FACULDADE DE ENGENHARIA DA UNIVERSIDADE DO PORTO

Analysis and Parallelization of Fortran Loops Using the LARA Framework

Afonso Castro Vaz Osório

Mestrado em Engenharia Informática e Computação

Supervisor: Prof. João Bispo

Second Supervisor: Luís Miguel Sousa

July 29, 2026

Analysis and Parallelization of Fortran Loops Using the LARA Framework

Afonso Castro Vaz Osório

Mestrado em Engenharia Informática e Computação

Approved in oral examination by the committee:

President: Prof. Bruno Lima
Examiner: Prof. João Sobral
Supervisor: Prof. João Bispo

July 29, 2026

Resumo

Apesar das correntes predominantes de programação, o Fortran mantém a sua relevância no contexto de Computação de Alto Desempenho (HPC). No entanto, a exploração total de arquiteturas multi-core através de programação paralela explícita pode ser um desafio, mesmo quando se considera o uso de APIs maduras como o OpenMP. Os compiladores *source-to-source* automáticos existentes, que historicamente estão focados em C/C++, carecem frequentemente de suporte para normas modernas de OpenMP, e os compiladores poliedrais, embora capazes de obter ganhos substanciais de desempenho para alguns algoritmos, produzem frequentemente código complexo que pode obscurecer o significado do código.

Esta dissertação introduz o Metafor-AutoPar, uma biblioteca de transpilação *source-to-source* de paralelização automática para Fortran moderno. Construída por cima do *frontend* do LLVM Flang e da *framework* LARA, a nossa abordagem integra capacidades de análise de dependências que são delegadas para a *Integer Set Library* (`isl`), tipicamente utilizada em compiladores poliedrais. As dependências de dados identificadas dentro do modelo poliedral guiam a inserção segura de diretrizes de paralelização de ciclos (*loops*) de OpenMP, preservando a semântica e as geometrias dos ciclos originais do programa.

A avaliação experimental através da *suite* PolyBench e dos *NAS Parallel Benchmarks* demonstra acelerações (*speedups*) lineares quase perfeitas em vários casos. Uma análise comparativa face ao LLVM Polly revela que, embora as transformações poliedrais completas sejam ideais para álgebra linear $O(n^3)$ aritmeticamente densa, a nossa abordagem ao nível do código fonte supera-o em algoritmos de *stencil* e em *kernels* $O(n^2)$ específicos, ao manter a estrutura original do ciclo. Adicionalmente, o *transpiler* inclui um mecanismo para a geração automática de diretrizes e cláusulas `ordered/depend` do OpenMP 4.5 para extrair paralelismo do estilo "doacross". A avaliação destas diretrizes expôs um *overhead* de sincronização significativo, validando as limitações deste tipo de paralelismo *fine-grained* para algoritmos leves.

Finalmente, uma caracterização detalhada num sistema NUMA *dual-socket* forneceu perspetivas sobre como um paralelizador automático deve interagir com topologias de memória física. Demonstramos que a colocação estática de dados através da *first touch policy* não é suficiente para algoritmos com escrita intensiva e afinidades dinâmicas entre *threads* e dados. Nestes ambientes dinâmicos, as transformações ao nível do código fonte podem ser combinadas com sistemas de balanceamento dinâmico ao nível do sistema operativo para alcançar melhores resultados.

Em última análise, este trabalho demonstra que, em vários casos, uma análise transparente ao nível da AST pode fornecer uma alternativa competitiva à compilação poliedral completa para a modernização de bases de código científico legadas.

Abstract

Despite shifts in mainstream programming trends, Fortran remains highly relevant in the context of High-Performance Computing (HPC). However, fully exploiting multi-core architectures via explicit parallel programming can still be challenging, even utilizing mature APIs such as OpenMP. Existing automated source-to-source compilers, which have historically been focused mainly on C/C++, often lack support for modern OpenMP standards, and polyhedral compilers, while able to obtain substantial performance gains for some algorithms, often produce complex code that obscures the programmer's initial intent.

This dissertation introduces Metafor-AutoPar, an automatic source-to-source parallelizing transpiler library for modern Fortran. Built on top of the LLVM Flang frontend and the LARA framework, our approach integrates dependence analysis capabilities that are offloaded to the Integer Set Library (`isl`), typically used in polyhedral compilers. Data dependencies identified within the polyhedral model guide the safe insertion of OpenMP worksharing loop constructs while preserving the original program's semantics and loop geometries.

Experimental evaluation across the PolyBench suite and the NAS Parallel Benchmarks demonstrates near-perfect linear speedups in several cases. A comparative analysis against LLVM Polly reveals that, while full polyhedral transformations are ideal for arithmetically dense $O(n^3)$ linear algebra, our source-level approach outperforms it on specific stencil algorithms and $O(n^2)$ kernels by maintaining the original loop's structure. Furthermore, the transpiler features a mechanism for automatically generating OpenMP 4.5 `ordered/depend` constructs and clauses to extract "doacross"-style parallelism. Evaluating these constructs exposed significant synchronisation overhead, validating the limitations of this type of fine-grained parallelism for light workloads.

Finally, extensive profiling on a dual-socket NUMA system provided insights into how an automatic parallelisation approach should interact with physical memory topologies. We demonstrate that static data placement via the first touch policy is not sufficient for write-heavy workloads with shifting thread-to-data affinities. In these dynamic environments, source-level transformations can be paired with dynamic OS-level balancing systems to achieve better results.

Ultimately, this work shows that, in several cases, transparent, AST-level analysis can provide a competitive alternative to full polyhedral compilation for modernizing legacy scientific codebases.

UN Sustainable Development Goals

The United Nations Sustainable Development Goals (SDGs) provide a global framework to achieve a better and more sustainable future for all. It includes 17 goals to address the world's most pressing challenges, including poverty, inequality, climate change, environmental degradation, peace, and justice.

The specific Sustainable Development Goals mentioned have the following names:

SDG 9 Build resilient infrastructure, promote inclusive and sustainable industrialisation and foster innovation

SDG 12 Ensure sustainable consumption and production patterns

SDG	Target	Contribution	Performance Indicators and Metrics
9	9.4	By automatically parallelizing legacy Fortran code, the transpiler increases the computational efficiency of HPC infrastructures. Shorter execution times and proper NUMA memory topology utilisation reduce the CPU time required for scientific workloads, directly lowering the load on the infrastructure.	Execution time reduction (speedup) of parallelized benchmarks versus sequential baselines; good use of physical memory topologies.
	9.5	Metafor-AutoPar facilitates the modernisation legacy scientific codebases. By automating OpenMP injection via polyhedral data dependence analysis, reduces the need for manual, error-prone parallel programming expertise.	Number of correctly parallelized loop nests in the PolyBench and NAS Parallel Benchmark suites.
12	12.2	This work reinforces how avoiding hardware bottlenecks prevents wasted computational cycles. This promotes responsible energy consumption by ensuring hardware operates efficiently.	Mitigation of memory access costs, resulting in reduced execution times.

Acknowledgements

Time flew by, and after 22 years and a few months I'm going to stop being a student for the first time in my life (I may be back later but that's a different matter). I feel proud of what I have achieved so far, and I intend to be able to say the same about what will come next. I've never been particularly skilled with the more emotional side of writing, so I'll likely keep this brief.

Starting off with family, a massive thank you to my mother, my father, and my brother. You have all been a key part of defining who I am, and have always supported me, each in your own way. Since a certain someone thought a long time ago that it would be a good idea to have thirteen children, I won't go into details when it comes to non-immediate family. I would also like to extend honourable mentions to Troll, who was there for me since I was born, and Alph.

To Diogo, Ismael, Ivan, João, and Rodrigo, thank you for keeping in touch even though we (mostly) went our separate ways after school. To Fu, Isabel, João Miguel (the two of them), and Tomás, thank you for making the last few years of this journey more lively. Lastly on the friendship section, thank you to everyone I've met over the years at ESN Porto, for the experiences and for giving me the opportunity of participating in an organisation with such a large impact.

On the academic side, a huge thanks to my supervisor, Professor João Bispo, for always being available, for giving me the freedom to implement my ideas, and in general for all the help during the development of this work. I also want to thank Luís Sousa for the support and feedback provided. Going back a bit further, I would like to thank my maths teachers who encouraged me when I started playing around with Python on my calculator.

To everyone who has supported me at some point or another, no matter how small it may seem, thank you.

Afonso Castro Vaz Osório

Declaração de Honra

Declaro que a presente dissertação é de minha autoria e não foi previamente apresentada e avaliada nesta ou em outra instituição. As referências a outros autores (afirmações, ideias, pensamentos), assim como a utilização de trabalhos publicados respeitam escrupulosamente as regras da atribuição, e encontram-se devidamente indicadas no texto e nas referências bibliográficas, de acordo corn as normas de referenciação. Tenho consciência de que a prática de plágio ou de autoplágio constituem ilícitos académicos, conforme estabelecido no Código de Ética da U.Porto.

Declaro, ainda, que utilizei ferramentas de Inteligência Artificial para a realização de parte(s) da presente dissertação. Em concreto, o Google Gemini foi usado para a geração de *shell scripts* e para revisão da escrita. Adicionalmente, o GitHub Copilot foi usado para assistência na escrita de código. Os resultados deste uso foram objeto de cuidada verificação para deteção de erros, imprecisões, atribuições infundadas ou outras formas de enviesamento de conteúdos e argumentos, bem como de determinação de autoria.

Afonso Castro Vaz Osório

Contents

List of Figures

List of Tables

List of Acronyms

API	Application Programming Interface
AST	Abstract Syntax Tree
CG	Conjugate Gradient
CPU	Central Processing Unit
CUDA	Compute Unified Device Architecture
DSL	Domain Specific Language
HPC	High-Performance Computing
isl	Integer Set Library
JSON	JavaScript Object Notation
LLM	Large Language Model
LLVM-IR	LLVM Intermediate Representation
MPI	Message Passing Interface
NPB	NAS Parallel Benchmarks
NUMA	Non Uniform Memory Access
OS	Operating System
RAR	Read after Read
RAW	Read after Write
SIMD	Single Instruction Multiple Data
UMA	Uniform Memory Access
WAR	Write after Read
WAW	Write after Write
XML	Extensible Markup Language

Chapter 1

Introduction

1.1 Context

The proper usage of parallelisation tools in High-Performance Computing (HPC) contexts is necessary in order to fully utilize the capabilities of the hardware they employ. When considering Central Processing Unit (CPU)-based parallelisation, OpenMP is a frequently used Application Programming Interface (API) that provides support for the C/C++ and Fortran languages. OpenMP extends these languages by defining multiple directives that can be used to annotate the source code. The compiler then uses these directives as an indication of which regions of the code should be executed by multiple threads. Several options can be used to control the specifics of how the parallel region is defined, such as specifying the number of threads, or which algorithm should be used to split the iterations of a loop.

In the mainstream software development space, Fortran is seen as archaic. This reputation is due in part to the existence of numerous instances of legacy software written in outdated Fortran standards. However, the language is still evolving, with its latest international standard dating back to 2023, and due to its long-standing use in science and its performance characteristics, it is still used for many HPC applications. Historical deficiencies in the developer experience are now being actively addressed, notably the lack of a package manager and a standard library. In addition, several compilers targeting LLVM are in active development. These include Intel's proprietary ifx compiler, LFortran, and LLVM Flang, the latter two being open-source [1]. All of these are aiming for, or have already achieved, full support for Fortran 2018, and in the case of ifx, some Fortran 2023 features are available in experimental builds. All of these compilers support OpenMP directives. However, they differ in what directives are fully supported and in which OpenMP standard is followed.

Source-to-source technologies are used in different contexts in order to provide some value to the developer by transforming the source code, such as optimisation, for example, by applying different transformations to loop constructs, or backward compatibility, which is a common use case for JavaScript and other web technologies, by providing alternatives to features that may not be supported in out-of-date browsers [11].

1.2 Motivation

Multiple tools have been created to automate the parallelisation of Fortran code. However, most are limited to proprietary compilers, old Fortran standards, or both. For example, Oracle's compiler targets the 1995 standard, with extensions for some Fortran 2003 and 2008 features. And in the case of Intel, the auto-parallelisation flag appears to be deprecated, as it is no longer mentioned in the current revision of the documentation, which focuses more on the explicit insertion of OpenMP directives. For the cases in which automated parallelisation is offered as a part of the compiler's options, it tends to act as somewhat of a black box, since the specific applications of parallelisation live only in the machine code (and parallelisation reports, if available) produced by the compiler. In other words, they are not persisted in the source code.

Modern compilers, while essential to the workflow for the development of software in many programming languages, can be extremely complex pieces of software themselves, and, as such, require a lot of knowledge and an in-depth study of its code base before additions can be made by a newcomer. One of the challenges that some source-to-source compilation technologies aim to address is the high barrier of entry to compiler research. Johnson et al. [2], for example, present examples of how a source-to-source compiler (Cetus) enabled the development of research projects. A source-to-source compiler has the same working principle as a typical compiler in the sense that it is a program that is fed code and outputs new code. In a typical compiler, the output code is usually machine code meant to be executed by a computer. In a source-to-source compiler, the output is more source code, be it in the same language as the input or in a different one [11]. Source-to-source approaches to parallelisation, while not having complete control over the final code produced by the actual compiler, do not require complete knowledge of the compiler's inner workings and are able to provide meaningful transformations that can be easily accessed and understood by the programmer in the transformed source code. Additionally, as long as compiler-specific extensions are not used, source-to-source transformations are compiler agnostic.

1.3 Problem

The application of automated parallelisation of Fortran source code has not kept up with the evolution of the language and the OpenMP standard, with major parallelisation projects, such as the aforementioned Cetus, having shifted their focus to C/C++ over the years. In this work, we explore a source-to-source approach, based on modern Fortran compilers and OpenMP standards, with a focus on the direct insertion of OpenMP directives in source code and the impact that it may have on the performance of sequential programs.

1.4 Research Questions

To address the problem described in the previous section, this dissertation seeks to answer the following questions regarding source-to-source approaches to the parallelisation of modern Fortran

code:

RQ1 To what extent can modern OpenMP directives, such as `ordered` and `depend`, be leveraged by an automated tool?

RQ2 How does conservative, source-level dependence analysis with minimal code transformation compare to full polyhedral optimisation when automatically parallelising scientific Fortran workloads?

RQ3 How do Abstract Syntax Tree (AST)-level automated parallelisation strategies interact with underlying physical hardware topologies, specifically Non Uniform Memory Access (NUMA) architectures?

1.5 Approach

To address the research questions outlined above, this work details the development of Metafor-AutoPar, a source-to-source transpilation pipeline designed to automatically safely inject OpenMP directives into sequential Fortran programs.

The approach is built on top of the LARA framework, utilising the LLVM Flang frontend to parse modern Fortran source code into an AST representation. Through LARA's TypeScript-based scripting environment, our approach traverses, analyses, and manipulates the source code while maintaining its high-level structure and readability.

The transpiler employs a conservative dependence analysis module integrated with the Integer Set Library (`isl`) via its official Python bindings. We extract iteration domains, array access relations, and statement schedules from the AST and map them into Presburger sets and relations. The `isl` is then used to compute exact data flows and distance vectors. By evaluating these vectors, the transpiler classifies loop-carried dependencies to determine whether a loop can be cleanly parallelised, requires parallelisation techniques such as privatisation and reductions, or must remain sequential to strictly preserve the original program's semantics.

Once a loop nest is validated, LARA's AST manipulation APIs wrap the target region in the appropriate OpenMP `PARALLEL DO` constructs. Furthermore, we explore the extraction of fine-grained pipeline parallelism by utilising distance vectors to automatically generate OpenMP 4.5 `ordered` and `depend` clauses for select stencil algorithms.

1.6 Contributions

In search of insights into these questions, we have made the following contributions:

- Advance the development of Metafor, a LARA-based Fortran source-to-source compiler that makes use of LLVM Flang's frontend.

- Integrate polyhedral dependence analysis into the transpiler via the **Integer Set Library** (`isl`) to evaluate data dependencies and guide the safe insertion of OpenMP worksharing directives.
- Propose and implement an approach to extract pipeline "doacross"-style parallelism from select stencil algorithms by inserting OpenMP 4.5 `ordered/depend` constructs and clauses based on extracted distance vectors.
- Systematically test this work's implementation across the PolyBench suite and one example from the NAS Parallel Benchmarks (NPB) through an automated execution pipeline that validates correctness and compares execution times.
- Perform a comprehensive analysis against LLVM Polly to determine how OpenMP directive-based, source-level transformations compare against full polyhedral compilation in different algorithmic domains.
- Evaluate the interaction between automated source-level transformations and Non Uniform Memory Access (NUMA) topologies, demonstrating the importance of initial static data placement paired with dynamic system-level balancing systems.

1.7 Document Structure

The rest of this document is structured as follows. **Chapter 2** presents an overview of the key concepts involved in the dissertation. **Chapter 3** delves into existing work on program optimisation through code transformations. **Chapter 4** discusses dependence analysis in the context of the polyhedral model and our implementation of automatic OpenMP directive insertion in Fortran programs. **Chapter 5** presents the main software components of the Metafor transpiler and some of the work that was developed to extend its support for the Fortran language. **Chapter 6** explains our evaluation methodology and presents the results obtained in this work, both in isolation and compared to a full polyhedral approach. Lastly, **Chapter 7** provides a conclusion and outlines the limitations of our implementation and how they could be addressed in future work.

