms on gpus", *Journal of Parallel and Distributed Computing*, vol. 134, pp. 50–64, 2019, ISSN: 0743-7315. DOI: `https://doi.org/10.1016/j.jpdc.2019.07.011`. [Online]. Available: `https://www.sciencedirect.com/science/article/pii/S0743731519301698`.

[J2] **G. D. Balogh**, T. S. Flynn, S. Laizet, G. R. Mudalige, and I. Z. Reguly, "Scalable many-core algorithms for tridiagonal solvers", *Computing in Science & Engineering*, vol. 24, no. 1, pp. 26–35, 2022. DOI: `10.1109/MCSE.2021.3130544`.

[J3] **G. D. Balogh**, J. Lotz, J. Du Toit, U. Naumann, and I. Reguly, "Performance portable adjoints for structured mesh applications with ops", *ACM Trans. Math. Softw.*, **under review**.

List of conference publications

[C1] **G. D. Balogh**, I. Z. Reguly, and G. R. Mudalige, "Comparison of parallelisation approaches, languages, and compilers for unstructured mesh algorithms on gpus", in *High Performance Computing Systems. Performance Modeling, Benchmarking, and Simulation*, S. Jarvis, S. Wright, and S. Hammond, Eds., Cham: Springer International Publishing, 2018, pp. 22–43, ISBN: 978-3-319-72971-8.

[C2] **G. D. Balogh**, G. R. Mudalige, I. Z. Reguly, S. F. Antao, and C. Bertolli, "Op2-clang: A source-to-source translator using clang/llvm libtooling", in *2018 IEEE/ACM 5th Workshop on the LLVM Compiler Infrastructure in HPC (LLVM-HPC)*, 2018, pp. 59–70. DOI: `10.1109/LLVM-HPC.2018.8639205`.

[C3] **G. D. Balogh** and I. Z. Reguly, "Automatic parallel implementations of adjoint codes for structured mesh applications", in *2020 20th IEEE/ACM International Symposium on Cluster, Cloud and Internet Computing (CCGRID)*, 2020, pp. 908–911. DOI: `10.1109/CCGrid49817.2020.00019`.

[C4] **G. D. Balogh** and I. Reguly, "Automatic parallelisation of sturctured mesh computations with sycl", in *2021 IEEE International Conference on Cluster Computing (CLUSTER)*, 2021, pp. 821–822. DOI: `10.1109/Cluster48925.2021.00083`.

List of references related to the dissertation

[1] J. Nickolls, I. Buck, M. Garland, and K. Skadron, "Scalable parallel programming with cuda", *Queue*, vol. 6, no. 2, pp. 40–53, Mar. 2008, ISSN: 1542-7730. DOI: 10.1145/1365490.1365500. [Online]. Available: http://doi.acm.org/10.1145/1365490.1365500.

[2] J. E. Stone, D. Gohara, and G. Shi, "Opencl: A parallel programming standard for heterogeneous computing systems", *IEEE Des. Test*, vol. 12, no. 3, pp. 66–73, May 2010, ISSN: 0740-7475. DOI: 10.1109/MCSE.2010.69. [Online]. Available: http://dx.doi.org/10.1109/MCSE.2010.69.

[3] S. Wienke, P. Springer, C. Terboven, and D. an Mey, "Openacc: First experiences with real-world applications", in *Proceedings of the 18th International Conference on Parallel Processing*, ser. Euro-Par'12, Rhodes Island, Greece: Springer-Verlag, 2012, pp. 859–870, ISBN: 978-3-642-32819-0. DOI: 10.1007/978-3-642-32820-6_85. [Online]. Available: http://dx.doi.org/10.1007/978-3-642-32820-6_85.

[4] *OpenMP 4.5 specification*, http://www.openmp.org/wp-content/uploads/openmp-4.5.pdf.

[5] S. J. Pennycook, J. D. Sewall, D. W. Jacobsen, T. Deakin, and S. McIntosh-Smith, "Navigating performance, portability, and productivity", *Computing in Science & Engineering*, vol. 23, no. 5, pp. 28–38, 2021. DOI: 10.1109/MCSE.2021.3097276.

[6] *C++ Single-source Heterogeneous Programming for OpenCL*, http://www.khronos.org/sycl, accessed: 2023.

[7] C. R. Trott, D. Lebrun-Grandié, D. Arndt, *et al.*, "Kokkos 3: Programming model extensions for the exascale era", *IEEE Transactions on Parallel and Distributed Systems*, vol. 33, no. 4, pp. 805–817, 2022. DOI: 10.1109/TPDS.2021.3097283.

[8] H. C. Edwards, C. R. Trott, and D. Sunderland, "Kokkos: Enabling manycore performance portability through polymorphic memory access patterns", *Journal of Parallel and Distributed Computing*, vol. 74, no. 12, pp. 3202–3216, 2014, Domain-Specific Languages and High-Level Frameworks for High-Performance Computing, ISSN: 0743-7315. DOI: https://doi.org/10.1016/j.jpdc.2014.07.003. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S0743731514001257.

[9] R. D. Hornung and J. A. Keasler, "The RAJA portability layer: Overview and status", Lawrence Livermore National Lab. (LLNL), Tech. Rep., Sep. 2014. DOI: 10.2172/1169830.

[10] D. A. Beckingsale, J. Burmark, R. Hornung, *et al.*, "Raja: Portable performance for large-scale scientific applications", in *2019 ieee/acm international workshop on performance, portability and productivity in hpc (p3hpc)*, IEEE, 2019, pp. 71–81.

[11] I. Z. Reguly, G. R. Mudalige, and M. B. Giles, "Loop tiling in large-scale stencil codes at run-time with ops", *IEEE Transactions on Parallel and Distributed Systems*, vol. 29, no. 4, pp. 873–886, 2018. DOI: 10.1109/TPDS.2017.2778161.

[12] G. Mudalige, M. Giles, I. Reguly, C. Bertolli, and P. Kelly, "Op2: An active library framework for solving unstructured mesh-based applications on multi-core and many-core architectures", in *2012 Innovative Parallel Computing (InPar)*, 2012, pp. 1–12. DOI: 10.1109/InPar.2012.6339594.

[13] *OP-DSL: The Oxford Parallel Domain Specific Languages*, https://op-dsl.github.io, 2015.