Chapter 2

Background

This chapter introduces the key concepts involved in the dissertation.

2.1 Fortran and Flang

Fortran is a high-level imperative programming language. Its high performance on mathematical operations and mature specification have earned Fortran a spot among the most used programming languages for scientific and HPC software. Since the language first surfaced in 1957, much like any other programming language that has been in use for several years, it has gone through a number of revisions. The first revision to be published as an ISO standard was Fortran 90. When the term "Modern Fortran" is used, it is typically in relation to Fortran 2003 or a newer standard. The latest significant revision to the language is Fortran 2018. Fortran 2023, the most recent standard, is mainly focused on small additions and refinements.

Among the existing Fortran compilers, we will highlight LLVM Flang, which is currently under active development. It is an official LLVM subproject, which aims to provide a Fortran frontend to the LLVM Intermediate Representation (LLVM-IR) and its associated tools and features, similarly to what the Clang compiler provides for C. It is supported by several hardware vendors and research laboratories that contribute to its development. Although support for later standards is planned, the project is currently targeting Fortran 2018.

2.2 Parallel Programming Models

When developing a parallel application, there are mainly two approaches that can be followed, depending on the needs of the application and the available hardware. These approaches are not mutually exclusive.

Shared-memory parallelism is intended for use in an environment in which there are multiple processing units that share their main memory. This is often achieved through the use of OpenMP [40], an open standard parallelisation API for C/C++ and Fortran with a wide adoption among each language's main compilers. One of the main strengths of OpenMP is its directive-based approach

to parallelisation: OpenMP code is not part of the program's source code itself in a sense; rather, it annotates the source code, providing the compiler with directives that guide the generation of parallel machine code. Of course, sequential code cannot always be parallelised directly with no structural changes, but this approach ensures that much of the boilerplate code related to shared-memory parallelism, such as the creation of threads, is abstracted away from the source code, and allows for the incremental parallelisation of existing sequential code.

Due to their inherent potential for parallelisation, this type of parallelism is mostly applied to loop constructs. This is the focus of OpenMP's simplest and most well-known directives, such as `omp parallel for` (used for C/C++ loops), which splits the iterations of the loop among the available threads. Since its inception, the capabilities exposed by OpenMP directives have vastly expanded, with the latest revision, OpenMP 6.0, having released in 2024. They range from simple parallel programming concepts, such as defining a critical region that only one thread must execute at a time, to complex features, such as a task model which allows for the dynamic creation of tasks (a region of code that is to be executed by any thread) which are passed to a scheduler that determines at runtime the allocation of tasks to threads.

The other main parallel programming paradigm, which will not be the focus of this dissertation, is **distributed memory parallelism**. Similarly to shared-memory parallelism, there exist several processing units. However, not all of them share their main memory. The topology of the system, that is, the mapping of processing units to memory units and the way in which memory units are connected, can vary greatly from case to case and must be taken into account in the development of distributed memory software if optimisation is a concern. The Message Passing Interface (MPI) [37] is a sort of catch-all term that covers many implementations of distributed memory parallelism standards. The Fortran standard from 2008 onwards specifies its own distributed memory parallelism standard, Coarray Fortran [39].

2.3 Source-to-Source Compilation

In most computer science contexts, the term "compiler" refers to software that takes source code written in a human-readable programming language and produces a sequence of instructions to be executed, be it directly by the CPU or by some mechanism such as a virtual machine (e.g., JVM [30]). The input source code is typically processed in a sort of pipeline that involves several intermediate representations and processing stages. This can contribute greatly to the complexity of the compiler's own code. A source-to-source compiler maintains the basic principle of receiving code and producing new code, but it diverges from the usual concept of a compiler in the sense that the output is also in a human-readable programming language. This can be used in order to apply transformations to the source code, or even to translate it to a different programming language (transpilation).

Source-to-source compilers made for HPC contexts and general program optimisation usually focus on the most computationally heavy regions of programs (loops). This can take the form of applying parallelism to sequential code, improving existing parallel code, or general algorithmic

optimisations. In many cases, the programmer can control the areas of the source code in which the source-to-source compiler should act through directives/annotations that are specific to it.

2.4 Data Dependencies

Whether at the source level or at a lower representation of the code, the concept of **data dependencies** is frequently cited when it comes to optimisation. The specifics may differ, but it is often useful to know how different accesses to the same memory location are related. In the case of loop constructs, broadly speaking, dependency analysis is used to determine if all of the loop's iterations can be executed independently of one another.

2.4.1 Types of Dependence

Data dependencies are classified into one of four categories, according to the order in which different types of accesses occur:

- **True dependence:** A value is dependent on the result of a previous operation (**Read after Write (RAW)**)
- **Anti dependence:** An operation requires a value that will be subsequently updated (**Write after Read (WAR)**)
- **Output dependence:** A value is updated several times (**Write after Write (WAW)**)
- **Input dependence:** A value is used several times (**Read after Read (RAR)**)

For the purposes of this work, input dependencies will not be relevant.

```
! Source (Write)
A(i) = 5
! Sink (Read)
B(i) = A(i)
```

```
! Source (Read)
B(i) = A(i)
! Sink (Write)
A(i) = 10
```

```
! Source (Write)
A(i) = 1
! Sink (Write)
A(i) = 2
```

Figure 2.1: Examples of RAW, WAR and WAW dependencies.

All dependencies are defined by a **sink** and a **source**. In the case of a Read-after-Write dependence, for example, the **source** is the statement in which the value is written, and the **sink** is the one where it is read (as shown in the leftmost section of Figure 2.1).

2.4.2 Dependence Analysis

Analysing the data dependencies present in a program can be used to determine which statements/instructions can be reordered or executed in parallel while maintaining the original program's behaviour.

The application of dependence analysis to loop constructs introduces some challenges, due to the possibility of each statement being executed several times. Thus, in addition to data dependencies that arise from the order of operations, data dependencies across loop iterations must also be considered. As such, a dependence is considered *loop-independent* if it is limited to a single iteration, and *loop-carried* otherwise.

2.4.3 Presburger Arithmetic

The Presburger language is a first-order language over the domain of integers that includes addition, equality, and inequalities but excludes the multiplication of variables [59]. This means that the language is quite restricted with regards to what it can express, but that is also the source of one of its defining properties. Unlike broader arithmetic systems that include variable multiplications, such as Peano arithmetic [33], any statement formulated in Presburger arithmetic can be definitively proven true or false (in other words, it is **decidable**).

Its decidability makes it a suitable choice for the analysis of loop nests, in which it can verify the legality of complex code transformations. Several optimising compilers (polyhedral or not) rely on solvers that use Presburger arithmetic as their mathematical foundation, such as the Integer Set Library [58] or the Omega library [27].

2.4.4 Distance Vectors

For a given data dependence, the **distance vector** is defined as a vector with as many dimensions as those of the iteration space, in which each element of the vector is the distance between the sink and the source's iteration number in the appropriate dimension [3].

For example, in a two-dimensional loop nest with iterators `i` and `j`, let's say that there is a dependence from iteration `i=n-1, j=n+2` to iteration `i=n, j=n`. The distance vector for this dependence would be `[1,-2]`. Using positive values for distances that come from previous iterations of a given dimension was an arbitrary decision that derived from the specific way in which we obtained them in our implementation.

2.4.5 Polyhedral Model

The **polyhedral model** is a mathematical framework for program optimisation [59]. It is used to reason about operations that occur in quantities that may be too large to be explicitly enumerated, and are thus described in a compact way through mathematical objects such as polyhedra and Presburger formulas.

Given that each of their statements may be executed multiple times, nested loop structures are the typical subject of polyhedral optimisation approaches. In this model, each statement/operation would translate to several "dynamic execution instances", as many as there are iterations in the loop. This representation requires affine loop bounds to make use of Presburger arithmetic's decidability. By expressing the loop structure as a mathematical object, abstracted from the original source code, polyhedral compilers are able to perform complex transformations, such as loop

fission, tiling and automatic parallelisation while guaranteeing that the semantics of the original loop structure are unchanged.

While the full scope of polyhedral compilation falls outside the methodology of this work, the manipulation of integer sets and relations as it exists in the polyhedral model is highly relevant for dependence analysis.

2.5 Non-Uniform Memory Access

Non Uniform Memory Access (NUMA) is a parallel memory design in which each processor is associated with a memory region close to it. Accesses to this memory will be faster than accesses to the memory of other processors. In practice, in systems that support NUMA, memory is segregated into two or more NUMA domains that are shared by multiple processors.

The data placement policy dictates where data is physically placed. To develop performant parallel software for a NUMA system, it must be taken into consideration. On Linux and other operating systems, the default is the **first touch policy**: data pages are allocated in the NUMA domain closest to the processor which first accessed it.

Making good use of the first touch policy is often quite simple. If the data initialisation step is parallelised and its access pattern matches that of the algorithm itself, each part of the data will be physically placed near the processor that needs it. However, data access patterns may not match up, especially in cases in which the outermost loop of the algorithm is sequential. The manual adjustment of benchmark implementations to improve performance on NUMA systems has been studied in the context of the NPB, for example, by McCurdy et al. [32].

Chapter 3

State of the Art

In this chapter, we will discuss some literature related to source-to-source program optimisation, specifying the strengths and pitfalls of existing work that may be relevant during the development of our approach.

3.1 Related Work

3.1.1 Cetus/iCetus

Bhosale et al. [9] recently discussed the impact of the different analysis passes and transformations provided in **Cetus**. Cetus [14] is a source-to-source C compiler, with the primary objective of being a parallelising compiler. C code is parsed by a custom grammar defined in ANTLR, and the transformations are implemented in Java, by manipulating an IR that aims to closely match a C compiler's AST. C++ is not supported since, according to the original authors, parsing C++ with an ANTLR grammar would be infeasible. One limitation of Cetus is that it analyses code at a local level, that is, the source code of functions called in loops will not be taken into account unless the user allows Cetus to inline them.

The most important transformations were identified by measuring the impact that disabling each feature had on a full optimisation pass of the benchmark code. This is not an exhaustive way to test the interactions between every feature, but it can be sufficient to determine which analyses have the biggest individual "impact". Scalar and array privatisation was found to be the most important analysis. An additional analysis pass is discussed, which is named "subscripted subscript analysis". This pass enables parallelisation for loops whose data dependencies are dependant on the properties of the elements of an array (namely, monotonicity), for cases in which these properties can be proven through static code analysis.

A web-based interactive frontend for Cetus, iCetus [7] is also presented. Although this side of Cetus as a whole is not of particular relevance for this work, we can highlight that one of the design principles behind this interactive tool was providing the user with output that explains why Cetus chose to (or not to) apply transformations to specific regions of the code.

3.1.2 TC

Skotnicki [54] presents **TC**, an optimising source-to-source compiler for loop nests. The approach is based on the **polyhedral compilation model**, as well as other well-known compilation tools, such as Pluto or Polly [20]. As previously discussed, this model abstracts the loop to a set of statement instances, one for each value that the loop iterators can take. Based on identified data dependencies in the loop, two main transformations are carried out. Tiling is responsible for locality enhancement by grouping the statement instances, and scheduling analyses the dependencies that remain across the identified tiles in order to identify an execution order that enables parallel processing.

TC extends the polyhedral compilation toolchain by introducing a data dependence analysis based on the transitive closure relation. By applying a transitive closure to the identified dependencies, it is possible to identify all statement instances that rely on an "earlier" instance. This is used for iteration space slicing, the technique used in TC to extract parallelism from the set of statement instances.

In the results presented, TC consistently led to higher speedups than Pluto, although the extent of the improvement varies. The main drawback is compilation time, due to the computation of the transitive closure relation. Moreover, in the case of infinite graphs, this computation is an undecidable problem, which means the exact, optimal result is not always attainable.

3.1.3 Integration of Polyhedral and AST Transformations

Shirako et al. [51] propose an optimisation flow that seeks to bridge the gap between polyhedral and AST-based transformations, with the goal of avoiding a limitation of the polyhedral mode: it can generate highly complex code that can be detrimental to backend optimisations, such as vectorisation. The authors developed a multi-stage framework that begins by using the polyhedral model to perform coarse-grained transformations such as tiling. Further transformations are performed by an AST-based approach that handles fine-grained mechanisms such as vectorisation, unrolling, and the insertion of thread-level parallelism directives, including pipeline/"doacross" parallelism.

This work demonstrates that the rigor of polyhedral compilation and traditional source-to-source AST-based transformations are not mutually exclusive, but rather complementary.

Arabnejad et al. [5] explored a similar avenue for parallelisation using Clava [10], in the sense that some of the approach's capabilities are offloaded to a solver (in this case, the Omega Library [27]).

3.1.4 ComPar

Mosseri et al. [36] present **ComPar**, a source-to-source parallelisation "multi-compiler". ComPar's approach to parallelisation did not involve developing new source code analysis tools; instead, the focus was on combining existing source-to-source compilers, with the goal of the resulting compilation being able to leverage the strengths of each underlying source-to-source compiler.

The only transformation that ComPar itself applies to the input source code is some instrumentation around loop constructs in order to facilitate identification and performance measurements. The novelty is in the way in which source-to-source compilers are used afterwards: several versions of the transformed source code are created, using different combinations of the compilers and of user-defined parameters. The performance of each permutation is stored in a database, and the best one is chosen for the final output. Additionally, ComPar can check for correctness by comparing the output of each permutation to that of the initial code.

As could be expected, the time that ComPar needs in order to provide its final output is longer than pure static code analysis tools, since it requires the program to be executed several times. The concrete overhead this introduces is dependent on the original program and on the number of combinations that must be generated. The results presented show how the use of ComPar compares to each underlying source-to-source compiler individually. ComPar consistently achieved better speedups than each of them, or at least a value comparable to the best speedup among the individual compilers.

3.1.5 Analysis and Pitfalls of Source-to-source Parallelisation Compilers

Harel et al. [22] performed an evaluation of the source-to-source compilers used in ComPar. Strengths and weaknesses were identified for each compiler. Their strengths are mostly associated with the breadth of code that they are able to parallelise and the explainability of their output. Weaknesses, besides those which are complementary to other compilers' strengths, are more varied, being identified in:

- The amount of user intervention required by the compiler (AutoPar [28] uses annotation files);
- Excessive code transformation (Par4All [4] may change the code structure, and Cetus inserts its own specific pragmas);
- Lack of correctness and portability (AutoPar may insert incorrect OpenMP directives in specific cases, and Cetus may insert non-standard reduction clauses)

The authors propose two approaches for future source-to-source parallelisation work. One of them is to move past the outdated OpenMP standard used by all three compilers (OpenMP 2.5, released in 2005). Later standards have introduced a plethora of directives and clauses, such as `#pragma omp simd`, that these tools could make use of. The other outlines the approach that was later applied in ComPar.

3.1.6 Source-to-source Vectorisation

Flynn et al. [17] applied explicit source-to-source transformations in the context of vectorisation. A vectorised program is one that uses Single Instruction Multiple Data (SIMD) instructions in order to simultaneously perform the same operation on multiple values in a single CPU core.

Although this is not the thread-level parallelism that has been the focus of most discussed works, it is still a form of optimisation via (instruction-level) parallelism and can be used in tandem with thread-level parallelism.

The authors list several methods of vectorising programs, in decreasing order of the programming effort required: manual vectorisation with hardware-specific intrinsics; explicit vectorisation directives (with OpenMP, for example); automatic vectorisation by the compiler. They argue that they each have their pitfalls, and so they sought to bridge the gap between manual and automatic vectorisation by using source-to-source transformations to automatically insert platform-appropriate intrinsic calls in the source code. The intermediate representation used to accomplish this was built on top of the ROSE framework [46].

The results showed that the performance of the programs compiled with this source-to-source approach was consistently on par with, and sometimes better than, when they were vectorised through OpenMP directives or through the compiler itself. This provides the programmer with the convenience of automatic compiler transformations without sacrificing the potential of fine-tuned, manual optimisation.

Henriques et al. [23] also explore source-to-source vectorisation, with the specific purpose of inlining RISC-V Unlimited Vector Extension assembly instructions in C/C++ source code. The transformation steps are presented in detail, and show some concepts that may be applicable to broader source-to-source loop optimisations, such as "lowering" the code to a representation that facilitates further analysis/transformation (for example, expanding conditional ternary operations into if/else constructs).

3.1.7 PragFormer/POSTER

Compared with what has been discussed so far, Harel et al. [21] adopt a very different methodology in **PragFormer**, which is based on transformer models and Natural Language Processing techniques, instead of deterministic static code analysis. This was motivated by the pitfalls identified by the authors in existing source-to-source compilers. Some of them could be solved with further refinement (use of more OpenMP directives, merging parallel regions), while others, such as the determination of the side effects of a function, are considerably more difficult in the context of static analysis.

The dataset used to train PragFormer consists of snippets of code, which are labeled according to their suitability for OpenMP parallelisation. Each item contains the original code, the OpenMP directive that was used (if applicable), and an AST representation of the OpenMP directive. When evaluated over the dataset created to test the model, PragFormer performed better than ComPar. In benchmarks such as PolyBench, PragFormer procuded, at worst, slightly better results than ComPar. Naturally, due to the nature of this approach, there are cases where the model produces incorrect code, though this isn't shown to be a significant hindrance. One of the possible reasons cited for ComPar's lacking performance is "the lack of association of functions, macros, and structure definitions and implementations in the code segments". This is inconclusive as to whether the

performance of traditional static tools could have increased if more of the code was provided or not.

3.1.8 OptiTrust

Bertholon et al. [8] present **OptiTrust**, a parallelisation/optimisation tool with a focus on correctness. For OptiTrust to optimise a segment of code, the source code itself is not enough: the programmer must annotate the code with **contracts**, a notion that is carried over from verification-aware programming languages. These annotations describe which regions of memory are read/-modified by functions or loops.

The transformations themselves aren't automatically defined either. Instead, they must be specified in an OCaml script. The tool then applies the transformations one at a time, verifying that the contracts still hold after each step via static resource analysis. Moreover, an HTML report is generated that shows the changes applied by each transformation. Despite OptiTrust's usage not being as "plug-and-play" as other approaches, it seeks to provide strong correctness guarantees with regards to the semantics of the input code.

A similar approach is described by Liu et al. [29]. However, it is limited to the highly specialised domain of tensor programs.

3.1.9 Other Works

The sources from the literature review that were not discussed at length are not directly relevant to the definition of our approach in the sense that they do not describe an approach to automatic source code parallelisation. However, they give us an overview of the current state of research in source-to-source transformations.

Several of the works can be classified into one of a few groups. Firstly, there are several which describe transformations that generate parallel code from code that is already parallelised, by adding additional parallel programming models ([55], [56]) or by replacing the one which was in use ([48], [19], [50], [13]). The goal can be to achieve better performance, to refactor code that is considered to be outdated, or both. Fridman et al. [19] shows that OpenMP-based GPU offloading offers comparable performance to outdated OpenACC implementations, while providing noticeable performance gains when compared to CPU-based usage of OpenMP. However, the lack of performance improvements is not necessarily a indication of failure, as was discussed in the case of source-to-source vectorisation, since these tools can provide the programmer with clear and up-to-date code that still allows for further fine-tuned optimisation. A more drastic version of this sort of transformation can be found in full transpilers, which aim to produce code equivalent to its input in a different programming language altogether [18].

Others present domain-specific languages that enable the programmer to describe structural patterns that are common to the domain in question, which are then processed by a source-to-source compiler in order to produce valid source code. This is the case of Faé et al. [16] and Hoffmann et al.'s [24] works, which are both related to SPar, a Domain Specific Language (DSL)

for stream processing programs. Here, there usually isn't a big focus on the automatic identification of transformations to apply. Cordasco et al. [12] and Schmitz et al. [49] have also developed DSL-based approaches. The transformations proposed by Mishra et al. [35], despite not making use of a DSL directly, address the domain-specific problem of data movement in GPU-accelerated programs.

Additionally, source-to-source research doesn't need to be focused specifically on performance improvements. Matos et al. [31] define a subset of C that facilitates static analysis and methods that reduce generic C code to that subset. Pinto et al.'s Pegasus [42] facilitates profiling, and Vanderbauwhede's approach [57] aims to automatically update FORTRAN 77 procedures with typing-related features that were introduced in Fortran 90.

Lastly, there is a lack of consensus regarding the applicability of source-to-source tools in HPC contexts. While some of the common complaints are inherent to the nature of source-to-source compilers (for example, it being another tool that has to be added to the programmer's workflow), there is research into alleviating other concerns. One of the problems identified by Milewicz et al. [34] is "S2S approaches lead to complex and fragile workflows", and a proposed solution is "Provide lowering transformations that normalise the code structure, reducing complexity and improving maintainability". This is precisely the sort of solution that works like Matos et al.'s [31] aim to provide. Another pitfall pertains to the difficulty of parsing source code. Some source-to-source compilers, such as Clava [10], address this by making use of the actual compiler's (in Clava's case, clang) parsing step to obtain an AST representation of the source code. The Polyhedral Extraction Tool [60], while not a full source-to-source compiler, makes similar use of clang's libraries. On the other hand, there are those who argue that there are tangible opportunities for source-to-source compilers moving forward, such as Bispo et al. [11], and there are documented cases of source-to-source transformations being applied to HPC contexts with good results. Examples include Siso et al.'s [53] work in optimising weather and climate applications, which involved the use of a DSL, and Palkowski et al.'s [41] use of a polyhedral source-to-source compiler to a protein folding algorithm.

3.2 Comparative Analysis

There have been several approaches to source-to-source parallelisation (or, more broadly, optimisation), some of which are compared in Table 3.1. Besides recent machine learning based approaches, they can be mainly grouped into AST-based approaches, those that rely on the polyhedral model, and hybrid frameworks. While pure polyhedral compilers can certainly be considered the state of the art with regards to performance gains, they are nevertheless restricted by the constraints of the model itself. Typically, this means that they can only be applied to loops that can be represented by affine expressions, whose body consists of array accesses that are themselves represented by affine expressions. A generic AST-based approach might not be able to achieve the same results when applied to code that is suitable for the polyhedral model, but it may be applicable to a wider range of code structures.

Table 3.1: Comparison of Related Work Approaches

Approach	**Underlying Representation**	**Optimisation Target**	**Level of Automation**	**Correctness**
Cetus / iCetus	AST	OpenMP + Loop Transformations	High	Standard
TC	Polyhedral Model	Loop tiling/scheduling	High	Exact (implied by the correctness of the model)
Poly+AST	Polyhedral + AST	OpenMP + Loop Transformations	High	Strong (partially dependent on the polyhedral model)
ComPar	n/a	Generic	Medium	Runtime
S2S Vectorisation	AST-based custom IR	SIMD	Medium	Standard
PragFormer	Transformers	OpenMP	High	Probabilistic
OptiTrust	AST	Generic	Low	Strong

An additional differentiating factor that must be considered is the amount of user intervention required for the use of the compiler. This is indirectly tied to the correctness guarantees provided by it, since their absence implies the need for the user to put some effort into verifying the transformed source code. In the AST-based approaches discussed previously, we can see that a focus on correctness may require the loss of some automation capabilities. In the case of OptiTrust, this can be seen in the need to manually annotate the source code with contracts that describe its behaviour. In other approaches, such as Cetus, while the transformations were tested, there is no method in place that provides a formal guarantee that the semantics of the code are unchanged (hence the use of "Standard" in Table 3.1). Note that all of the approaches listed in the table support C/C++ programs. Approaches based on the polyhedral model have an advantage in this regard: since the polyhedral model is a well-defined mathematical concept, with clear rules on how its representation should be extracted from source code, the correctness of the transformed code is implied by the correctness of the model. Machine learning-based transformations, on the other hand, are unable to provide any guarantees without some additional verification process, such as the one found in ComPar.

Additionally, the design choices taken in these approaches have an effect on the time it takes for the compiler to perform its optimisations. This metric is not always presented, and can in some cases be quite unpredictable. In the case of TC, the application of an analysis pass based on an abstract mathematical concept (the transitive closure of an infinite graph) leads to unpredictable runtimes, depending on the specific conditions of the original source code. This can also be caused by the introduction of non-static components, as is the case of ComPar, which compares several

possible transformations of the code by executing each one in a profiling step.

3.3 Summary

This analysis allows us to draw some conclusions with regards to our planned approach of using generic AST-based transformations in order to parallelise Fortran source code, mainly through the insertion of OpenMP directives. There is no expectation that it will reach the performance that a polyhedral-based approach can provide, when it is applied to code that it is suited to. This is not without its upsides, as previously discussed. Moreover, the usage of our planned approach does not invalidate that of the polyhedral model. Firstly, it can serve as a base for the future implementation of polyhedral transformations. This could even be achieved by offloading some of the work to existing libraries, which is something that can be observed in TC, with the use of, for example, the Polyhedral Extraction Tool [60]. We already do this in part by using the integer Set Library for dependence analysis. Secondly, there are polyhedral transformations that can be enabled in the backend of LLVM compilers, which are implemented in Polly [20], against which we performed a comparative analysis of our approach.

Concerning correctness guarantees, the tools that have been implemented in the past with the LARA framework do not support any formal verification for the transformations they apply, and implementing such a system in the context of this work could prove to be a significant challenge. However, this does not mean that no verification can exist: if some care is put into validating a set of simple transformations, this could be extended to compositions of those transformations, and, once again, serve as a base for future research.

Of the presented approaches, ours is closest to Cetus/iCetus, in the sense that it is based on AST transformations and is focused on OpenMP directives (though Cetus does have support for a few code transformations, such as loop interchange). Works related to Cetus discuss the relative importance of each of its features, which can provide insights into what to prioritise in the early stages of a parallelising source-to-source compiler. Cetus (and, to a greater degree, iCetus) also provides the programmer with feedback regarding the transformations that were applied to the code, which can aid in the analysis of the transformed code in the absence of a formal verification. Additionally, as with Shirako et al.'s [51] hybrid framework, we integrated the polyhedral model into our approach, though its involvement is limited to data dependence analysis.

Chapter 4

Dependence Analysis

This chapter presents an overview of how loop dependence analysis was performed on Fortran code. This involves extracting a suitable representation for a dependence analysis framework from the source code, obtaining information on the data dependencies that exist in the loop nest, and determining how to parallelize the loop (if possible) while maintaining the original program's behaviour.

4.1 Representation of a Loop Nest

Before performing the dependency analysis itself, we must express the information contained in a loop nest as Presburger sets and relations [59] which will be processed by a solver.

There are multiple elements that must be specified for each statement. We will present them in a way that is already tailored to the solver we chose (the Integer Set Library) [58]. It is possible that the specifics of how the information is organised could have been different had we used a different solver, but the concepts are nevertheless broadly applicable to polyhedral compilation. In particular, we will discuss:

- Iteration Domains
- Schedules
- Access Relations
- Contexts

4.1.1 Iteration Domain

A statement's **iteration domain** specifies the values that each iterator will assume during the loop nest's execution. For example, for a loop that begins with `DO i = 1, 10`, the loop variable `i` will assume all integer values from 1 to 10. An iteration domain for a simple two-dimensional loop nest is shown in Figure 4.1. The domain string is not restricted to loop variable ranges; any

condition that can be expressed in Presburger arithmetic can be added. For example, this can be useful in case the statement's execution is restricted by some conditional construct.

$$\mathscr{D}_{S1} = \{S1[i,j] : 1 \leq i \leq 16 \wedge 1 \leq j \leq 32\}$$

Figure 4.1: Example of an iteration domain in a two-dimensional loop nest.

4.1.2 Schedule

A statement's **schedule** contains an array that is used to express the order in which it is executed in relation to the other statements of the loop nest. For a loop with a single statement, a schedule such as `[i]` would suffice for said statement. Constant values can be inserted into the schedule to specify an ordering across statements of the same loop.

In Figure 4.2 and subsequent examples, each statement will be annotated with a comment referring to its name, such as S1 for the first statement. We are aware that the CPU and the compiler itself may perform instruction reordering, potentially placing instructions from statement S2 before statement S1, but since this is performed at a lower level than our analysis and respects data dependencies, it did not impact our approach.

```
DO i = 1, 10
  A(i) = B(i) * 2 !S1
  x = A(i) !S2
END DO
```

$$\mathscr{S}_{S1} = \{S1[i] \rightarrow [i,0]\}\ \mathscr{S}_{S2} = \{S2[i] \rightarrow [i,1]\}$$

Figure 4.2: Example of a valid schedule for a simple loop nest.

4.1.3 Access Relations

An **access relation** expresses the fact that a certain index of an array is accessed on a given iteration. An element of the relation by itself does not provide any information regarding whether the access is a read or write. That distinction is created by storing them in separate data structures. One such relation is shown in Figure 4.3.

```
DO i = 1, 10
  A(i) = 2 !S1
END DO
```

$$\{S1[i] \rightarrow A[i]\}$$

Figure 4.3: Example of an access relation in a trivial loop nest.

4.1.4 Context

The **context** is used to apply restrictions to the values that a certain variable can take. For example, if an integer variable `n` is used in a loop nest, and it is known that the value of `n` is always 5, this can be added to the global context (Figure 4.4). This can be useful for the dependence analysis in the case that these values are used as part of indexing expressions, since it limits the range of indexes to which the expression may be referring.

The context is not limited to assigning a single integer value to a variable. It can also be used to express the range of values it can take ($n > 0$) or even to establish connections between variables ($n > m$).

```
DO i = 1, 10
  A(i + n) = 2 !S1
END DO
```

$$[n] \rightarrow \{ : n = 5 \}$$

Figure 4.4: Example of a context string in a trivial loop nest if it is known that `n` is 5.

4.2 The Integer Set Library

Dependence analysis can have a varying degree of complexity depending on the type of variable analysed. For scalar variables (integers, floating point numbers, ...), the analysis is fairly straightforward since it is clear what value each access corresponds to (in this case, always the same one), ignoring for now language-specific concerns such as aliasing. However, when considering accesses to array variables, we have to consider that each access to the variable may correspond to distinct values. This will depend on the expression used to index the array, which can depend on other variables, such as loop iterators.

The Integer Set Library (`isl`) [58] is described as being capable of "manipulating sets and relations of integer points bounded by linear constraints". "Linear constraints" refers to Presburger arithmetic. In the context of loop dependence analysis, we will deal with **affine expressions in which the loop iterators are the variables**.

Multiple parallelisation tools, such as TC ([54]) and Polly ([20]), employ the Integer Set Library as their Presburger arithmetic solver. It already includes functions that streamline the process of dependence analysis in particular.

4.3 Parallelisation Techniques

Loop-carried dependencies are the main obstacle when determining whether a given loop nest can be parallelised, as their presence can enforce an ordering over the iterations. If certain conditions are met, there are techniques that can eliminate these dependencies while maintaining the original program's behaviour.

4.3.1 Scalar Privatisation

In the context of parallel execution, **privatising** a variable gives each thread an independent copy of the variable. Consider the loop nest presented in Figure 4.5. There is a loop-carried anti-dependence on scalar variable `temp` from statement S2 to S1 (in the next iteration). This loop nest cannot be parallelised as-is, since this dependence would lead to a race condition on the variable. If every thread were to have its own copy of `temp`, however, the dependence is entirely eliminated, as references to `temp` in different iterations would effectively refer to completely distinct variables.

```
DO i = 1, n
   temp = b(i) !S1
   c(i) = temp + 1.0 !S2
END DO
```

Figure 4.5: Simple loop nest requiring privatisation of the variable `temp`.

A scalar variable is said to be privatisable with respect to a loop nest if and only if "every path from the beginning of the loop body to a use of [the variable] within that body must pass through a definition of [the variable] before reaching that use" [3]. From a dependence analysis point of view, we can express this as **scalar variables that are not subject to loop-carried flow dependencies**. Scalar privatisation cannot eliminate these dependencies, since thread-local copies are incompatible with a scenario in which a value defined in one iteration must be carried into another.

4.3.2 Reduction Operations

A **reduction** is a common computational pattern in which a set of values is aggregated into a single variable using an associative and commutative operator, such as addition, multiplication, or a maximum/minimum function. Figure 4.6 illustrates a standard summation reduction over an array.

```
total = 0.0
DO i = 1, n
   total = total + A(i) !S1
END DO
```

Figure 4.6: A simple loop nest performing a sum reduction.

From a dependence analysis perspective, statement S1 generates a cycle of loop-carried dependencies on the scalar variable `sum`. Specifically, it creates a loop-carried flow dependence (the value computed in iteration i is read in $i+1$), an anti-dependence (iteration $i+1$ overwrites

the value read in *i*), and an output dependence (each iteration overwrites the previous result) [3]. Privatisation cannot fully eliminate this cycle, as it cannot safely resolve flow dependencies.

However, because the reduction operator is mathematically associative and commutative, any ordering of the iterations will produce a correct final result (in the absence of other problematic dependencies). Parallel execution models exploit this property to safely bypass the dependence. Each thread is assigned a local accumulator, in which it will accumulate the partial results from its assigned iterations. Upon completion of the loop, the threads are synchronised and their results are merged into a global result.

4.3.3 "Doacross" Loops

The `ordered` clause for worksharing loop constructs and the `ordered` construct have been a part of the OpenMP specification since OpenMP 1.0. Their purpose was to allow regions of code (delimited by an `ordered` construct) of a parallel loop to be executed in iteration order while the rest of the loop's code was executed in parallel. In the OpenMP 4.5 specification, new functionality was added to the `ordered` construct by means of the `depend` clause, which provides a more powerful way to express inter-iteration dependencies.

Figure 4.7 shows an example of how these clauses can be used to parallelise a loop in which iteration depends on its top-right "neighbour". When a `depend` clause of the **sink** type is encountered, the thread will stop until the dependencies specified by the sinks have been met; when one of the **source** type is encountered, the current iteration is marked as complete. In this particular example, the threads will proceed along the grid in a staggered fashion, lagging behind each other by one element each (hence the other name given to this kind of parallelism, **pipeline parallelism**).

```
1  !$OMP PARALLEL DO ORDERED(2) PRIVATE(i, j)
2  DO i = 2, N
3    DO j = 1, M - 1
4
5      ! Wait for the top-right neighbor (i-1, j+1) to finish
6      !$OMP ORDERED DEPEND(SINK: i-1, j+1)
7
8      A(j, i) = A(j, i) + A(j+1, i-1) ! S1
9
10     ! Signal that this iteration (i, j) is complete
11     !$OMP ORDERED DEPEND(SOURCE)
12
13   END DO
14 END DO
15 !$OMP END PARALLEL DO
```

Figure 4.7: Example of stencil loop parallelisation using OpenMP doacross loops.