[14] K. Asanović, R. Bodik, B. C. Catanzaro, *et al.*, "The landscape of parallel computing research: A view from berkeley", EECS Department, University of California, Berkeley, Tech. Rep. UCB/EECS-2006-183, 2006. [Online]. Available: http://www2.eecs.berkeley.edu/Pubs/TechRpts/2006/EECS-2006-183.html.

[15] M. Giles, G. Mudalige, C. Bertolli, P. Kelly, E. László, and I. Reguly, "An analytical study of loop tiling for a large-scale unstructured mesh application", in *2012 SC Companion: High Performance Computing, Networking Storage and Analysis*, 2012, pp. 477–482. DOI: 10.1109/SC.Companion.2012.68.

[16] M. Lange, N. Kukreja, M. Louboutin, *et al.*, "Devito: Towards a generic finite difference dsl using symbolic python", in *Proceedings of the 6th Workshop on Python for High-Performance and Scientific Computing*, ser. PyHPC '16, Salt Lake City, Utah: IEEE Press, 2016, pp. 67–75, ISBN: 978-1-5090-5220-2. DOI: 10.1109/PyHPC.2016.9.

[17] T. Gysi, C. Osuna, O. Fuhrer, M. Bianco, and T. C. Schulthess, "Stella: A domain-specific tool for structured grid methods in weather and climate models", in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, ser. SC '15, Austin, Texas: ACM, 2015, 41:1–41:12, ISBN: 978-1-4503-3723-6. DOI: 10.1145/2807591.2807627.

[18] *PSyclone Project - GitHub Repository*, https://github.com/stfc/PSyclone, 2018.

[19] I. Z. Reguly, G. R. Mudalige, C. Bertolli, *et al.*, "Acceleration of a full-scale industrial cfd application with op2", *IEEE Transactions on Parallel and Distributed Systems*, vol. 27, no. 5, pp. 1265–1278, 2016.

[20] K. B. Ølgaard, A. Logg, and G. N. Wells, "Automated Code Generation for Discontinuous Galerkin Methods", *CoRR*, vol. abs/1104.0628, 2011.

[21] F. Rathgeber, D. A. Ham, L. Mitchell, *et al.*, "Firedrake: Automating the finite element method by composing abstractions", *ACM Trans. Math. Softw.*, vol. 43, no. 3, 2016, ISSN: 0098-3500. DOI: 10.1145/2998441. [Online]. Available: https://doi.org/10.1145/2998441.

[22] C. T. Jacobs and M. D. Piggott, "Firedrake-Fluids v0.1: numerical modelling of shallow water flows using an automated solution framework", *Geoscientific Model Development*, vol. 8, no. 3, pp. 533–547, 2015. DOI: 10.5194/gmd-8-533-2015.

[23] P. Vincent, F. Witherden, B. Vermeire, J. S. Park, and A. Iyer, "Towards green aviation with python at petascale", in *SC16: International Conference for High Performance Computing, Networking, Storage and Analysis*, 2016, pp. 1–11. DOI: 10.1109/SC.2016.1.

[24] I. Z. Reguly, D. Gopinathan, J. H. Beck, M. B. Giles, S. Guillas, and F. Dias, "The volna-op2 tsunami code (version 1.0)", *Geoscientific Model Development Discussions*, 2018.

[25] G. Mudalige, M. Giles, J. Thiyagalingam, *et al.*, "Design and initial performance of a high-level unstructured mesh framework on heterogeneous parallel systems", *Parallel Computing*, vol. 39, no. 11, pp. 669–692, 2013, ISSN: 0167-8191. DOI: 10.1016/j.parco.2013.09.004.

[26] G. R. Mudalige, I. Z. Reguly, and M. B. Giles, "Auto-vectorizing a large-scale production unstructured-mesh cfd application", in *Proceedings of the 3rd Workshop on Programming Models for SIMD/Vector Processing*, ser. WPMVP '16, Barcelona, Spain: ACM, 2016, 5:1–5:8, ISBN: 978-1-4503-4060-1. DOI: 10.1145/2870650.2870651.

[27] I. Z. Reguly, E. László, G. R. Mudalige, and M. B. Giles, "Vectorizing unstructured mesh computations for many-core architectures", in *Proceedings of Programming Models and Applications on Multicores and Manycores*, ser. PMAM'14, Orlando, FL, USA: Association for Computing Machinery, 2018, pp. 39–50, ISBN: 9781450326575. DOI: 10.1145/2560683.2560686. [Online]. Available: https://doi.org/10.1145/2560683.2560686.

[28] I. Z. Reguly and G. R. Mudalige, "Productivity, performance, and portability for computational fluid dynamics applications", *Computers & Fluids*, vol. 199, p. 104 425, 2020, ISSN: 0045-7930. DOI: https://doi.org/10.1016/j.compfluid.2020.104425. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0045793020300013.

[29] C. Othmer, "A continuous adjoint formulation for the computation of topological and surface sensitivities of ducted flows", *International Journal for Numerical Methods in Fluids*, vol. 58, no. 8, pp. 861–877, 2008.

[30] K. Becker, K. Heitkamp, and E. Kügeler, "Recent progress in a hybrid-grid cfd solver for turbomachinery flows", in *Proceedings fifth European conference on computational fluid dynamics ECCOMAS CFD*, vol. 2010, 2010.

[31] Y. Wang, K. Hua, and J. Zhang, "Fast and high accuracy numerical methods for solving pdes in computational finance", in *2011 International Conference on Business Computing and Global Informatization*, IEEE, 2011, pp. 307–310.

[32] B. Düring, M. Fournié, and C. Heuer, "High-order compact finite difference schemes for option pricing in stochastic volatility models on non-uniform grids", *Journal of Computational and Applied Mathematics*, vol. 271, pp. 247–266, 2014.

[33] R. Mollapourasl, M. Haghi, and A. Heryudono, "Numerical simulation and applications of the convection–diffusion–reaction equation with the radial basis function in a finite-difference mode", *Journal of Computational Finance*, vol. 23, no. 5, 2020.

[34] A. Clevenhaus, M. Ehrhardt, and M. Gunther, "An adi sparse grid method for pricing efficiently american options under the heston model", *ADVANCES IN APPLIED MATHEMATICS AND MECHANICS*, vol. 13, no. 6, pp. 1384–1397, 2021.