This is not a silver bullet for the parallelisation of such programs. As one would expect, the mechanisms required for the enforcement of these dependencies introduce additional synchronisation work that must be performed by the OpenMP runtime and thus more overhead. Additionally,

not all threads will be actively working on the computation at all times. At the start of the loop nest shown in Figure 4.7, the first threads must complete their iterations before the rest can clear their `SINK` barriers. This introduces a stalled period while waiting for the initial pipeline to propagate across the iteration space. A similar scenario occurs at the termination boundaries of the loop domains. If the working set is not large enough and the computations performed in the loop are not sufficiently intensive, this pipelining overhead may overshadow the computational throughput of the threads.

Additionally, note that unlike the previous two techniques, this does not eliminate dependencies; it enforces an ordering that respects the dependencies.

4.4 Extracting Information from Fortran code

4.4.1 Benefits of Fortran for Dependence Analysis

In essence, dependence analysis is language-agnostic. However, there are a few characteristics of Fortran that facilitate the development of automated dependence analysis.

4.4.1.1 Lack of Aliasing

In C/C++, variables may be freely aliased, which means two names can refer to the same memory region. This is problematic for dependence analysis, because an access to `b[i]` could refer to the same value that would be accessed with `a[i+5]` without it being clear in the source code. Thus, a source-to-source C/C++ dependence analysis approach must employ some form of aliasing analysis in order to be robust, and in cases in which static aliasing analysis isn't sufficient, the approach must make pessimistic assumptions in order to maintain correctness guarantees.

Fortran does not have this issue to the same extent. Unless specific modifiers are attached to a variable's declaration, it cannot be aliased with another name. Therefore, we have not implemented aliasing analysis for this work, as none of the tested benchmarks include these cases, and it is our understanding that it is considered bad practice to use these language features unless necessary.

4.4.1.2 Pointer Restrictions

In addition, the use of pointers in C/C++ can lead to several obstacles in the context of dependence analysis, one of which is directly tied to the previous section, since pointers can be used to create several names for the same memory region. In comparison, Fortran pointers are highly restricted. For example, only a variable that is declared with the `TARGET` modifier can be pointed to. Once again, the benchmarks we tested did not include such declarations, so we did not consider the use of pointers in our analysis. Our implementation at the moment does not report the presence of `TARGET` variables or possible aliasing.

4.4.1.3 Loop Semantics

The syntax of traditional "for" loops in languages such as C/C++ or Java is fairly lax, with the loop header consisting of an initial statement, a stopping condition, and a statement that executes after each iteration. For cases in which the loop does not follow the canonical format that is used to express a range of integer values, it can be hard to translate the context of the loop into a valid iteration domain. Fortran "do" loops are more restrictive, with headers that invariably consist of an iterator name and two integer expressions for the lower and upper bounds (and, optionally, a third expression for the stride).

Additionally, when a do-loop is entered in Fortran, the number of iterations is defined before its execution begins. Consider a C/C++ loop that begins with `for (int i = 0; i < n; i++)`, where `n` has a value of `5`. If the value of `n` were to become `10` during the loop's execution, the loop will run for 10 iterations. The complications that this produces from a dependence analysis standpoint are reflected, for example, in the Polyhedral Extraction Tool [60], which does not support loop conditions that depend on a variable that is written to inside the loop. In Fortran, it would have been defined at the start that the loop would run for 5 iterations; the value of `n` is not consulted before each iteration.

4.4.2 Basic Representations

The construction of a loop nest's presentation is performed by iterating over each of its statements in program order, irrespective of their level of nesting. There is no global state regarding the structure of the loop nest; that information is built on a per-statement basis, as shown in Algorithm 1. This is helpful because in an imperfect loop nest, different statements can live under different loop structures.

The data structure resulting from this procedure is used as the basis for the construction of iteration domains and schedules.

4.4.2.1 Iteration Domains

The domain expression is a conjunction of the constraints imposed by the loop bounds. The loop bound expressions of inner loops can depend on the iterators of outer loops, which allows for the correct processing of triangular loops (among others), as long as their bounds can be expressed in Presburger arithmetic. Since the order of execution is enforced by the schedules, the order in which constraints appear in the domain is not a concern.

4.4.2.2 Schedules

A statement's schedule map consists of alternating iterators and AST indexes that represent the position of the statement relative to each loop. This method enforces proper ordering for statements regardless of the "shape" of the loop nest, as shown in Figure 4.8. For the schedules to be

Algorithm 1 Extraction of Loop Information from an AST Statement

```
1:  procedure ExtractLoopInfo(stmt, root)
2:      loops ← empty list
3:      indices ← empty stack
4:      curr ← GetParent(stmt)
5:      prev ← stmt
                                  ▷ Traverse the AST upward to collect loop ancestors
6:      while curr ≠ null do
7:          if curr is a DoStatement then
8:              Prepend(loops, curr)
9:              idx ← GetChildIndex(curr, prev)
10:             Push(indices, idx)
11:         end if
12:         if curr = root then
13:             break
14:         end if
15:         prev ← curr
16:         curr ← GetParent(curr)
17:     end while
18:     result ← empty list
                                  ▷ Process loops from outermost to innermost
19:     for each loop ∈ loops do
20:         var ← loop.iterator
21:         lower ← ProcessBound(loop.lower)
22:         upper ← ProcessBound(loop.upper)
23:         index ← Pop(indices)
24:         Append(result, ⟨var, lower, upper, index⟩)
25:     end for
26:     return result
27: end procedure
```

comparable, they must all contain the same number of elements, so the schedules of statements that are not at the deepest level of loop nesting are padded with zeros.

4.4.2.3 Access Relations

Access relations are constructed by going through the data accesses present in each statement. Depending on the context of the data access, the relation is classified as a read or a write (for example, the left-hand side of an assignment statement is considered a write). Scalar variables are treated as arrays with a single element. In practice, this only means that `[0]` is appended to their names when building the relation string. As discussed previously, it is possible for an array access expression to not have a valid equivalent in Presburger arithmetic. However, this does not mean that the presence of invalid expressions necessarily invalidates the analysis of a loop nest. Let us say that array `a` is indexed through an element of another array (e.g., `a[b[i]]`). As long as array `a` is not written to within the loop nest, it is **not relevant** for the dependence analysis and

```
DO i = 1, 10
  A(i) = B(i) * 2 !S1
  DO j = 1, 10
    B(i) = A(i) !S2
    x = 2 !S3
  END DO
  DO j = 1, 10
    A(i) = x * B(i) !S4
  END DO
END DO
```

$$\mathscr{S}_{S1} = \{S1[i,j] \rightarrow [i,0,0,0]\}$$

$$\mathscr{S}_{S2} = \{S2[i,j] \rightarrow [i,1,j,0]\}$$

$$\mathscr{S}_{S3} = \{S3[i,j] \rightarrow [i,1,j,1]\}$$

$$\mathscr{S}_{S4} = \{S4[i,j] \rightarrow [i,2,j,0]\}$$

Figure 4.8: `isl` schedule maps for a sample loop nest. Note that ordering is enforced across the two j-loops by the constants.

can safely be ignored, since all of the dependence types that we are interested in contain at least one write (the accesses to a read-only variable can be performed in any order).

4.4.2.4 Parameters and Context

A read-only scalar variable that is used in array indexing expressions becomes a **parameter**, which the `isl` will treat as a constant integer value. Iteration domains/access relations that make use of parameters must be prepended with a list of said parameters. An example is shown in Figure 4.9.

$$[n] \rightarrow \{S1[i] \rightarrow A[i+n]\}$$

Figure 4.9: Access relation for expression `A[i + n]`, in which `n` is a parameter.

Additionally, in case the variable was declared with the `PARAMETER` specifier and initialised with an integer literal, it is added to the global context, as shown in Figure 4.4. The presence of this specifier means that the name it is attached to is no more than a name used to refer to a certain constant value, and therefore its use does not need to be policed to the same extent as that of a typical variable.

4.4.3 Non-trivial cases

We have implemented support for some situations that may involve expressions that are data-dependent or that cannot be expressed in Presburger arithmetic, as well as conditional constructs. Our work here does not represent the full extent to which the information contained in conditional constructs and data-dependent expressions can be expressed in Presburger sets and relations. For example, the Polyhedral Extraction Tool [60] has wider support for these cases.

For multiple of the situations described in this section, it was useful to determine whether a given expression was appropriate for the `isl` before proceeding with the analysis. Since this is not trivial in case the expression is not written in a simplified form, an additional library, *Math.js* ([25]), was used in our LARA scripts to facilitate this.

4.4.3.1 Non-affine Loop Bounds

If a loop bound expression that cannot be expressed in Presburger arithmetic is encountered, it is replaced with a parameter (that is to say, an unknown constant), which we will refer to as a surrogate expression. We can safely do this because, as discussed previously, in Fortran, a change to the value a loop bound would evaluate to has no effect on the loop's execution if it happens after the loop is entered.

Since the `isl` will have no knowledge of the value of the surrogate expression, it is free to make **any pessimistic assumption** it desires, leading to a conservative result when the dependence analysis is performed. An example is shown in Figure 4.10.

```
DO i = nx*ny, C[4]
  A(i) = B(i) * 2 !S1
END DO
```

$$\mathscr{D}_{S1} = [nx\_ny, c\_4\_] \rightarrow \{S1[i] : nx\_ny \leq i \leq c\_4\_\}$$

Figure 4.10: Example of a iteration domain produced by our approach when faced with non-affine loop bounds.

4.4.3.2 Conditional Execution

Conditions that restrict the execution of a statement (due to conditional constructs such as `IF`s and `ELSE`s) can be added to its iteration domain, providing a more accurate representation of the loop's execution. The procedure for identifying conditions can be piggybacked off Algorithm 1 by adding additional processing in case a conditional construct is encountered. The different branches of the construct are then traversed until encountering the one that contains the statement that is currently being analysed. Every condition from a branch that precedes it will be marked for **negation**.

A condition expression that cannot be expressed in Presburger arithmetic does not break the analysis, even if it is part of a long `if / else if / else` chain. These can be safely ignored while maintaining the valid components of the chain. This is because, by ignoring a condition, the `isl` will assume that the statement is executed with fewer restrictions, which will at worst produce a pessimistic result.

It would have been possible to insert condition expressions directly into the domain iteration string. However, the `isl`'s syntax does not contain a negation operator. To work around this, each condition is parsed individually in order to create a set, and in case the condition needs to be negated, the set's complement is calculated. These sets are then intersected with the iteration domain. This does not solve the problem entirely, since condition expressions that themselves contain negation operators have to be manipulated so that they can be expressed without them (leaving only one negation at the root of the expression would also be sufficient). The latter (expressions that contain negation operators) is not currently supported but can be implemented, for example, with AST manipulations.

Figure 4.11 shows an example of valid iteration domains in the presence of alternating loops and conditionals. Note that, as stated previously, conditions are each passed as a separate set to the `isl`, but for readability, here they are written in full. The condition found directly above statement S2 cannot be analysed, since it depends on the contents of array X, which the `isl` has no knowledge of. Nevertheless, the negation of the initial `IF`'s condition is added to S2 and S3's domains. Once again, this is **pessimistic**, since it causes an overlap between their domains, but it will not lead to an incorrect result.

```
DO i = 1, 100
  IF (i <= 50) THEN
    DO j = 1, 50
      A(j, i) = 0.0 !S1
    END DO
  ELSE IF (X(i) == 0) THEN
    B(i) = 1.0 !S2
  ELSE
    C(i) = 2.0 !S3
  END IF
END DO
```

$$\mathscr{D}_{S1} = \{S1[i,j] : 1 \leq i \leq 100 \wedge i \leq 50 \wedge 1 \leq j \leq 50\}$$

$$\mathscr{D}_{S2} = \{S2[i] : 1 \leq i \leq 100 \wedge i > 50\}$$

$$\mathscr{D}_{S3} = \{S3[i] : 1 \leq i \leq 100 \wedge i > 50\}$$

Figure 4.11: Example of iteration domains for an alternating loop-conditional structure.

4.5 Using the Integer Set Library

Once a suitable representation of the loop nest is formed, it can be passed along to the `isl`. We call the library's functions through its Python bindings (`islpy`[1]).

The sets and relations described in the previous sections can be fed to the `isl` in plaintext. The library will parse these and convert them into the appropriate internal representation. If conditions are to be applied to a statement, they are each parsed individually and intersected with the statement's domain. Read and write accesses are stored separately. For each type of dependence, the appropriate lists are marked as source and sink (the same for both in the case of output dependencies), and the corresponding dependencies are calculated.

Distance vectors are obtained by extracting the respective coordinates from each element of the dependence sets returned by the library. To avoid a fragmented development environment, the Python script was kept minimal, processing only the data and returning a list of dependencies. The analysis itself is handled by the main TypeScript code. The main steps of the script are shown in Algorithm 2, which consists mainly of calling the `isl` functions associated with dependence analysis.

[1] https://pypi.org/project/islpy

Algorithm 2 Polyhedral Dependence Analysis Pipeline (`isl`)

procedure ANALYSEDEPENDENCIES($\mathscr{D}, \mathscr{S}, \mathscr{R}, \mathscr{W}, \mathscr{C}$)
  Input: Domains $\mathscr{D}$, Schedules $\mathscr{S}$, Read Maps $\mathscr{R}$, Write Maps $\mathscr{W}$, Global Context $\mathscr{C}$
  ▷ 1. Parse and Assemble Polyhedral Sets
  $U_{\text{domain}} \leftarrow \text{UNION}(\mathscr{D}) \cap \mathscr{C}$
  $U_{\text{schedule}} \leftarrow \text{UNION}(\mathscr{S}) \cap \mathscr{C}$
  $U_{\text{reads}} \leftarrow \text{UNION}(\mathscr{R}) \cap \mathscr{C}$
  $U_{\text{writes}} \leftarrow \text{UNION}(\mathscr{W}) \cap \mathscr{C}$
  ▷ 2. Compute Exact Data Flows
  function GETFLOW($sink, source$)
    $acc \leftarrow \text{ACCESSINFO}(sink \cap U_{\text{domain}})$
    $acc \leftarrow \text{SETSOURCE}(acc, source \cap U_{\text{domain}})$
    $acc \leftarrow \text{SETSCHEDULE}(acc, U_{\text{schedule}})$
    return COMPUTEFLOW(acc)
  end function
  $Flow_{\text{RAW}} \leftarrow \text{GETFLOW}(U_{\text{reads}}, U_{\text{writes}})$
  $Flow_{\text{WAW}} \leftarrow \text{GETFLOW}(U_{\text{writes}}, U_{\text{writes}})$
  $Flow_{\text{WAR}} \leftarrow \text{GETFLOW}(U_{\text{writes}}, U_{\text{reads}})$
  $Deps_{\text{RAW}} \leftarrow \text{EXTRACTDEPENDENCES}(Flow_{\text{RAW}})$
  $Deps_{\text{WAW}} \leftarrow \text{EXTRACTDEPENDENCES}(Flow_{\text{WAW}})$
  $Deps_{\text{WAR}} \leftarrow \text{EXTRACTDEPENDENCES}(Flow_{\text{WAR}})$
  $results \leftarrow$ empty list
  ▷ 3. Extract Distance Vectors for each Dependence Type
  for each $depType \in \{RAW, WAW, WAR\}$ **do**
    for each $map \in Deps_{depType}$ **do**
      $sourceStmt \leftarrow map.in$
      $sinkStmt \leftarrow map.out$
      $var \leftarrow \text{IDENTIFYTARGETVARIABLE}(map, \mathscr{R}, \mathscr{W})$
      ▷ Map the dependence into the schedule space
      $schedMap \leftarrow map \circ U_{\text{schedule}}^{-1} \circ U_{\text{schedule}}$
      $deltas \leftarrow \text{COMPUTEDELTAS}(schedMap)$
      for each $vector \in deltas$ **do**
        $\text{APPEND}(results, \langle depType, var, sourceStmt, sinkStmt, vector \rangle)$
      end for
    end for
  end for
  return $results$
end procedure

4.6 Dependence Resolution and Code Generation

Once an exhaustive list of data dependencies and their corresponding distance vectors is obtained, the algorithm must classify each variable to determine if the loop can be safely parallelised. This classification is performed by a dedicated dependence analysis module that evaluates the source and sink of each dependence, its type (RAW, WAW, or WAR), and the first element of its distance vector.

4.6.1 Distance Vector Evaluation

Since the goal is to parallelize the outermost loop, we will be mostly focused on the first element of each distance vector. If the distance is zero (loop-independent dependence), it does not serialize the loop. However, if the target variable of the dependence is a scalar, it is flagged for privatisation.

If the distance is greater than zero (outer loop-carried dependence), it must be resolved through scalar privatisation or a reduction. If this is not possible, the loop is flagged as sequential, and the parallelisation is aborted.

4.6.2 Scalar Privatisation

If an outer loop-carried dependence acts upon a scalar variable, and the dependence is strictly limited to WAW or WAR types, the variable is safely added to the privatisation set. Because no loop-carried flow (RAW) dependencies exist, the iterations do not fundamentally rely on data produced by previous iterations, and allocating thread-local memory successfully eliminates the storage-reuse conflicts.

4.6.3 Detecting and Validating Reductions

To identify reduction variables, we employ a hybrid approach that considers its structure and positioning in the AST and the dependence pattern discussed in Section 4.3.2.

When a loop-carried dependence is detected, the source statement's AST node is inspected. If the statement is an assignment utilizing a binary operator supported by OpenMP's `reduction` clause (e.g., addition or multiplication) and the target variable appears on both sides of the expression, it is flagged as a reduction candidate.

The candidate is validated by tallying the dependencies acting upon the variable. As previously established, a cyclical pattern of three dependencies (one of each type) is necessary for a reduction to exist. Therefore, the transpiler expects exactly one RAW, one WAW and one WAR dependence mapping the statement to itself. If the final count deviates from this, the reduction validation fails.

A failure in reduction validation does not always imply that the loop is sequential. If the reduction candidate is a scalar variable, and the reduction pattern was broken by a lack of loop-carried flow dependencies, the candidate can be privatised.

4.6.4 AST Manipulation and OpenMP Injection

If the dependence resolution phase concludes without detecting any unresolved obstacles to parallelisation, the transpiler proceeds to the code generation phase. Using the LARA framework's AST manipulation APIs, the original `DO` loop is wrapped in an `!$OMP PARALLEL DO` construct. The validated sets of private and reduction variables are directly translated into `PRIVATE()` and `REDUCTION()` OpenMP clauses. If the specification part of the corresponding Fortran program unit lacks a `USE omp_lib` statement, one is inserted.

4.7 Stencil Transformations

Unlike purely parallel or sequentially dependent loops, stencil algorithms can often be parallelised using pipelined execution. In these programs, an outer temporal loop typically drives calculations over a two-dimensional spatial grid. Because computations frequently perform in-place updates using neighbouring grid elements, they introduce multiple conflicting dependencies. To resolve this, our approach implements "Doacross" synchronisation.

Generating synchronizing directives for pipeline (or "doacross") parallelism is generally not supported by standard polyhedral AST generation algorithms. Shirako et al. [51], who contributed to the proposal for the addition of these directives into the OpenMP standard [52], presented an early approach that also integrated polyhedral analysis and AST-based transformations to extract fine-grained parallelism. This approach predates the OpenMP 4.5 standard, and as such is reliant on custom extensions.

4.7.1 Distance Vectors

So far, we have only been concerned with the first element of the distance vectors, since inner loop-carried dependencies are not an obstacle to the insertion of a simple parallelisation directive on the outer loop. However, subsequent elements of the vector can be used to obtain a more fine-grained view of how a statement depends on previous iterations. This can be used to determine the content of a `depend` clause. For the statement in Figure 4.7, the distance vector would be `[1,-1]`: one step back in the `i` dimension, one step forward in the `j` dimension. With this information, we are able to obtain the correct sink expression.

A more complex stencil may make use of several neighbouring grid elements. Let us say that the formula for element `[i, j]` reads from elements `[i-1, j-1]` and `[i-1, j]`. Since we are trying to apply a parallelisation directive to the i-loop, each "instance" of the j-loop is executed sequentially by one of the threads, which means `[i-1, j-1]` will always be ready by the time `[i-1, j]` is. Thus, a `[i-1, j-1]` sink would be redundant. To avoid unnecessarily verbose `depend` clauses, the distance vectors are lexicographically compared and only the "largest" ones are kept.

The stencil analysis process was kept separate from the rest of the transformations, as we consider them to be a proof of concept at this time. Unless a section of this document explicitly discusses them, they were not used to obtain the results they present.

4.7.2 OpenMP Scheduling

The efficiency of "doacross" pipeline parallelism is, in part, dependent on the strategy used for thread scheduling. As noted in a practical case study regarding OpenMP's "doacross" constructs [26], employing `schedule(static, 1)` **yields the best work balancing**. By default, a static schedule divides the iteration space into large, contiguous chunks. This can severely disrupt the pipeline pattern that these constructs produce; a thread assigned to a subsequent chunk remains

entirely idle until the preceding thread finishes its entire block and satisfies its dependencies at the boundary. This delay prevents the pipeline from filling and effectively serializes execution. A chunk size of 1 distributes individual iterations to each thread in a round-robin fashion, allowing the pipeline to rapidly propagate to all available threads.

4.8 Post Processing

After parallelisation, a few transformation passes are applied to the new code to perform slight refinements before presenting the final result.

4.8.1 Parallel Region Merging

When two or more consecutive loop nests are successfully parallelised by our transformations, they will each outline one parallel region (`!$OMP PARALLEL DO`). One post processing step consists of taking these consecutive loop nests and wrapping them in a single parallel region. Modern OpenMP runtimes implement complex mechanisms to manage their thread pools, so minimizing the number of parallel regions is not likely to have a substantial impact on performance. However, it can make further changes to the code more convenient. For example, with several parallel loops in one parallel region, it is possible to control synchronisation across the loops with `nowait` clauses, if appropriate.

4.8.2 Parallel Region Lifting

Under the assumptions that we make for the initial version of our stencil transformations, parallelisation is applied to an outer spatial loop that is the sole direct child of a temporal loop. As such, for reasons similar to those stated in the previous section, we "lift" the limits of the parallel region to the temporal loop.

4.9 Summary

This chapter described the concepts, algorithms, and steps required to perform automated dependence analysis and safe OpenMP parallelisation of Fortran loop nests. We detailed how high-level AST information is extracted and translated into a mathematical polyhedral representation tailored for the Integer Set Library (`isl`).

The dependence analysis algorithm determines the exact data flows and calculates distances vectors across loop iterations. By applying conservative approximations on non-trivial Fortran constructs, such as non-affine loop bounds, support is extended beyond that which can be directly translated to Presburger arithmetic. Dependencies are evaluated according to their characteristics, determining whether they serialize execution or can be safely mitigated by using parallel programming patterns such as privatisation.

After performing the dependence resolution, this information was used to guide AST transformations, automatically inserting the appropriate OpenMP `PARALLEL DO` constructs and data-sharing clauses into the source code. Furthermore, we detailed a proof-of-concept extension for stencil algorithms that exploits fine-grained information obtained through distance vectors to automatically generate OpenMP "doacross"-style constructs. Finally, post-processing steps such as parallel region merging and lifting were presented as mechanisms to further clean the generated code.

Chapter 5

Source-to-source Fortran Compilation with the LARA Framework

The LARA framework [43] is a set of libraries and tools for building source-to-source compilers that support analyses and transformations defined in LARA scripts. LARA scripts are plain, Node.JS compatible TypeScript/JavaScript scripts which use LARA APIs for accessing and modifying an AST representation of the code. A notable example of a LARA-based source-to-source compiler is Clava [10], which targets C/C++. The Fortran source-to-source compiler used in this work (Metafor) is still in its infancy compared to Clava, and as such, some work had to be done in Metafor itself to implement the features needed to support a dependence analysis and parallelisation framework.

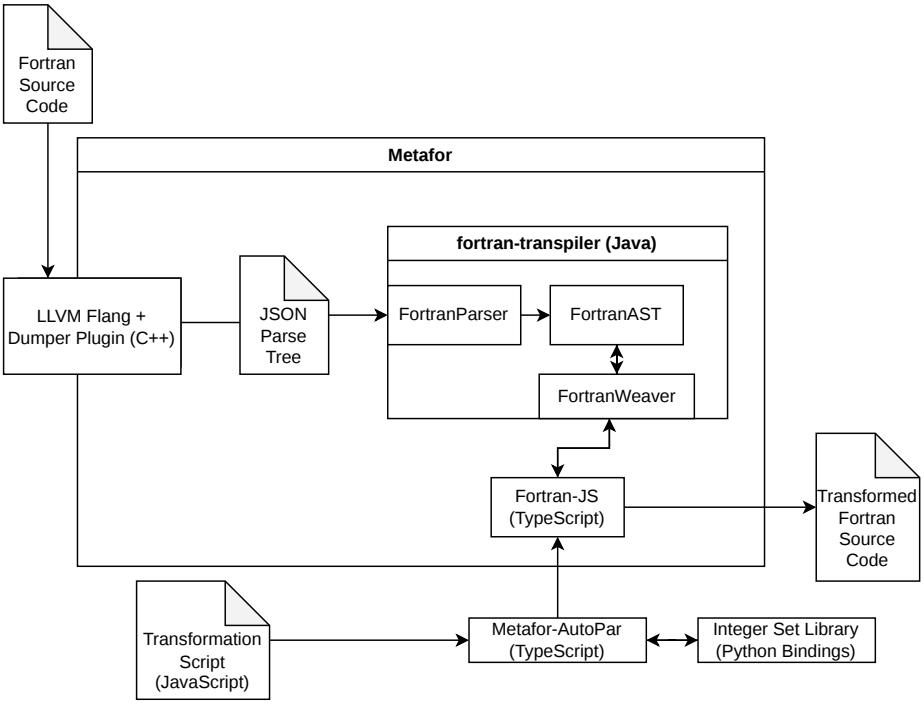

Figure 5.1: Metafor-AutoPar high-level architectural diagram.

5.1 Metafor Components

Metafor is mainly made up of three components that come together to form a full end-to-end execution flow for source code transformations. A high-level architectural overview of Metafor[1] and Metafor-AutoPar[2] is presented in Figure 5.1. This chapter will provide details on the functions of Metafor's main components (`fortran-transpiler` Java projects, `Fortran-JS` scripting entry point) as well as the dumper plugin that extracts information from LLVM Flang's parsing step.

5.1.1 Fortran Transpiler

The source-to-source compilation itself is written in **Java**. It is split into three Java projects that work alongside each other:

- **Fortran AST** outlines the different nodes that make up our AST representation of Fortran programs. This includes their hierarchical structure (for example, a binary operation is an expression), the children they may have, and how to generate source code for a given node.
- **Fortran Parser** builds the AST based on the information received from Flang's parsing step.
- **Fortran Weaver** defines the interface that is exposed to the programmer via the Node.js component.

5.1.2 Flang Dumper

Flang's parsing step is used to produce the representation of a program that is fed into the transpiler. This is advantageous for a couple of reasons. The most obvious one is that parsing source code is hard, and developing a custom parsing solution would be a lengthy process. Additionally, if a new version of the compiler is released, Metafor can be updated to be compatible with it with relative ease (depending on the scope of the changes, of course, but the parsing structure is expected to be mostly stable).

The dumper[3] is implemented as a Flang plugin that outputs the compiler's parse tree in a JavaScript Object Notation (JSON) format. There is a mismatch in the way hierarchy is handled in Flang and in Metafor. While Metafor uses inheritance (a binary operation is an expression), Flang's parse tree uses composition (an expression can contain a binary operation). This can lead to some complications; for example, it may not be clear what the inheritance structure should be in case a node can be contained in one of several types of node. However, Metafor's AST does not need to be a perfect reflection of Flang's parse tree, and they do differ in some cases.

[1] https://github.com/specs-feup/fortran-transpiler
[2] https://github.com/Onso37/metafor-omp
[3] https://github.com/specs-feup/flang-dumper

5.1.3 Node.js Entry Point

Modern LARA-based compilers expose their functionality to the programmer via a TypeScript interface. The nodes of the AST can be accessed through **join points**, ephemeral objects that represent points of interest in the code, and that in our case wrap the underlying Java object representing the AST node. For example, querying the AST twice to find the node corresponding to the root of the main program will yield two distinct joinpoint objects that refer to the same Java AST node.

The structure of the join points is declared in Extensible Markup Language (XML) files found in the FortranWeaver Java project. In that same project, implementations of the join points are created. Generally, these implementations contain little more than wrapped calls to the node's methods, but they are kept separate from the underlying node class by design: the high-level nature of join points allows for the incremental exposure of features to the scripting interface.

5.2 Adding Support for Language Constructs

The overwhelming majority of the effort put into Metafor itself in the context of this work was focused on adding support for Fortran language constructs. This can be more or less challenging depending on the complexity of the construct and whether the automatic behaviour of the Flang dumper is able to locate all the information that is required for the output. We will go over the full process of implementing support for a language construct that requires changes to the dumper by exemplifying with the `DO` construct. Afterwards, we will discuss a couple of small challenges that were faced during the implementation of support for other language constructs.

5.2.1 Manually Adjusting the Dumper

A sizable portion of the parse tree follows a consistent naming scheme for the elements of its data structures. The dumper is prepared for this and automatically dumps any information that can be found by searching for names that follow this scheme. When this is not the case, the JSON output produced by the dumper will contain incomplete information and dangling references to nodes that seem to not be part of any other structure. Figure 5.2 shows a snippet of the output for the bounds of a do-loop using automatic dumping.

The "LoopBounds" node is empty, and the nodes that pertain to the loop variable's name ("i") and its bound expressions are detached from the overall structure. To figure out how to obtain the node's information, we must look at the source code for Flang's parse tree (the data structure for LoopBounds is shown in Figure 5.3) and manually add the desired fields (in this case, `name`, `lower`, `upper`, and `step`) to the node's dump function (Figure 5.4).

5.2.2 Creating an AST Node

To create a Java class that represents a node of the AST, we must first consider the structure we want it to have in the tree. As previously discussed, this does not need to be a one-to-one recreation

```
{
    "id": "0x5da936a45c58-LoopBounds"
},
{
    "id": "0x5da936a45c58-Name",
    "source": "i"
},
{
    "id": "0x5da936a46940-Expr",
    "variantKey": "LiteralConstant",
    "LiteralConstant": "0x5da936a46960-LiteralConstant"
},
{
    "id": "0x5da936a46960-LiteralConstant",
    "variantKey": "IntLiteralConstant",
    "IntLiteralConstant": "0x5da936a46960-IntLiteralConstant"
},
{
    "id": "0x5da936a46960-IntLiteralConstant",
    "CharBlock": "1"
},
```


Figure 5.2: Snippet of the output produced by the unmodified dumper when faced with loop bounds.

of Flang's parse tree. In the "do" construct's case, the main construct has three children in the parse tree: a "do statement" corresponding to the line that starts the loop, a block of statements, and an "end do statement". To match the nodes that were already implemented, the "end do statement" is not stored in the AST, as it does not contain any information that can't already be obtained from the "do statement". It is the main loop class' responsibility to produce that statement in its code generation method.

Some levels of composition are also entirely eliminated, in order to avoid excessive nesting in the AST. If the intermediate nodes that were eliminated contained useful information, said information can be stored in the parent (or child) node. In Flang's parse tree, the "do statement" contains a "loop control". In our AST, the loop control is a direct descendant of the main construct. Additionally, our LoopControl class is an abstract base that allows the different kinds of loop control (range-based, while, `DO CONCURRENT`) to be interchangeable, which creates a similar behaviour to what Flang achieves with C++ variants[4].

5.2.3 Parsing the Parse Tree

After confirming that a node is dumped correctly and creating an AST class for it, we must implement a method that takes the information from the dumped node and loads it into an AST object. The complexity of this method depends partly on how many liberties were taken in the previous step: the more our AST differs from Flang's parse tree for a given node, the more processing is

[4] https://en.cppreference.com/cpp/utility/variant

```
template <typename VAR, typename BOUND> struct LoopBounds {
  LoopBounds(LoopBounds &&that) = default;
  LoopBounds(
      VAR &&name, BOUND &&lower, BOUND &&upper, std::optional<BOUND> &&step)
      : name{std::move(name)}, lower{std::move(lower)}, upper{std::move(upper)},
        step{std::move(step)} {}
  LoopBounds &operator=(LoopBounds &&) = default;
  VAR name;
  BOUND lower, upper;
  std::optional<BOUND> step;
};
```

Figure 5.3: LoopBounds data structure in Flang's parse tree.

required to bridge the gap between the two. These functions, which are implemented in the FortranParser Java project, extract the node's information from its JSON representation and add its children to the node itself.

When parsing a node, the parser only "stops" and calls one of the parsing functions if the node matches one of the node types that is associated with a parsing method. Otherwise, the parser tries to repeat this step with the node's direct descendant, if it exists. For example, a generic "Expr" node has no associated parsing method, but it may contain a "Multiply" node, which does.

5.2.4 Exposing the Node to the TypeScript Interface

Creating the joinpoint class that will be visible in the TypeScript interface is a fairly straightforward process. The properties of the class are defined in XML files (Figure 5.5). This information is used to generate an abstract Java class for which a concrete implementation must be created manually, creating a link between the properties of the joinpoint and the underlying AST node.