[35] G. Mudalige, I. Reguly, S. Jammy, C. Jacobs, M. Giles, and N. Sandham, "Large-scale performance of a DSL-based multi-block structured-mesh application for Direct Numerical Simulation", *Journal of Parallel and Distributed Computing*, vol. 131, pp. 130–146, 2019, ISSN: 0743-7315. DOI: `10.1016/j.jpdc.2019.04.019`.

[36] I. Z. Reguly, A. M. Owenson, A. Powell, S. A. Jarvis, and G. R. Mudalige, "Under the hood of sycl–an initial performance analysis with an unstructured-mesh cfd application", in *High Performance Computing: 36th International Conference, ISC High Performance 2021, Virtual Event, June 24–July 2, 2021, Proceedings 36*, Springer, 2021, pp. 391–410.

[37] E. Raut, J. Meng, M. Araya-Polo, and B. Chapman, "Evaluating performance of openmp tasks in a seismic stencil application", in *OpenMP: Portable Multi-Level Parallelism on Modern Systems: 16th International Workshop on OpenMP, IWOMP 2020, Austin, TX, USA, September 22–24, 2020, Proceedings 16*, Springer, 2020, pp. 67–81.

[38] P. Yang, F. Dong, D. Williams, *et al.*, "Improving utility of gpu in accelerating industrial applications with user-centred automatic code translation", *IEEE Transactions on Industrial Informatics*, 2017.

[39] D. Williams, V. Codreanu, P. Yang, *et al.*, "Evaluation of autoparallelization toolkits for commodity gpus", in *International Conference on Parallel Processing and Applied Mathematics*, Springer, 2013, pp. 447–457.

[40] D. Unat, X. Cai, and S. B. Baden, "Mint: Realizing cuda performance in 3d stencil methods with annotated c", in *Proceedings of the international conference on Supercomputing*, ACM, 2011, pp. 214–224.

[41] D. J. Quinlan *et al.*, *Rose compiler project*, 2012.

[42] C. Bertolli, A. Betts, G. Mudalige, M. Giles, and P. Kelly, "Design and performance of the op2 library for unstructured mesh applications", in *European Conference on Parallel Processing*, Springer, 2011, pp. 191–200.

[43] S.-Z. Ueng, M. Lathara, S. S. Baghsorkhi, and W. H. Wen-mei, "Cuda-lite: Reducing gpu programming complexity", in *International Workshop on Languages and Compilers for Parallel Computing*, Springer, 2008, pp. 1–15.

[44] S. Lee, S.-J. Min, and R. Eigenmann, "Openmp to gpgpu: A compiler framework for automatic translation and optimization", *ACM Sigplan Notices*, vol. 44, no. 4, pp. 101–110, 2009.

[45] S.-I. Lee, T. A. Johnson, and R. Eigenmann, "Cetus–an extensible compiler infrastructure for source-to-source transformation", in *International Workshop on Languages and Compilers for Parallel Computing*, Springer, 2003, pp. 539–553.

[46] T. D. Han and T. S. Abdelrahman, "Hi cuda: A high-level directive-based language for gpu programming", in *Proceedings of 2nd Workshop on General Purpose Processing on Graphics Processing Units*, ACM, 2009, pp. 52–61.

[47] O. Krzikalla, K. Feldhoff, R. Müller-Pfefferkorn, and W. E. Nagel, "Scout: A source-to-source transformator for simd-optimizations", in *Proceedings of the 2011 International Conference on Parallel Processing - Volume 2*, ser. Euro-Par'11, Bordeaux, France: Springer-Verlag, 2012, pp. 137–145, ISBN: 978-3-642-29739-7. DOI: `10.1007/978-3-642-29740-3_17`.

[48] I. Z. Reguly, G. R. Mudalige, M. B. Giles, D. Curran, and S. McIntosh-Smith, "The ops domain specific abstraction for multi-block structured grid computations", in *Proceedings of the 2014 Fourth International Workshop on Domain-Specific Languages and High-Level Frameworks for High Performance Computing*, ser. WOLFHPC '14, IEEE Computer Society, 2014, pp. 58–67, ISBN: 978-1-4673-6757-8. DOI: `10.1109/WOLFHPC.2014.7`.

[49] C. T. Jacobs, S. P. Jammy, and N. D. Sandham, "OpenSBLI: A framework for the automated derivation and parallel execution of finite difference solvers on a range of computer architectures", *Journal of Computational Science*, vol. 18, pp. 12–23, 2017.

[50] M. Marangoni and T. Wischgoll, "Togpu: Automatic source transformation from c++ to cuda using clang/llvm", *Electronic Imaging*, vol. 2016, no. 1, pp. 1–9, 2016.

[51] N. Jacobsen, "Llvm supported source-to-source translation-translation from annotated c/c++ to cuda c/c++", M.S. thesis, 2016.

[52] J. Choi, J. Demmel, I. Dhillon, *et al.*, "Scalapack: A portable linear algebra library for distributed memory computers — design issues and performance", *Computer Physics Communications*, vol. 97, no. 1, pp. 1–15, 1996, High-Performance Computing in Science, ISSN: 0010-4655. DOI: `https://doi.org/10.1016/0010-4655(96)00017-3`. [Online]. Available: `https://www.sciencedirect.com/science/article/pii/0010465596000173`.

[53] J. Choi, J. Dongarra, R. Pozo, and D. Walker, "Scalapack: A scalable linear algebra library for distributed memory concurrent computers", in *The Fourth Symposium on the Frontiers of Massively Parallel Computation*, Los Alamitos, CA, USA: IEEE Computer Society, Oct. 1992, pp. 120, 121, 122, 123, 124, 125, 126, 127. DOI: `10.1109/FMPC.1992.234898`. [Online]. Available: `https://doi.ieeecomputersociety.org/10.1109/FMPC.1992.234898`.

[54] S. Tomov, J. Dongarra, and M. Baboulin, "Towards dense linear algebra for hybrid GPU accelerated manycore systems", *Parallel Computing*, vol. 36, no. 5-6, pp. 232–240, Jun. 2010, ISSN: 0167-8191. DOI: `10.1016/j.parco.2009.12.005`.