An automated process then generates TypeScript classes that are ready to be used by code transformation scripts (Figure 5.6).

```
DUMP_NODE_MANUAL(Fortran::parser::LoopControl::Bounds, {
    dump(v.name.thing, "var");
    dump(v.lower.thing, "lower");
    dump(v.upper.thing, "upper");
    if(v.step.has_value()) dump(v.step.value().thing, "step");
})
```

Figure 5.4: Dump function for loop bounds in Flang's parse tree.

```
<joinpoint class="doStatement" extends="executableStatement" tooltip="Represents a
    do loop"/>

<artifact class="doStatement">
    <attribute name="control" type="loopControl"/>
    <attribute name="body" type="execution"/>
</artifact>
```

Figure 5.5: XML snippets pertaining to the "DoStatement" joinpoint type.

```
export class DoStatement extends ExecutableStatement {
  /**
   * @internal
   */
  static readonly _defaultAttributeInfo: {readonly map?: DefaultAttributeMap,
      readonly name: string | null, readonly type?: PrivateMapper, readonly
      jpMapper?: typeof JoinpointMapper} = {
    name: null,
  };
  get body(): Execution { return wrapJoinPoint(this._javaObject.getBody()) }
  get control(): LoopControl { return wrapJoinPoint(this._javaObject.getControl())
      }
}
```

Figure 5.6: TypeScript class representing the "DoStatement" joinpoint type.

Chapter 6

Experimental Evaluation

This chapter presents the methodology for our experimental evaluation process, as well as its results and our analysis of the results.

6.1 Execution Environment

6.1.1 Hardware Specification

The experimental evaluation was conducted on a dual-socket system equipped with two Intel Xeon E5-2630 v3 processors running at a base frequency of 2.40 GHz, which is potentially subject to temporary reductions via thermal throttling. To ensure stable execution times, Intel Turbo Boost was disabled. Each physical processor contains 8 independent compute cores. Hyper-threading was not used.

The memory sub-system is organised under a symmetric NUMA topology split across two distinct hardware domains, as detailed in Table 6.1. The NUMA domains are evenly split across the two sockets.

Architectural Metric	**System Specification**
Processor Model	Intel Xeon E5-2630 v3 (Haswell-EP)
Sockets $\times$ Cores per Socket	2×8 (16 Physical Cores Total)
Base Clock Frequency	2.40 GHz (Turbo Boost Disabled)
L1 Cache	64 KiB per Core
L2 Cache	256 KiB per Core
L3 Shared Cache	40 MiB Total (2×20 MiB Shared per Socket)
NUMA Configuration	2 Nodes

Table 6.1: Hardware architectural specification of the system.

6.1.2 Thread Affinity

Some OpenMP environment variables were used to precisely control the mapping of OpenMP threads to physical processors. Because they are sensitive to the physical placement of the pro-

gram's data, this is of particular relevance to our NUMA-related results. `OMP_PLACES` was used to map one thread to one physical processor, and `OMP_PROC_BIND` was used to instruct the OpenMP runtime not to move threads during execution and to place them close to each other.

Environment Variable	Configured Value
`OMP_NUM_THREADS`	`16`
`OMP_PLACES`	`cores`
`OMP_PROC_BIND`	`close`

Table 6.2: OpenMP environment variable configurations used for testing.

Under these configurations, the thread pool is packed sequentially across the processors' physical cores. Threads 0-7 will be placed in the first socket, and threads 8-15 in the second one. For parallel loops, static scheduling with an unspecified chunk size is used.

6.1.3 NUMA

The system's 128 GiB of RAM are evenly split between the two sockets, each associated with one NUMA domain. In each socket, four 16 GiB memory modules (all running at 1866 MT/s) are arranged in a quad-channel configuration.

NUMA can be enabled or disabled in the system's BIOS. Note that this has no effect on the physical configuration of the memory; there are still two distinct memory domains, and the cost of accessing a given value is still dependent on its physical placement. All this does is "trick" the system into seeing the memory as one cohesive whole, rendering it unaware of the NUMA nature of the memory hardware. In practice, data pages are interleaved across the two memory domains, with the goal of establishing a sufficiently uniform cost for memory accesses. When NUMA is enabled, the first touch data placement policy discussed in Section 2.5 is always used.

For the purposes of our transformations, parallelising the initialisation step is not a major concern, as it is often embarrassingly parallel. Threads are pinned to the two CPUs using the ordering outlined in Section 6.1.2, placing half of a given benchmark's dataset on each socket, close to the threads that will make use of that data (ideally), when the data intialisation step is parallelised.

6.1.4 Compilation

Every benchmark was compiled with version 22 of LLVM Flang, using the `-O3` optimisation flag. When compiling with Polly, the following flags are added: `-mllvm -polly -mllvm -polly-parallel -mllvm -polly-omp-backend=LLVM`. Because LLVM Flang does not currently support OpenMP's `ordered` construct, `gfortran` 15.2.0 is exceptionally used for their evaluation, also with the `-O3` flag.

6.1.5 Benchmarks

To evaluate our transformations, we used **PolyBench/Fortran** [44] and version 3.3.1 of the **NAS Parallel Benchmarks (NPB)** [6]. For the latter, we were unable to apply our approach to most of the individual programs since their code is split into multiple files. Metafor is still in its infancy compared to other LARA compilers (e.g., Clava), and lacks the inter-translation unit (e.g., multiple source files) analysis features that were used for similar code transformations applied to C/C++ [5]. Thus, among the benchmarks found in the NPB, we focus only on the Conjugate Gradient (CG) benchmark.

In the case of CG, both sequential and parallelised versions of the benchmark are available. Thus, we can measure the speedup our transformed version achieves and how it compares to a manually parallelised version. In the case of PolyBench, parallelised versions are not included. Some "manually" parallelised versions were produced by prompting generic Large Language Models (LLMs).

6.1.5.1 Benchmark Configurations

For each of the benchmarks that were tested, several parameters were considered, resulting in different configurations. The first pertains to whether NUMA is enabled on the system's BIOS or not. One of the other parameters applies only to cases in which NUMA is enabled, that being the presence of **automatic NUMA balancing**. This is a feature present in some Linux-based operating systems that seeks to improve the performance of generic applications in a NUMA environment. It can operate in two ways: moving threads/processes closer to the data they access, or moving data closer to the threads/processes that access it. Due to the way in which we define OpenMP thread affinity, the former will not happen. We used kernel-level balancing due to its ease of use and availability, but other approaches have been developed as alternatives that show more promise in the context of scientific computing [15].

6.1.5.2 Transformation Script

The script we used to transform the benchmarks makes use of custom compiler directives (`!DIR$`) to delimit the portions of the program in which we are interested. Every loop nest found within the limits is analysed, and if it is determined to be sequential, further analysis is attempted for inner loops.

6.2 Validation

The most extensive testing and validation was done with PolyBench. We developed a bash script pipeline to automate the testing process for all of the benchmarks. This pipeline creates a transformed version of each benchmark, which are then compiled and executed along with the original. The results for each are stored in separate text files, which can be compared to ensure correctness. If performance markers (such as runtime) are present, they are stripped before the comparison.

A simple exact text match was sufficient for nearly every case; only benchmarks in which the transformed code included a reduction operation on an array were found to have slight differences in their output. These differences are not statistically significant, yielding a negligible relative error of less than 10^{-14}. This drift can be attributed to the non-associativity of floating-point operations: because OpenMP does not guarantee the exact order in which threads contribute their local accumulations to the global reduction, minor errors accumulate.

In the case of the NPB's CG benchmark, verification is built into the program.

6.3 Algorithmic Complexity and Data Placement

The difference in execution times after parallelizing the data initialisation step is not uniform across the benchmarks. To understand why, we must examine the underlying algorithms, specifically their asymptotic complexities and operational intensities.

First, consider **`gemm`**, **`2mm`**, and **`3mm`**, which consist of one, two, or three matrix multiplications, respectively. These are arithmetically intensive $O(n^3)$ algorithms that already display near-perfect speedups without explicitly harnessing the first touch policy (in other words, when the data initialisation step is performed sequentially by a single thread, placing the data in that thread's NUMA domain). The values loaded into the shared L3 cache are reused extensively without requiring subsequent reads from main memory, amortizing the initial cost of fetching data from a remote NUMA domain.

Conversely, benchmarks such as **`gemver`** and **`gesummv`** consist of $O(n^2)$ matrix-vector operations. Because fewer arithmetic operations are performed per byte loaded, the execution will likely be bandwidth-bound, making the cost of remote memory accesses significant compared to the computations themselves.

Even among benchmarks with similar loop structures, the benefits of parallelizing the initialisation step vary. The **`mvt`** benchmark is one of the cases that yields subpar gains for large datasets. In its kernel function, the matrix `a` is accessed via both `a(i, j)` and `a(j, i)`. While `a(j, i)` matches the contiguous access pattern found in the initialisation function, the `a(i, j)` traversal conflicts with the physical placement of the data, forcing large strides between memory accesses, negating spatial locality benefits from the data that could be obtained at the cache and NUMA level.

However, an array access expression that does not perfectly match the initialisation does not universally negate the first touch policy. In the successfully parallelised stencils, like **`jacobi-2d-imper`**, the computation of a single grid element reads from adjacent spatial coordinates (Figure 6.1). Because the data grid is cleanly split down the middle between the two NUMA domains due to the thread affinity environment variables we specified previously, adjacent elements almost always reside in the correct, local NUMA domain. Remote memory accesses are limited to the vicinity of the boundary line that splits the two domains.

```
1 b(j, i) = 0.2D0 * (a(j, i) + a(j - 1, i) + a(1 + j, i) + &
2                    a(j, 1 + i) + a(j, i - 1))
```

Figure 6.1: Main formula of the `jacobi-2d-imper` kernel

6.4 PolyBench/Fortran

Our evaluation of these benchmarks is focused on its `LARGE` and `EXTRALARGE` configurations. The difference in their memory footprints was important for validating that execution time differences produced by our NUMA-focused transformations resulted from the properties of a NUMA topology.

For the experiments presented in this section, **Parallel Init.** and **Sequential Init.** refer to configurations in which the data initialisation step is parallel or sequential respectively. Each version of the benchmark is executed with and without automatic NUMA balancing. Each benchmark is executed 20 times. For this section and subsequent sections, the error bars shown in figures represent one standard deviation. For the most part, execution times are stable and there is little overlap between results for any two configurations that have different means in a given benchmark, with the exception of benchmarks with a small execution time, which are prone to being affected by minute variations caused, for example, by the operating system.

Two of the benchmarks (**`gesummv`** and **`durbin`**) are not considered, since we are unable to execute them in their original EXTRALARGE configurations due to memory issues. They are the only benchmarks that attempt to allocate two 100000×100000 matrices, which requires roughly 170 GiB of RAM, exceeding the capacity of our system.

6.4.1 Parallelisation Coverage

The extent to which our approach was able to parallelize PolyBench/Fortran's benchmarks is presented in Table 6.3. The number of lines of code in each benchmark's kernel function is also shown. The parallelisation of outer loops is generally ideal, though the number can be misleading. For example, in **`durbin`**'s case, the parallelised outer loop nest is just a loop with a single statement that copies the results from one array to another after the algorithm is finished. We can observe that linear algebra kernels tend to be the most inherently parallel.

6.4.2 Cache-Resident Datasets (LARGE)

The results for the LARGE dataset display an unexpected symmetry between the cases in which NUMA is enabled and those in which it is not. Starting with the former, when the data initialisation step is not parallelised, all the data will be physically placed in Core 0's NUMA domain. This is the worst-case scenario for the threads on the other side of the system, since all of their data will have to be pulled from non-local memory. By making use of the first-touch policy, the data is

Benchmark	Kernel Lines	Parallel Loops by Nesting Level			
		Outer	One Level	Two Levels	Total
Datamining					
correlation	43	4	–	–	4
covariance	26	3	–	–	3
Medley					
floyd-warshall	9	–	–	–	0
reg_detect	30	–	3	1	4
Stencils					
fdtd-2d	21	–	4	–	4
fdtd-apml	61	1	–	–	1
adi	46	–	5	2	7
jacobi-1d-imper	9	–	2	–	2
jacobi-2d-imper	13	–	2	–	2
seidel-2d	10	–	–	–	0
Linear Algebra – Kernels					
mvt	10	2	–	–	2
syr2k	13	2	–	–	2
syrk	12	2	–	–	2
cholesky	14	–	1	–	1
bicg	11	2	–	–	2
gemver	18	4	–	–	4
3mm	29	3	–	–	3
atax	13	2	–	–	2
2mm	17	2	–	–	2
symm	11	–	1	–	1
gemm	8	1	–	–	1
doitgen	14	1	–	–	1
gesummv	9	1	–	–	1
trisolv	7	–	–	–	0
trmm	7	–	–	–	0
Linear Algebra – Solvers					
durbin	21	1	1	–	2
dynprog	20	–	2	–	2
ludcmp	33	–	2	–	2
lu	10	–	2	–	2
gramschmidt	19	–	3	–	3

Table 6.3: PolyBench/Fortran Benchmark Statistics: Kernel Lines and Parallel Loops by Nesting Level

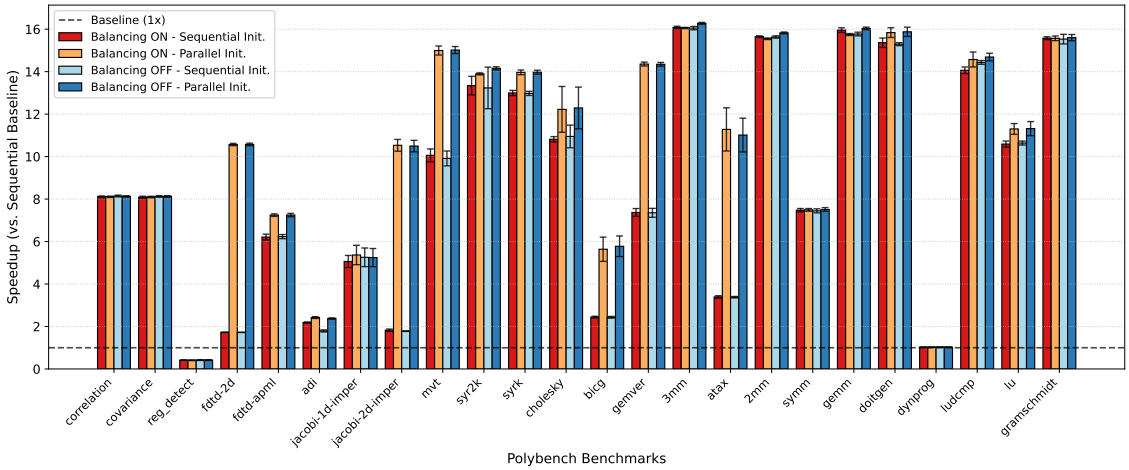

Figure 6.2: Average speedup of PolyBench when NUMA is turned on (LARGE).

placed in the local NUMA domain, leading to better runtimes for latency-bound benchmarks, as shown in Figure 6.2.

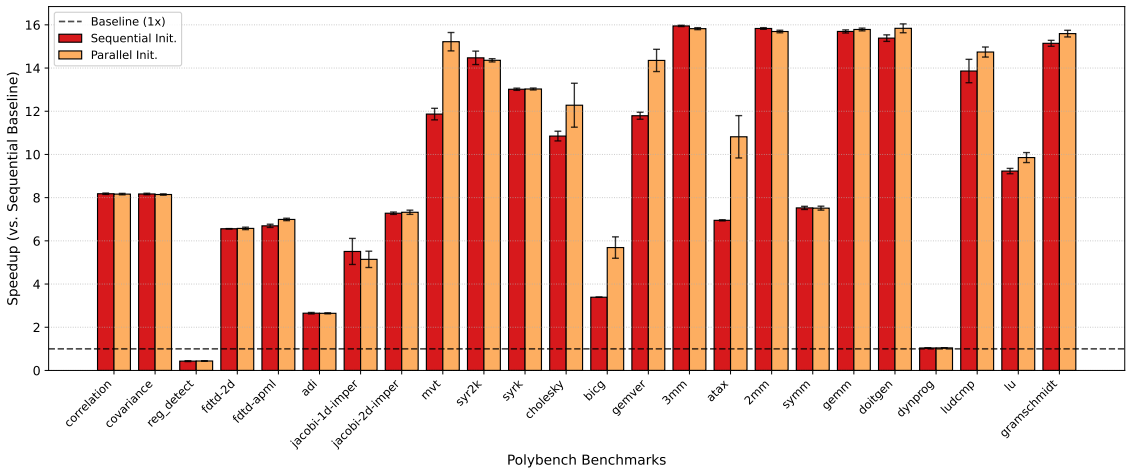

Figure 6.3: Speedup of PolyBench when NUMA is turned off (LARGE).

With NUMA disabled, it is expected that every thread will be in an "average" scenario, in which roughly half of the data it needs will have to be pulled from the opposing domain. However, parallelizing the initialisation when NUMA is disabled leads to results similar to those observed when NUMA is enabled, in the sense that some benchmarks perform better, as shown in Figure 6.3. At first, this seems counterintuitive since the first touch policy does not apply, but to understand why this occurs, we must consider a secondary effect of parallelised initialisation. Assuming the processors do not have significant work to perform between initialisation and computation, **their caches, in particular the shared L3 cache, might already be "warmed up" with local data by the time the computation begins**. The same can apply to each core's Translation Lookaside Buffer. We can confirm that for some cases, all of the problem data fits in the combined 40 MiB shared L3 cache by calculating the memory footprint of the benchmarks (an example is shown

in Figure 6.4). An additional factor to consider is that the OpenMP runtime is able to "keep" the threads that are created in the initialisation region and reuse them when the next parallel region (the benchmark's kernel) is entered, avoiding additional forking overhead, which will be mostly visible on benchmarks with very short execution times. A Welch's t-test ($\alpha = 0.05$) shows a significant difference in all but 8 benchmarks.