[55] J. Dongarra, M. Gates, A. Haidar, *et al.*, "Accelerating numerical dense linear algebra calculations with gpus", *Numerical Computations with GPUs*, pp. 1–26, 2014.

[56] A. Buttari, J. Langou, J. Kurzak, and J. Dongarra, "A class of parallel tiled linear algebra algorithms for multicore architectures", *Parallel Computing*, vol. 35, no. 1, pp. 38–53, 2009, ISSN: 0167-8191. DOI: `https://doi.org/10.1016/j.parco.2008.10.002`. [Online]. Available: `https://www.sciencedirect.com/science/article/pii/S0167819108001117`.

[57] C. Yang, A. Buluç, and J. D. Owens, "Graphblast: A high-performance linear algebra-based graph framework on the gpu", *ACM Transactions on Mathematical Software (TOMS)*, vol. 48, no. 1, pp. 1–51, 2022.

[58] H. Anzt, T. Cojean, G. Flegar, *et al.*, "Ginkgo: A modern linear operator algebra framework for high performance computing", vol. 48, no. 1, Feb. 2022, ISSN: 0098-3500. DOI: `10.1145/3480935`. [Online]. Available: `https://doi.org/10.1145/3480935`.

[59] D. W. Peaceman and H. H. Rachford Jr, "The numerical solution of parabolic and elliptic differential equations", *Journal of the Society for industrial and Applied Mathematics*, vol. 3, no. 1, pp. 28–41, 1955. DOI: `10.1137/0103003`.

[60] L. Thomas, "Elliptic problems in linear differential equations over a network: Watson scientific computing laboratory", *Columbia Univ., NY*, 1949.

[61] R. A. Sweet, "A cyclic reduction algorithm for solving block tridiagonal systems of arbitrary dimension", *SIAM Journal on Numerical Analysis*, vol. 14, no. 4, pp. 706–720, 1977. DOI: `10.1137/0714048`.

[62] W. Gander and G. H. Golub, "Cyclic reduction—history and applications", *Scientific computing (Hong Kong, 1997)*, vol. 7385, pp. 73–86, 1997.

[63] Y. Zhang, J. Cohen, and J. D. Owens, "Fast tridiagonal solvers on the gpu", *SIGPLAN Not.*, vol. 45, no. 5, pp. 127–136, Jan. 2010, ISSN: 0362-1340. DOI: `10.1145/1837853.1693472`. [Online]. Available: `https://doi.org/10.1145/1837853.1693472`.

[64] H. Kim, S. Wu, L. Chang, and W. W. Hwu, "A scalable tridiagonal solver for gpus", in *2011 International Conference on Parallel Processing*, 2011, pp. 444–453. DOI: `10.1109/ICPP.2011.41`.

[65] E. Laszlo, M. Giles, and J. Appleyard, "Manycore algorithms for batch scalar and block tridiagonal solvers", *ACM Transactions on Mathematical Software (TOMS)*, vol. 42, no. 4, pp. 1–36, 2016. DOI: `10.1145/2830568`.

[66] Y. Zhang, J. Cohen, A. A. Davidson, and J. D. Owens, "Chapter 11 - a hybrid method for solving tridiagonal systems on the gpu", in *GPU Computing Gems Jade Edition*, ser. Applications of GPU Computing Series, W.-m. W. Hwu, Ed., Boston: Morgan Kaufmann, 2012, pp. 117–132, ISBN: 978-0-12-385963-1. DOI: `10.1016/B978-0-12-385963-1.00011-3`.

[67] *Tridiagonal solvers on the GPU and applications to fluid simulation*, Presented at the GPU Technology Conference, San Jose, CA. Retrieved June 02, 2021 from `https://www.nvidia.com/content/GTC/documents/1058_GTC09.pdf`.

[68] J. Hofhaus and E. F. Van de Velde, "Alternating-direction line-relaxation methods on multicomputers", *SIAM Journal on Scientific Computing*, vol. 17, no. 2, pp. 454–478, 1996.

[69] N. Mattor, T. J. Williams, and D. W. Hewett, "Algorithm for solving tridiagonal matrix problems in parallel", *Parallel Computing*, vol. 21, no. 11, pp. 1769–1782, 1995.

[70] H. H. Wang, "A parallel method for tridiagonal equations", *ACM Trans. Math. Softw.*, vol. 7, no. 2, pp. 170–183, Jun. 1981, ISSN: 0098-3500. DOI: `10.1145/355945.355947`. [Online]. Available: `https://doi.org/10.1145/355945.355947`.

[71] S. Bondeli, “Divide and conquer: A parallel algorithm for the solution of a tridiagonal linear system of equations”, *Parallel Computing*, vol. 17, no. 4, pp. 419–434, 1991, ISSN: 0167-8191. DOI: https://doi.org/10.1016/S0167-8191(05)80145-0.

[72] H.-S. Kim, S. Wu, L.-w. Chang, and W.-m. W. Hwu, “A scalable tridiagonal solver for gpus”, in *2011 International Conference on Parallel Processing*, 2011, pp. 444–453. DOI: 10.1109/ICPP.2011.41.

[73] L. Chang, J. A. Stratton, H. Kim, and W. W. Hwu, “A scalable, numerically stable, high-performance tridiagonal solver using gpus”, in *SC’12: Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, IEEE, 2012, pp. 1–11.

[74] E. Polizzi and A. H. Sameh, “A parallel hybrid banded system solver: The spike algorithm”, *Parallel Computing*, vol. 32, no. 2, pp. 177–194, 2006, Parallel Matrix Algorithms and Applications (PMAA’04), ISSN: 0167-8191. DOI: https://doi.org/10.1016/j.parco.2005.07.005. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0167819105001353.

[75] A. Pérez Diéguez, M. Amor López, and R. Doallo Biempica, “Solving multiple tridiagonal systems on a multi-gpu platform”, in *2018 26th Euromicro International Conference on Parallel, Distributed and Network-based Processing (PDP)*, 2018, pp. 759–763. DOI: 10.1109/PDP2018.2018.00123.

[76] D. Ghosh, E. M. Constantinescu, and J. Brown, “Efficient implementation of nonlinear compact schemes on massively parallel platforms”, *SIAM Journal on Scientific Computing*, vol. 37, no. 3, pp. C354–C383, 2015. DOI: 10.1137/140989261.

[77] K.-H. Kim, J.-H. Kang, X. Pan, and J.-I. Choi, “Pascal_tdma: A library of parallel and scalable solvers for massive tridiagonal system”, *Computer Physics Communications*, vol. 260, p. 107 722, 2021, ISSN: 0010-4655. DOI: https://doi.org/10.1016/j.cpc.2020.107722.

[78] K.-H. Kim, J.-H. Kang, and J.-I. Choi, *Parallel and scalable library for tridiagonal matrix algorithm*, 2019. [Online]. Available: https://github.com/MPMC-Lab/PaScaL_TDMA.

[79] M. B. Giles and N. A. Pierce, “An introduction to the adjoint approach to design”, *Flow, turbulence and combustion*, vol. 65, pp. 393–415, 2000.

[80] C. Bischof, G. Corliss, L. Green, A. Griewank, K. Haigler, and P. Newman, “Automatic differentiation of advanced cfd codes for multidisciplinary design”, *Computing Systems in Engineering*, vol. 3, no. 6, pp. 625–637, 1992.

[81] A. Carle, L. L. Green, P. Newman, and C. Bischof, “Applications of automatic differentiation in cfd”, *NASA STI/Recon Technical Report N*, vol. 95, p. 16 828, 1994.

[82] R. Sanchez, T. Albring, R. Palacios, N. Gauger, T. Economon, and J. Alonso, “Coupled adjoint-based sensitivities in large-displacement fluid-structure interaction using algorithmic differentiation”, *International Journal for Numerical Methods in Engineering*, vol. 113, no. 7, pp. 1081–1107, 2018.

[83] T. A. Albring, M. Sagebaum, and N. R. Gauger, “Efficient aerodynamic design using the discrete adjoint method in su2”, in *17th AIAA/ISSMO Multidisciplinary Analysis and Optimization Conference.* 2016, p. 3518. DOI: 10.2514/6.2016-3518. eprint: https://arc.aiaa.org/doi/pdf/10.2514/6.2016-3518. [Online]. Available: https://arc.aiaa.org/doi/abs/10.2514/6.2016-3518.

[84] A. G. Baydin, B. A. Pearlmutter, A. A. Radul, and J. M. Siskind, “Automatic differentiation in machine learning: A survey”, *Journal of Marchine Learning Research*, vol. 18, pp. 1–43, 2018.

[85] L. Guasch, O. Calderón Agudo, M.-X. Tang, P. Nachev, and M. Warner, “Full-waveform inversion imaging of the human brain”, *NPJ digital medicine*, vol. 3, no. 1, p. 28, 2020.

[86] C. Homescu, “Adjoints and automatic (algorithmic) differentiation in computational finance”, *Available at SSRN 1828503*, 2011.

[87] S. Jain, Á. Leitao, and C. W. Oosterlee, “Rolling adjoints: Fast greeks along monte carlo scenarios for early-exercise options”, *Journal of Computational Science*, vol. 33, pp. 95–112, 2019, ISSN: 1877-7503. DOI: https://doi.org/10.1016/j.jocs.2019.03.001. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S1877750318312547.

[88] T.-M. Li, M. Gharbi, A. Adams, F. Durand, and J. Ragan-Kelley, “Differentiable programming for image processing and deep learning in halide”, *ACM Transactions on Graphics (ToG)*, vol. 37, no. 4, pp. 1–13, 2018.

[89] M. Grabner, T. Pock, T. Gross, and B. Kainz, “Automatic differentiation for gpu-accelerated 2d/3d registration”, in *Advances in Automatic Differentiation*, C. H. Bischof, H. M. Bücker, P. Hovland, U. Naumann, and J. Utke, Eds., Berlin, Heidelberg: Springer Berlin Heidelberg, 2008, pp. 259–269, ISBN: 978-3-540-68942-3.

[90] L. Capriotti and J. Lee, “Case studies of real-time risk management via adjoint algorithmic differentiation (aad)”, in *High-Performance Computing in Finance*, Chapman and Hall/CRC, 2018, pp. 339–370.

[91] U. Naumann, *The art of differentiating computer programs: an introduction to algorithmic differentiation.* SIAM, 2011. DOI: 10.1137/1.9781611972078.

[92] C. C. Margossian, “A review of automatic differentiation and its efficient implementation”, *Wiley interdisciplinary reviews: data mining and knowledge discovery*, vol. 9, no. 4, e1305, 2019.

[93] E. Özkaya and N. R. Gauger, “Automatic transition from simulation to one-shot shape optimization with navier-stokes equations”, *GAMM-Mitteilungen*, vol. 33, no. 2, pp. 133–147, 2010. DOI: https://doi.org/10.1002/gamm.201010011. eprint: https://onlinelibrary.wiley.com/doi/pdf/10.1002/gamm.201010011. [Online]. Available: https://onlinelibrary.wiley.com/doi/abs/10.1002/gamm.201010011.

[94] T. Verstraete, L. Müller, and J.-D. Müller, “Adjoint-based design optimisation of an internal cooling channel u-bend for minimised pressure losses”, *International Journal of Turbomachinery, Propulsion and Power*, vol. 2, no. 2, 2017, ISSN: 2504-186X. DOI: `10.3390/ijtpp2020010`. [Online]. Available: `https://www.mdpi.com/2504-186X/2/2/10`.

[95] S. Vitale, M. Pini, and P. Colonna, “Multistage turbomachinery design using the discrete adjoint method within the open-source software su2”, *Journal of Propulsion and Power*, vol. 36, no. 3, pp. 465–478, 2020. DOI: `10.2514/1.B37685`. eprint: `https://doi.org/10.2514/1.B37685`. [Online]. Available: `https://doi.org/10.2514/1.B37685`.

[96] E. Larour, J. Utke, A. Bovin, M. Morlighem, and G. Perez, “An approach to computing discrete adjoints for mpi-parallelized models applied to ice sheet system model 4.11”, *Geoscientific Model Development*, vol. 9, no. 11, pp. 3907–3918, 2016. DOI: `10.5194/gmd-9-3907-2016`. [Online]. Available: `https://gmd.copernicus.org/articles/9/3907/2016/`.

[97] M. Towara, M. Schanen, and U. Naumann, “Mpi-parallel discrete adjoint openfoam”, *Procedia Computer Science*, vol. 51, pp. 19–28, 2015, International Conference On Computational Science, ICCS 2015, ISSN: 1877-0509. DOI: `https://doi.org/10.1016/j.procs.2015.05.181`.