This affects both the NUMA and non-NUMA executions, indicating that the gains seen in the NUMA cases are partially due to caching and independent of the first-touch policy. We can, however, conclude that not accounting for the physical memory topology has a negative effect on execution times, since benchmarks with sequential initialisations perform better when NUMA is disabled.

$$\text{Footprint} = (2000 \times 2000 \times 8\ \text{B}) + (2000 \times 8\ \text{B}) \approx 32\ \text{MiB}$$

Figure 6.4: Memory footprint calculation for the **`cholesky`** kernel under the `LARGE` dataset configuration.

6.4.3 Memory-Bound Datasets (EXTRALARGE)

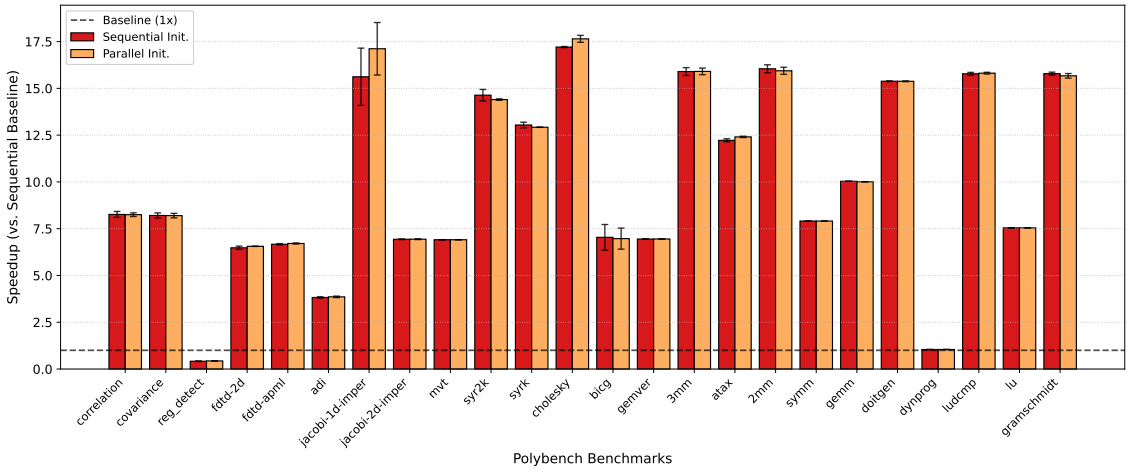

Figure 6.5: Average speedup of PolyBench when NUMA is turned off (EXTRALARGE).

The results when NUMA is switched off confirm our cache warming and thread activation hypothesis for the LARGE dataset; with EXTRALARGE data, there is no longer any substantial difference in execution times when the data initialisation step is parallelised (Figure 6.5), with the exception of **`jacobi-1d-imper`**, which maintains a very short execution time even for this dataset. The t-test still shows a significant difference in several benchmarks (all but 12), but the difference in practice is greatly reduced: for example, for LARGE, the proportion between the two configurations for **`atax`** was 1.54×, but for EXTRALARGE it is 1.01×. As such, with NUMA

turned on, we can be confident that execution time discrepancies across the different configurations are tied to physical NUMA topologies rather than cache effects.

Figure 6.6 shows the execution times for the EXTRALARGE dataset with NUMA enabled. The "Balanced" configurations refer to the use of automatic NUMA balancing. "Sequential" and "Parallel" refer to whether the data initialisation step was parallelised. Each benchmark was executed 20 times.

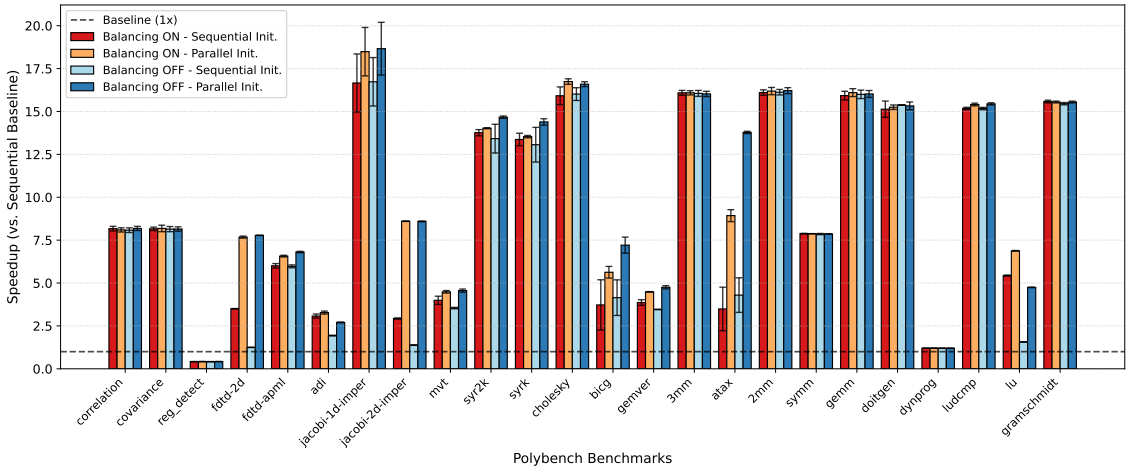

Figure 6.6: Average speedup of PolyBench under different configurations when NUMA is turned on (EXTRALARGE).

Nearly all stencil benchmarks exhibit distinct behaviours between configurations. Because they are $O(n^2)$ algorithms with low arithmetic intensity, they are inherently memory-bound:

- **`fdtd-2d`** and **`jacobi-2d-imper`** show the greatest gaps between sequential and parallel initialisations. Unlike most benchmarks, they also display a notable performance gap between the two automatic NUMA balancing configurations. This can be attributed to their short overall runtimes (roughly 1 second), which amplify the latency overhead of the Operating System (OS)-level balancing system and provide it with insufficient time to scan the processes' memory space and perform data migrations.

- **`adi`** performs slightly better with automatic NUMA balancing, even when utilizing parallel initialisation. This may be because this benchmark's algorithm is structurally split into two phases that traverse the grid in opposing dimensions, allowing the dynamic balancing system to migrate data as the program alternates between access patterns.

- **`fdtd-apml`** and **`jacobi-1d-imper`** are less sensitive to these configurations. The former contains significantly more floating-point operations per iteration than **`fdtd-2d`**, increasing arithmetic intensity, while the latter uses exclusively one-dimensional arrays that are less prone to becoming memory bottlenecks.

The **`atax`** and **`bicg`** benchmarks allocate one large 100000×100000 matrix each, exhibiting a substantial performance leap when the first touch policy is harnessed. As established in Section

6.3, while other $O(n^2)$ kernels allocate similarly large matrices, benchmarks such as **`mvt`** fail to benefit from the initial data placement due to conflicting inner loop access patterns.

Finally, **`lu`** benefits in particular from dynamic automatic balancing. Due to its triangular loop bounds, the mapping from data pages to active threads that use them does not remain constant throughout the execution. A dynamic balancing system accommodates these shifting workloads at runtime. Once again, while **`cholesky`** shares a similar triangular iteration space, **`lu`** writes to a matrix heavily in its innermost loop, whereas **`cholesky`**'s innermost loop is read-only. Consequently, **`lu`** generates significantly more cache coherence traffic and memory invalidations whose impact can be minimised by correctly managing the physical data placement.

6.5 Baseline Clarifications

The speedups presented so far are relative to a sequential baseline measured with the same NUMA BIOS setting as the parallel version. This means that the speedups from different figures may not be directly comparable. The runtimes are very similar for the most part, with a few cases that differ more greatly depending on the presence of NUMA. These differences are shown in Figure 6.7. When comparing the runtimes themselves, there are cases in which a parallelised version performs worse with NUMA enabled, even with a parallelised data initialisation step. We believe that this might not have been the case in a larger NUMA system, since the "average" penalty caused by interleaving data pages scales with the number of NUMA domains (for example, if there were 4 domains, only approximately 25% of the data needed by a thread would be found in its local domain).

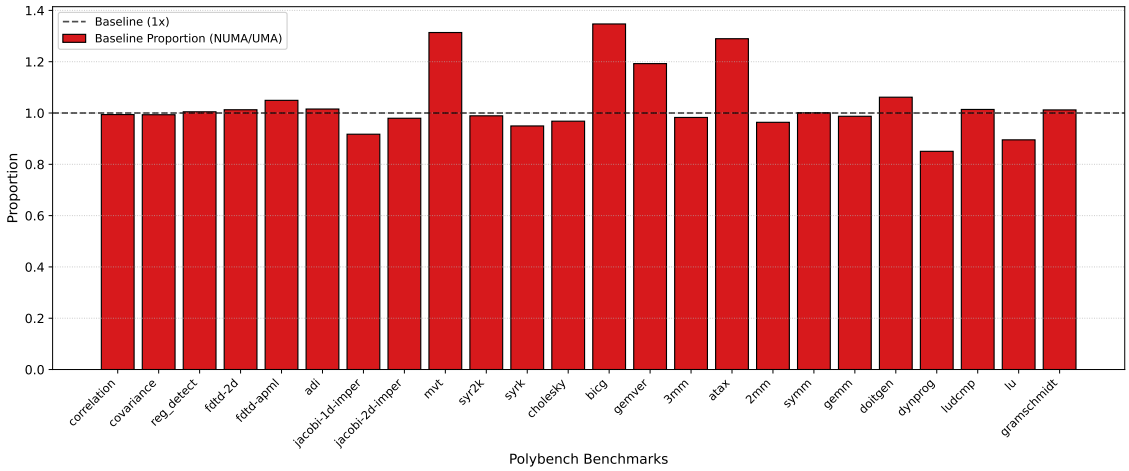

Figure 6.7: Sequential baseline comparison across NUMA and UMA configurations, or "speedup" of the NUMA baseline relative to the UMA baseline (EXTRALARGE).

6.6 NAS Parallel Benchmarks (CG)

The execution section of CG's main subroutine contains 49 lines of code. Within this subroutine, Metafor-AutoPar successfully parallelised four outer loops (corresponding to the setup and verification phases of the algorithm) and five nested loops located within the main computational loop.

When applying our approach to the CG benchmark, we obtain near-perfect linear speedups on select configurations when automatic NUMA balancing is enabled. Predictably, this scaling degrades when NUMA balancing is turned off, mirroring the bandwidth-bound behaviour observed in similar PolyBench algorithms.

```
1 do j = 1, nrows
2  do k = rowstr(j), rowstr(j+1)-1
3      a(k) = 0.d0
4      colidx(k) = 0
5  enddo
6  nzloc(j) = 0
7 enddo
```

Figure 6.8: One of the data initialisation loops in the CG benchmark.

However, unlike the PolyBench examples, CG's data initialisation step is not embarrassingly parallel from a dependence analysis standpoint. Consider the loop nest shown in Figure 6.8. To match the main algorithm's access pattern, the outer loop must be parallelised. The obstacle lies in the inner loop bounds: `rowstr(j)` and `rowstr(j+1)-1` are data-dependent, requiring deeper analysis passes to prove that the memory ranges assigned to `k` never overlap across parallel iterations. Cetus [7] has some support for this kind of situation, using what the authors refer to as "subscripted subscript analysis".

We evaluated three configurations with differing levels of parallelised initialisation: (i) purely sequential initialisation, (ii) partial parallelisation (including only the regions successfully parallelised by Metafor-AutoPar), and (iii) manual adjustments (combining the automated transformations with a manual parallelisation of the loop shown in Figure 6.8). Each configuration was executed 6 times with and without automatic NUMA balancing. We will be mostly uninterested in Classes S, W and A, whose execution times are a fraction of a second, and therefore are not as relevant as larger problem sizes for studying the scaling of parallel execution. The resulting speedups are presented in Figure 6.9.

Crucially, the subsequent initialisation phase for the non-zero elements of array `a` (which represents a sparse matrix) is not trivially parallelizable and therefore not considered. As a result, in the third configuration, the initialisation of `a` becomes fragmented between a parallel component and a sequential component. This split fragments the initial placement of the data pages. This data scattering may explain why the manually adjusted configuration performs poorly for some problem sizes (Classes B and C). This is substantiated by the fact that the automatic NUMA

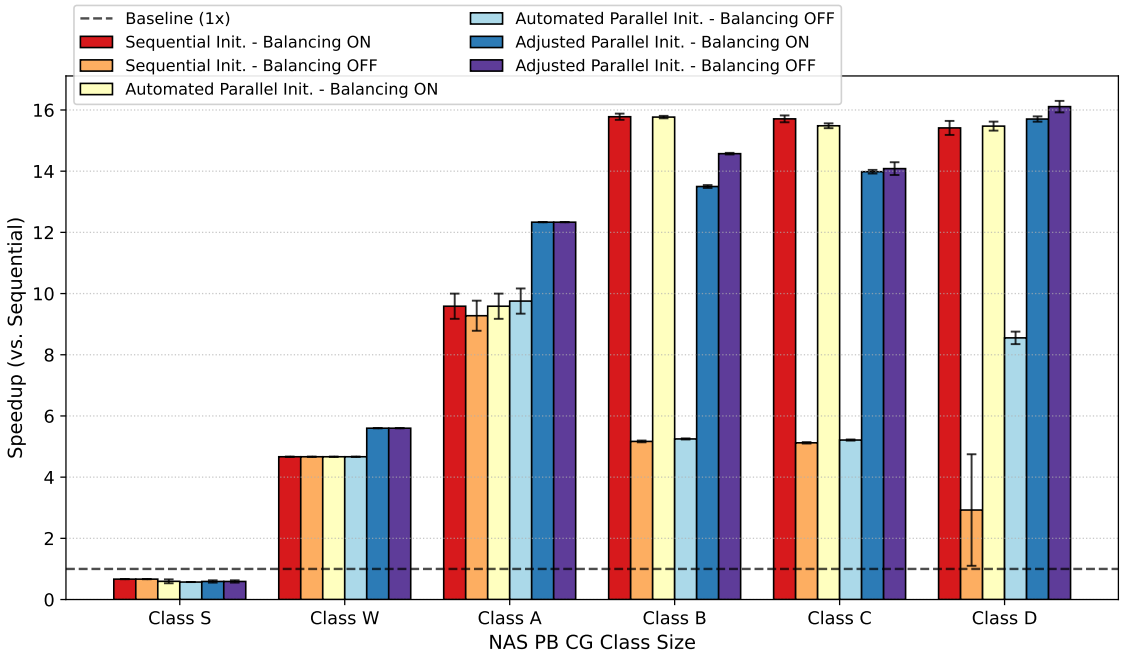

Figure 6.9: Average speedup of CG under different NUMA balancing configurations.

balancing system is unable to bring the performance up to the level of the other configurations with balancing, even causing a slight further degradation. We believe that the kernel-level balancing system is unable to efficiently manage the placement of memory pages that have been split across sockets during different initialisation phases.

Other configurations without this fragmented initialisation display near-perfect linear speedups with balancing enabled. This indicates that in the absence of an automatically parallelizable initialisation step, kernel-level NUMA balancing can serve as a sufficient fallback solution. Additionally, it reveals that it can be preferable to completely forgo parallel initialisation if it is only partially achievable or misaligned with subsequent modifications.

Finally, examining the scaling across problem sizes reveals a behavioural shift between different configurations. Up to Class A, the problem's memory footprint ($\approx$ 27 MiB) is small enough to fit within the shared L3 cache, similarly to what was observed in PolyBench. From Class B onward, the dataset overflows the cache ($\approx$ 193 MiB), making performance more dependent on the underlying memory topology. In the case of Class D specifically, the volume of data and the long execution time appear to amortize the memory-related concerns expressed for classes B and C.

6.6.1 Comparison with Manually Parallelised Benchmark

Figure 6.10 compares our approach (using one of the configurations shown previously) to the manually parallelised version of CG included in the benchmarks. We can see that the manually parallelised version does not seem to benefit greatly from automatic NUMA balancing, which indicates that it is parallelised in a way that correctly handles data placement. Class A is somewhat

of an outlier, though it may be due to some cache effect, since its total runtime is roughly one tenth of a second. A similar pattern could also be observed in the previous section, for the manually adjusted configuration (Figure 6.9).