[98] M. Towara and U. Naumann, “A discrete adjoint model for openfoam”, *Procedia Computer Science*, vol. 18, pp. 429–438, 2013, 2013 International Conference on Computational Science, ISSN: 1877-0509. DOI: `https://doi.org/10.1016/j.procs.2013.05.206`.

[99] J.-D. Mueller, J. Hueckelheim, and O. Mykhaskiv, “Stamps: A finite-volume solver framework for adjoint codes derived with source-transformation ad”, in *2018 Multidisciplinary Analysis and Optimization Conference*. 2018, p. 2928. DOI: `10.2514/6.2018-2928`. eprint: `https://arc.aiaa.org/doi/pdf/10.2514/6.2018-2928`. [Online]. Available: `https://arc.aiaa.org/doi/abs/10.2514/6.2018-2928`.

[100] R. J. Hogan, “Fast reverse-mode automatic differentiation using expression templates in c++”, *ACM Trans. Math. Softw.*, vol. 40, no. 4, 2014, ISSN: 0098-3500. DOI: `10.1145/2560359`. [Online]. Available: `https://doi.org/10.1145/2560359`.

[101] J. Lotz, “Hybrid approaches to adjoint code generation with dco/c++”, Dissertation, Department of Computer Science, RWTH Aachen University, 2016. [Online]. Available: `http://publications.rwth-aachen.de/record/667318`.

[102] D. M. Gay, “Semiautomatic differentiation for efficient gradient computations”, in *Automatic Differentiation: Applications, Theory, and Implementations*, M. Bücker, G. Corliss, U. Naumann, P. Hovland, and B. Norris, Eds., Berlin, Heidelberg: Springer Berlin Heidelberg, 2006, pp. 147–158, ISBN: 978-3-540-28438-3.

[103] M. Sagebaum, T. Albring, and N. R. Gauger, “Expression templates for primal value taping in the reverse mode of algorithmic differentiation”, *Optimization Methods and Software*, vol. 33, no. 4-6, pp. 1207–1231, 2018. DOI: `10.1080/10556788.2018.1471140`. eprint: `https://doi.org/10.1080/10556788.2018.1471140`. [Online]. Available: `https://doi.org/10.1080/10556788.2018.1471140`.

[104] B. Carpenter, M. D. Hoffman, M. Brubaker, D. Lee, P. Li, and M. Betancourt, “The stan math library: Reverse-mode automatic differentiation in c++”, *arXiv preprint arXiv:1509.07164*, 2015.

[105] S. H. K. Narayanan, B. Norris, and B. Winnicka, “Adic2: Development of a component source transformation system for differentiating c and c++”, *Procedia Computer Science*, vol. 1, no. 1, pp. 1845–1853, 2010, ICCS 2010, ISSN: 1877-0509. DOI: `https://doi.org/10.1016/j.procs.2010.04.206`. [Online]. Available: `https://www.sciencedirect.com/science/article/pii/S1877050910002073`.

[106] L. Hascoet and V. Pascual, “The tapenade automatic differentiation tool: Principles, model, and specification”, *ACM Transactions on Mathematical Software (TOMS)*, vol. 39, no. 3, 2013, ISSN: 0098-3500. DOI: `10.1145/2450153.2450158`. [Online]. Available: `https://doi.org/10.1145/2450153.2450158`.

[107] J. Utke, U. Naumann, M. Fagan, *et al.*, “Openad/f: A modular open-source tool for automatic differentiation of fortran codes”, *ACM Trans. Math. Softw.*, vol. 34, no. 4, 2008, ISSN: 0098-3500. DOI: `10.1145/1377596.1377598`. [Online]. Available: `https://doi.org/10.1145/1377596.1377598`.

[108] H. M. Bücker, B. Lang, D. an Mey, and C. H. Bischof, “Bringing together automatic differentiation and openmp”, in *Proceedings of the 15th International Conference on Supercomputing*, ser. ICS ’01, Sorrento, Italy: Association for Computing Machinery, 2001, pp. 246–251, ISBN: 158113410X. DOI: `10.1145/377792.377842`. [Online]. Available: `https://doi.org/10.1145/377792.377842`.

[109] H. Bucker, B. Lang, A. Rasch, C. Bischof, and D. an Mey, “Explicit loop scheduling in openmp for parallel automatic differentiation”, in *Proceedings 16th Annual International Symposium on High Performance Computing Systems and Applications*, 2002, pp. 121–126. DOI: `10.1109/HPCSA.2002.1019144`.

[110] H. M. Bücker, A. Rasch, and A. Wolf, “A class of openmp applications involving nested parallelism”, in *Proceedings of the 2004 ACM Symposium on Applied Computing*, ser. SAC ’04, Nicosia, Cyprus: Association for Computing Machinery, 2004, pp. 220–224, ISBN: 1581138121. DOI: `10.1145/967900.967948`. [Online]. Available: `https://doi.org/10.1145/967900.967948`.

[111] E. Phipps, R. Pawlowski, and C. Trott, “Automatic differentiation of c++ codes on emerging manycore architectures with sacado”, *ACM Trans. Math. Softw.*, vol. 48, no. 4, 2022, ISSN: 0098-3500. DOI: `10.1145/3560262`. [Online]. Available: `https://doi.org/10.1145/3560262`.

[112] P. Heimbach, C. Hill, and R. Giering, “Automatic generation of efficient adjoint code for a parallel navier-stokes solver”, in *Computational Science — ICCS 2002*, P. M. A. Sloot, A. G. Hoekstra, C. J. K. Tan, and J. J. Dongarra, Eds., Berlin, Heidelberg: Springer Berlin Heidelberg, 2002, pp. 1019–1028, ISBN: 978-3-540-46080-0.

[113] M. Schanen, U. Naumann, L. Hascoët, and J. Utke, “Interpretative adjoints for numerical simulation codes using mpi”, *Procedia Computer Science*, vol. 1, no. 1, pp. 1825–1833, 2010, ICCS 2010, ISSN: 1877-0509. DOI: `https://doi.org/10.1016/j.procs.2010.04.204`.

[114] M. Schanen, M. Förster, J. Lotz, K. Leppkes, and U. Naumann, "Adjoining hybrid parallel code", in *Proceedings of the eighth international conference on engineering computational technology*, Citeseer, vol. 100, 2012, p. 18.

[115] J. Lotz, U. Naumann, M. Sagebaum, and M. Schanen, "Discrete adjoints of petsc through dco/c++ and adjoint mpi", in *Euro-Par 2013 Parallel Processing*, F. Wolf, B. Mohr, and D. an Mey, Eds., Berlin, Heidelberg: Springer Berlin Heidelberg, 2013, pp. 497–507, ISBN: 978-3-642-40047-6.

[116] J. Blühdorn, M. Sagebaum, and N. Gauger, "Event-based automatic differentiation of openmp with opdilib", *ACM Trans. Math. Softw.*, vol. 49, no. 1, 2023, ISSN: 0098-3500. DOI: `10.1145/3570159`. [Online]. Available: `https://doi.org/10.1145/3570159`.

[117] M. Sagebaum, T. Albring, and N. R. Gauger, "High-performance derivative computations using codipack", *ACM Trans. Math. Softw.*, vol. 45, no. 4, 2019, ISSN: 0098-3500. DOI: `10.1145/3356900`. [Online]. Available: `https://doi.org/10.1145/3356900`.

[118] R. Giering, T. Kaminski, R. Todling, R. Errico, R. Gelaro, and N. Winslow, "Tangent linear and adjoint versions of nasa/gmao's fortran 90 global weather forecast model", in *Automatic Differentiation: Applications, Theory, and Implementations*, M. Bücker, G. Corliss, U. Naumann, P. Hovland, and B. Norris, Eds., Berlin, Heidelberg: Springer Berlin Heidelberg, 2006, pp. 275–284, ISBN: 978-3-540-28438-3.

[119] R. Giering, T. Kaminski, and T. Slawig, "Generating efficient derivative code with taf: Adjoint and tangent linear euler flow around an airfoil", *Future Generation Computer Systems*, vol. 21, no. 8, pp. 1345–1355, 2005, ISSN: 0167-739X. DOI: `https://doi.org/10.1016/j.future.2004.11.003`. [Online]. Available: `https://www.sciencedirect.com/science/article/pii/S0167739X04001785`.

[120] T. Kaler, T. B. Schardl, B. Xie, *et al.*, "Parad: A work-efficient parallel algorithm for reverse-mode automatic differentiation", in *Symposium on Algorithmic Principles of Computer Systems (APOCS)*. 2021, pp. 144–158. DOI: `10.1137/1.9781611976489.11`. eprint: `https://epubs.siam.org/doi/pdf/10.1137/1.9781611976489.11`. [Online]. Available: `https://epubs.siam.org/doi/abs/10.1137/1.9781611976489.11`.

[121] J. Hückelheim and L. Hascoët, "Source-to-source automatic differentiation of openmp parallel loops", *ACM Trans. Math. Softw.*, vol. 48, no. 1, 2022, ISSN: 0098-3500. DOI: `10.1145/3472796`. [Online]. Available: `https://doi.org/10.1145/3472796`.

[122] J. Hückelheim and L. Hascoët, "Automatic differentiation of parallel loops with formal methods", in *Proceedings of the 51st International Conference on Parallel Processing*, ser. ICPP '22, Bordeaux, France: Association for Computing Machinery, 2023, ISBN: 9781450397339. DOI: `10.1145/3545008.3545089`. [Online]. Available: `https://doi.org/10.1145/3545008.3545089`.

[123] W. Moses and V. Churavy, "Instead of rewriting foreign code for machine learning, automatically synthesize fast gradients", in *Advances in Neural Information Processing Systems*, H. Larochelle, M. Ranzato, R. Hadsell, M. Balcan, and H. Lin, Eds., vol. 33, Curran Associates, Inc., 2020, pp. 12 472–12 485.

[124] W. S. Moses, S. H. K. Narayanan, L. Paehler, *et al.*, “Scalable automatic differentiation of multiple parallel paradigms through compiler augmentation”, in *SC22: International Conference for High Performance Computing, Networking, Storage and Analysis*, IEEE, 2022, pp. 1–18.

[125] J. Hückelheim, N. Kukreja, S. H. K. Narayanan, F. Luporini, G. Gorman, and P. Hovland, “Automatic differentiation for adjoint stencil loops”, in *Proceedings of the 48th International Conference on Parallel Processing*, ser. ICPP ’19, Kyoto, Japan: Association for Computing Machinery, 2019, ISBN: 9781450362955. DOI: `10.1145/3337821.3337906`. [Online]. Available: `https://doi.org/10.1145/3337821.3337906`.

[126] F. Gremse, A. Höfter, L. Razik, F. Kiessling, and U. Naumann, “Gpu-accelerated adjoint algorithmic differentiation”, *Computer Physics Communications*, vol. 200, pp. 300–311, 2016, ISSN: 0010-4655. DOI: `https://doi.org/10.1016/j.cpc.2015.10.027`. [Online]. Available: `https://www.sciencedirect.com/science/article/pii/S0010465515004099`.

[127] J. Blühdorn, N. R. Gauger, and M. Kabel, “Automat: Automatic differentiation for generalized standard materials on gpus”, *Computational Mechanics*, vol. 69, no. 2, pp. 589–613, 2022. DOI: `10.1007/s00466-021-02105-2`. [Online]. Available: `https://doi.org/10.1007/s00466-021-02105-2`.

[128] W. S. Moses, V. Churavy, L. Paehler, *et al.*, “Reverse-mode automatic differentiation and optimization of gpu kernels via enzyme”, in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, ser. SC ’21, St. Louis, Missouri: Association for Computing Machinery, 2021, ISBN: 9781450384421. DOI: `10.1145/3458817.3476165`. [Online]. Available: `https://doi.org/10.1145/3458817.3476165`.

[129] “Libtooling”. (2018), [Online]. Available: `%5Curl%7Bhttp://clang.llvm.org/docs/LibTooling.html%7D` (visited on 08/13/2018).

[130] E. Bendersky. “Modern source-to-source transformation with clang and libtooling”. (2014), [Online]. Available: `https://eli.thegreenplace.net/2014/05/01/modern-source-to-source-transformation-with-clang-and-libtooling` (visited on 04/07/2018).

[131] J. Wu, A. Belevich, E. Bendersky, *et al.*, “Gpucc: An open-source gpgpu compiler”, in *Proceedings of the 2016 International Symposium on Code Generation and Optimization*, ACM, 2016, pp. 105–116.

[132] *OP2-Clang github repository*, `https://github.com/OP-DSL/clang-op-translator`.

[133] M. B. Giles, G. R. Mudalige, B. Spencer, C. Bertolli, and I. Reguly, “Designing op2 for gpu architectures”, *Journal of Parallel and Distributed Computing*, vol. 73, no. 11, pp. 1451–1460, 2013.

[134] *Matching the clang ast*, `https://clang.llvm.org/docs/LibASTMatchers.html`, 2018.

[135] “Refactoringtool class reference”. (), [Online]. Available: `https://clang.llvm.org/doxygen/classclang_1_1tooling_1_1RefactoringTool.html` (visited on 08/13/2018).

[136] E. Bendersky. “Ast matchers and clang refactoring tools”. (2014), [Online]. Available: `https://eli.thegreenplace.net/2014/07/29/ast-matchers-and-clang-refactoring-tools` (visited on 08/13/2018).

[137] “Replacements class reference”. (2018), [Online]. Available: `%5Curl%7Bhttps://clang.llvm.org/doxygen/classclang_1_1tooling_1_1Replacements.html%7D` (visited on 08/13/2018).

[138] M. B. Giles, G. R. Mudalige, Z. Sharif, G. Markall, and P. H. Kelly, “Performance analysis and optimization of the op2 framework on many-core architectures”, *The Computer Journal*, vol. 55, no. 2, pp. 168–180, 2011.

[139] M. Giles, G. Mudalige, and I. Reguly, “Op2 airfoil example”, 2012.

[140] *OP2 github repository*, `https://github.com/OP2/OP2-Common`.

[141] D. Dutykh, R. Poncet, and F. Dias, “The volna code for the numerical modeling of tsunami waves: Generation, propagation and inundation”, *European Journal of Mechanics - B/Fluids*, vol. 30, no. 6, pp. 598–615, 2011, Special Issue: Nearshore Hydrodynamics, ISSN: 0997-7546. DOI: `https://doi.org/10.1016/j.euromechflu.2011.05.005`. [Online]. Available: `https://www.sciencedirect.com/science/article/pii/S0997754611000574`.

[142] C. C. Douglas, S. Malhotra, and M. H. Schultz, “Parallel multigrid with adi-like smoothers in two dimensions”, *Preprint*, 1998.

[143] J. Douglas and H. H. Rachford, “On the numerical solution of heat conduction problems in two and three space variables”, *Transactions of the American mathematical Society*, vol. 82, no. 2, pp. 421–439, 1956.

[144] J. Douglas and J. E. Gunn, “A general formulation of alternating direction methods”, *Numèrische mathèmatik*, vol. 6, no. 1, pp. 428–453, 1964.

[145] T. H. Pulliam, “Implicit solution methods in computational fluid dynamics”, *Applied numerical mathematics*, vol. 2, no. 6, pp. 441–474, 1986.

[146] Y. Wang, M. Baboulin, J. Dongarra, J. Falcou, Y. Fraigneau, and O. Le Maître, “A parallel solver for incompressible fluid flows”, *Procedia Computer Science*, vol. 18, pp. 439–448, 2013, 2013 International Conference on Computational Science, ISSN: 1877-0509. DOI: `https://doi.org/10.1016/j.procs.2013.05.207`.

[147] T. Brandvik and G. Pullan, “An Accelerated 3D Navier–Stokes Solver for Flows in Turbomachines”, *Journal of Turbomachinery*, vol. 133, no. 2, Oct. 2011, 021025, ISSN: 0889-504X. DOI: `10.1115/1.4001192`. [Online]. Available: `https://doi.org/10.1115/1.4001192`.

[148] P. Bartholomew, G. Deskos, R. A. Frantz, F. N. Schuch, E. Lamballais, and S. Laizet, “Xcompact3d: An open-source framework for solving turbulence problems on a cartesian mesh”, *SoftwareX*, vol. 12, p. 100 550, 2020. DOI: `10.1016/j.softx.2020.100550`.

[149] T. Deakin, J. Price, M. Martineau, and S. McIntosh-Smith, "Gpu-stream v2. 0: Benchmarking the achievable memory bandwidth of many-core processors across diverse parallel programming models", in *International Conference on High Performance Computing*, Springer, 2016, pp. 489–507. DOI: `10.1007/978-3-319-46079-6_34`.

[150] A. Griewank and A. Walther, "Algorithm 799: Revolve: An implementation of checkpointing for the reverse or adjoint mode of computational differentiation", *ACM Transactions on Mathematical Software (TOMS)*, vol. 26, no. 1, pp. 19–45, 2000.

[151] W. H. Hundsdorfer, J. G. Verwer, and W. Hundsdorfer, *Numerical solution of time-dependent advection-diffusion-reaction equations.* Springer, 2003, vol. 33. DOI: `10.1007/978-3-662-09017-6`.

[152] B. Christianson, "Reverse accumulation and attractive fixed points", *Optimization Methods and Software*, vol. 3, no. 4, pp. 311–326, 1994.

[153] M. Wyns and J. Du Toit, "A finite volume–alternating direction implicit approach for the calibration of stochastic local volatility models", *International Journal of Computer Mathematics*, vol. 94, no. 11, pp. 2239–2267, 2017.

[154] A. Mallinson, D. A. Beckingsale, W. Gaudin, J. Herdman, J. Levesque, and S. A. Jarvis, "Cloverleaf: Preparing hydrodynamics codes for exascale", *The Cray User Group*, vol. 2013, 2013.

[155] Y. J. Lo, S. Williams, B. Van Straalen, *et al.*, "Roofline model toolkit: A practical tool for architectural and program analysis", in *High Performance Computing Systems. Performance Modeling, Benchmarking, and Simulation*, S. A. Jarvis, S. A. Wright, and S. D. Hammond, Eds., Cham: Springer International Publishing, 2015, pp. 129–148, ISBN: 978-3-319-17248-4. DOI: `10.1007/978-3-319-17248-4\_7`. [Online]. Available: `https://doi.org/10.1007/978-3-319-17248-4%5C_7`.