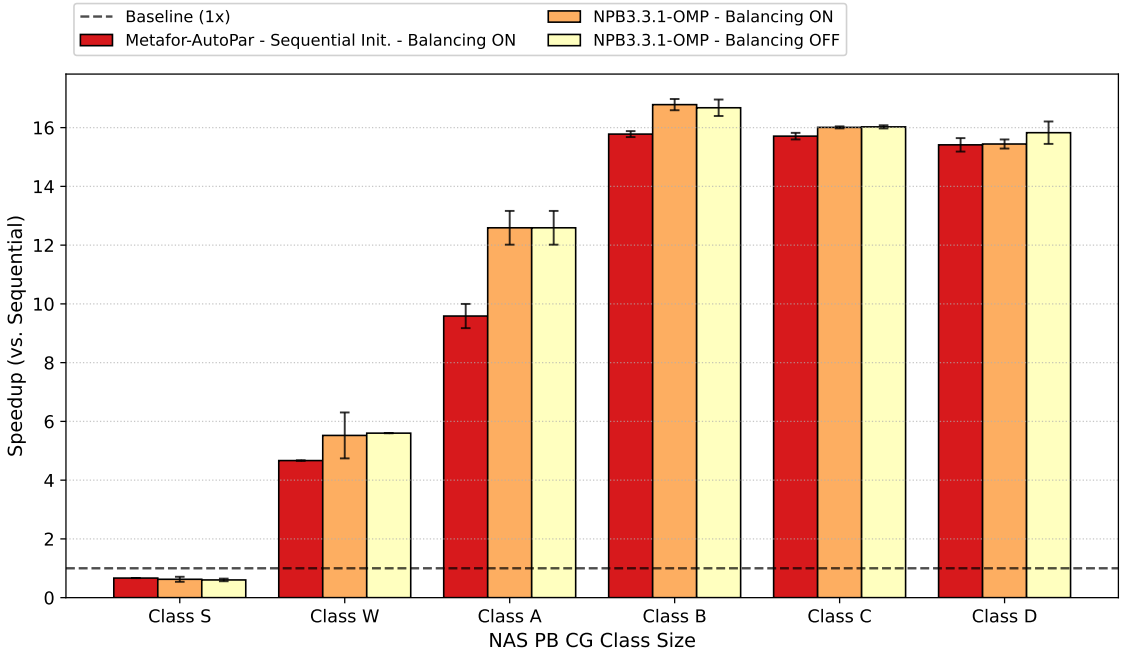

Figure 6.10: Average speedup of CG's manually parallelised implementation compared to Metafor-AutoPar.

6.7 Comparison with Other Approaches

Finding a comparable optimisation tool compatible with Fortran against which we could evaluate our results was challenging. The ones that are still maintained focus on C/C++ programs; several of the ones that support Fortran are not publicly available. We attempted to reach out to the authors of the PolyOpt [45] compilers, for which a Fortran version has been developed, via their mailing list, but did not receive a response.

6.7.1 Polly

Since we are compiling with LLVM Flang, we can easily use Polly, the LLVM Project's polyhedral optimiser. Polly's usage guide refers to its usage alongside Clang, and there is a lack of literature documenting its use with Flang, but since Polly acts on the LLVM-IR level and not on the source code, it is, in principle, fully compatible with Flang. In effect, the results we obtained when compiling PolyBench with Polly enabled did not affect the benchmarks' output and closely match the published results for the C version with regards to which benchmarks display the best performance gains.

When comparing our parallelised runtimes with those obtained via Polly, the results are very diverse. Among the cases in which better runtimes were achieved via Polly, the scale of how better they are varies. Additionally, there are benchmarks that were parallelised by our approach that Polly seems to not have transformed. The results are shown in Figure 6.11, which compares execution times between both our parallelised initialisation configurations (values taken from results shown in previous figures) and Polly with automatic NUMA balancing on or off. The benchmarks compiled with Polly were each executed 5 times. We can observe that Polly generally performs better with balancing enabled, although the difference is only noticeable in a few benchmarks.

Polly supports the generation of vectorised code, but to keep the comparison focused on thread-level parallelism and since Polly's published PolyBench results seem quite inconsistent when vectorisation is enabled, we decided not to use it: when both thread-level parallelism and vectorisation are enabled, multiple benchmarks perform worse than when only thread-level parallelism is enabled, while a couple perform considerably better.

Additionally, our evaluation hardware isn't ideal for the evaluation of modern vectorisation, since its CPUs lack support for recent vector extensions such as AVX512.

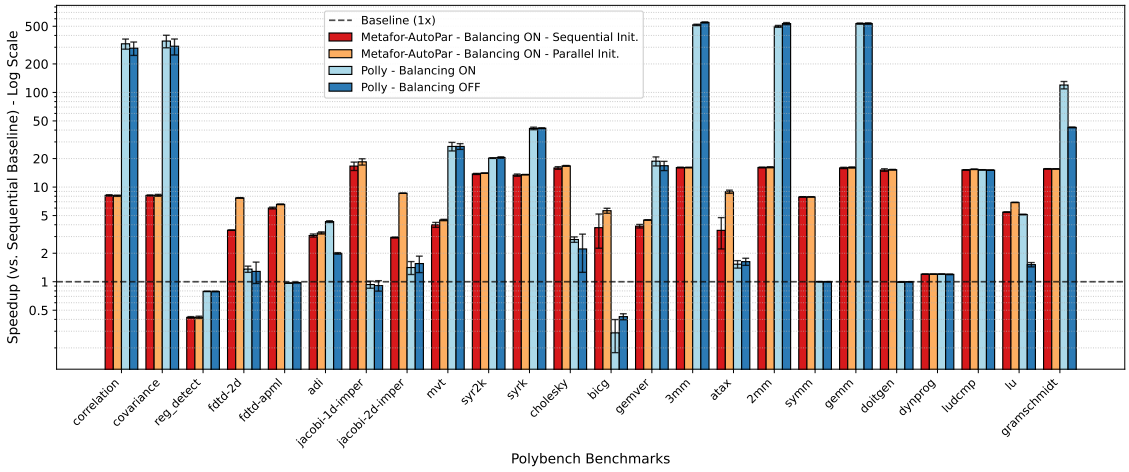

Figure 6.11: Speedup of PolyBench comparing Polly and Metafor-AutoPar (EXTRALARGE).

6.7.1.1 Matrix Multiplications and Other $O(n^3)$ Algorithms

A closer look at the benchmarks in which Polly showed significantly better results than our approach (**`covariance`**, **`correlation`**, **`gemm`**, **`2mm`**, **`3mm`**, **`gramschmidt`**), as shown in Figure 6.11, reveals that they all contain $O(n^3)$ loops that perform some sort of matrix-matrix operation. As stated previously, these types of computation reuse a lot of data over several operations, and as such, they are the perfect application for loop tiling, one of the optimisations that Polly applies. Loop tiling consists in taking a nested loop structure and adding additional loops that split the processing of the data into (typically) rectangular tiles. With the right tile size, this can greatly improve data locality for the L1 and L2 caches.

However, the difference in performance is not entirely due to the presence of loop tiling. The data access pattern for PolyBench's matrix multiplications is not optimised for data locality: the memory accesses required in each subsequent iteration of the inner loop are strided instead of contiguous. To solve this, the two innermost loops must be interchanged. Strictly speaking, Polly does not perform a syntactic loop interchange; rather, it computes an optimal affine schedule within the polyhedral model that maximizes spatial locality. Nevertheless, in the context of these benchmarks, the functional outcome of this mathematical rescheduling is equivalent to an interchange. We manually performed the necessary loop interchange on the **`gemm`** benchmark and parallelised it with our approach to get an estimation of how much of Polly's gains could be attributed to loop interchange (Table 6.4). The runtime for this version of the benchmark is much closer to Polly's, though it is still over three times slower.

Approach	**Execution Time (seconds)**
Polly	2.02
Metafor-AutoPar	6.85
Metafor-AutoPar (no interchange)	66.99

Table 6.4: Average `gemm` execution times comparing Polly to our approach when applied to a manually adjusted benchmark.

6.7.1.2 Performance Degradations

There are three cases (**`reg_detect`**, **`jacobi-1d-imper`**, **`bicg`**) for which Polly's version of the benchmark performs worse than even the original sequential version. Polly includes heuristics to determine the profitability of optimizing regions of the program. The results shown in Polly's original publication [20] were taken before this was implemented, which can explain why we have fewer cases of performance degradation, comparing our Figure 6.11 with Figure 7 of Polly's paper.

We are unsure of the origin of these performance degradations. We believe that it may be due to the profitability model making an incorrect decision, or some specific quirk of the LLVM produced by Flang that creates some small incompatibility with Polly.

6.7.1.3 Stencils

Among the stencils, our approach achieved better execution times than Polly for most cases, with two exceptions. The simplest case is **`seidel-2d`**, which our approach and Polly failed to parallelize. **`seidel-2d`**'s calculations are performed entirely in-place, which creates several hard to resolve dependencies.

In the **`jacobi`** and **`fdtd`** families of benchmarks, our approach was the fastest. We believe that Polly applied skewing transformations to the iteration space to produce its tiled polyhedral model, leading to complex, non-rectangular loop bounds. Since these benchmarks do not contain arithmetically intensive $O(n^3)$ kernels such as those discussed in Section 6.7.1.1, data reuse is much less frequent and there is little to no benefit in tiling. Such cases are acknowledged in

Polly's original publication [20]. Thus, simply parallelizing the inner loops with no dependencies while maintaining the original rectangular loop bounds proved to be a better solution in these cases.

However, for **adi**, Polly achieved a somewhat better runtime. The structure of the data dependencies in this case is different. For the **jacobi** and **fdtd** families, each point of the grid depended on a few neighboring elements from the previous timestep, which enables the skewing that Polly probably attempted. In **adi**, the entire grid must be processed at a given timestep before the next one can begin. As such, Polly may have been blocked from performing the same transformations that were applied to other cases, therefore being limited to working on the inner loops of the kernel.

6.7.1.4 Other Notable Cases

atax and **bicg** are two other benchmarks that perform better under our approach. Notably, these are the only ones in which an **array reduction** is used. Polly's version of these benchmarks does not employ reductions, which was verified by looking for calls to OpenMP reduction procedures in the resulting LLVM-IR. It must have then resorted to some form of atomic update or the serialisation of part of the program, eliminating some of the potential for parallelism.

cholesky is one of the benchmarks that has a triangular iteration domain. Forcing tiling onto these programs results in complex loop bound expressions, resulting in some control flow overhead, as for the stencil benchmarks. As such, our approach outperforms Polly. However, this is not universal across structurally similar benchmarks. In the case of **lu**, there is no substantial difference in performance, although, as discussed previously, **lu** and **cholesky** differ in other aspects.

6.7.1.5 CG

Polly is limited in the complexity of the programs to which it can be applied. CG contains structures which are, essentially, entirely absent in PolyBench, such as indirect array accessing (e.g., `A[B[i]]`). It was unable to effectively parallelize the benchmark, as runtimes were matched with that of the original sequential version.

The optimisation report that can be obtained through the use of Flang's `-fsave-optimisation -record` flag showed error messages such as "Failed to derive an affine function from the loop bounds" and "The array subscript of [...] is not affine". In our approach, as discussed previously, the loop bounds are replaced by surrogate expressions of an unknown value, and since the array that produced the second error message is read-only with regards to the loop nest, it is ignored.

6.7.2 LLM-based Parallelisation

As discussed previously, there are parallelisation approaches based on fine-tuned transformer models. However, their focus is on C/C++, so in order to compare our approach with one focused on

artificial intelligence, we made use of GPT 5.4, from OpenAI. The model was prompted to produce parallelised versions of the PolyBench benchmarks. Specifically, the prompt was:

- "Parallelize all Fortran files (extension .F90) in project [...] using OpenMP directives. You must not change the original code of any Fortran file, you are only allowed to add OpenMP directives."

For the most part, the resulting programs contained the exact directives that our approach inserted, with some minor differences that do not impact performance, such as explicitly specifying default values for clauses we omitted, or not merging parallel loops into one large parallel region. However, there were cases in which the models produced different code. We evaluated the effect of some of those differences. As expected with the use of LLMs, different runs of the same query produced slightly different results.

All comparisons in this section are made using the EXTRALARGE dataset, with parallelised initialisation and automatic NUMA balancing enabled, with the average execution times of 20 repetitions.

6.7.2.1 Atomic Updates

In **`atax`** and **`bicg`**, one of the models generated a version of the benchmark that is parallelised via an atomic update, instead of an array reduction. As shown in Table 6.5, the execution time of the resulting program was slower. Using an atomic update in the benchmark's main loop when there is no substantial amount of work besides what is contained in the atomic update and when the number of threads is small compared to the number of iterations, the program is effectively serialised due to lock contention.

Benchmark	Metafor-AutoPar	w/ `atomic update`
`atax`	2.57	9.96
`bicg`	3.33	16.51

Table 6.5: Average execution times (in seconds) of two benchmarks, comparing atomic updates and array reductions.

The use of an atomic update does have one upside, which is memory usage. For an array reduction, each thread must create its local copy of the array. For **`atax`** and **`bicg`**'s largest configurations, this was not prohibitive in our experimental setup, but there is nonetheless a memory cost associated with array reductions that is not present for an atomic update.

6.7.2.2 Use of the `collapse` clause

The OpenMP **`collapse`** clause, which has one positive integer argument, specifies the "depth of the canonical loop nest to which to apply the semantics of the directive". For example, in a two-dimensional loop nest (i and j), **`collapse(2)`** will make the OpenMP runtime consider (i,j) iterations when assigning iterations to threads. If the clause was absent, the existence of the j-loop

would be irrelevant to the OpenMP runtime, as only the first dimension of the iteration space is considered for scheduling.

In workloads with heavy load imbalance between iterations, the `collapse` clause can be helpful since it creates a fine-grained representation of the iteration space that allows further division of the work among the threads, especially when paired with dynamic scheduling. It can also be useful when the number of iterations is very small compared to that of the inner loop, particularly when the outer loop has fewer iterations than there are active threads.

Some of the LLM generated versions included collapse clauses. The algorithms in PolyBench are overwhelmingly "regular", with the same number and kind of operations performed at each iteration. As such, we did not expect the use of the collapse to make a significant difference in execution time, which we verified. However, in NUMA-sensitive benchmarks, the use of collapse seems to have a negative impact. As discussed previously, for stencils that make use of adjacent neighbors from each grid element, remote memory accesses occur only near the border between the two NUMA domains because the grid is evenly split between them. The scheduling that OpenMP applies to the fine-grained iteration space may not result in such a clean distribution. Execution times for some of the benchmarks in which the `collapse` clause was found are presented in Table 6.6. These results were measured with NUMA balancing enabled.

Benchmark	Metafor-AutoPar	w/ `collapse`
`2mm`	134.25	134.77
`fdtd-2d`	1.34	2.20

Table 6.6: Average execution times (in seconds) of two benchmarks, showing the effects of the `collapse` clause.

6.7.3 Stencil Transformations

Among the benchmarks we have presented so far, our "doacross" transformations are directly applicable to `seidel-2d` from PolyBench. As discussed previously, conventional worksharing OpenMP loop constructs are insufficient for this case. With automatic "doacross" transformations enabled, we were able to obtain a parallelised version of the benchmark. However, it performs worse than the sequential version, due to the pitfalls described in Section 4.3.3. The execution times are listed in Table 6.7.

Version	Original Loop	w/ Dummy Computation
Sequential	25.59	432.36
Parallel	38.23	71.62

Table 6.7: Average `seidel-2d` "doacross" and sequential execution times (in seconds).

To determine whether this transformation had the potential to provide positive results if the inner loop's arithmetic intensity were higher, we inserted dummy computations in the benchmark

as shown in Figure 6.12. These were sufficient to mask the OpenMP overhead, as the parallel version became faster than the sequential version (Table 6.7).

```
1 dummy = 0.0d0
2 DO k = 1, 10
3  dummy = dummy + sin(dble(i * j * k))
4 END DO
5
6 a(j, i) = (a(j - 1, i - 1) + a(j, i - 1) + a(j + 1, i - 1) + a(j - 1, i) + a(j, i)
      + a(j + 1, i) + a(j - 1, i + 1) + a(j, i + 1) + a(j + 1, i + 1)) / 9.0d0
7 a(j, i) = a(j, i) + (dummy * 0.000000001d0)
```

Figure 6.12: Altered contents of `seidel-2d`'s inner loop.

6.8 Summary

The results produced by our approach demonstrate the viability of automated source-to-source parallelisation for Fortran programs. Our approach employs high-level AST manipulation while borrowing some concepts from polyhedral compilation models to perform dependence analysis via the Integer Set Library. We observed near-perfect linear speedups in several of the tested benchmarks, and unresolvable dependencies were correctly identified to maintain the original program's behaviour.

However, our evaluation highlighted that thread-level parallelism is only one facet of program optimisation; by itself, it does not resolve underlying memory bottlenecks present in the original program. This was shown in the comparison with LLVM Polly. For dense, arithmetically intense $O(n^3)$ kernels, Polly's polyhedral restructuring of loop iteration spaces optimised for cache locality significantly outperforms our approach. Conversely, we observed that advanced polyhedral scheduling is not universally optimal, as some algorithms do not stand to benefit from complex loop transformations such as tiling. For some $O(n^2)$ kernels and stencils, our approach outperformed Polly by preserving the original loop structure and employing some OpenMP features that Polly does not use, such as array reductions.

Our evaluation on a dual-socket NUMA architecture allowed us to provide insights into how automated parallelisation and NUMA environments can interact. While leveraging the First Touch Policy by parallelizing the data initialisation step successfully mitigated memory access costs for most rectangular loop structures found in the tested benchmarks, we observed that statically controlling the initial placement of data pages is not sufficient when dynamic iteration spaces are present. In triangular loop structures, for example, the thread-to-data relationship shifts over time, creating NUMA overhead, especially in write-intensive benchmarks due to the high amount of cache coherence traffic. In these scenarios, static source code transformations can be coupled with a system-level automatic NUMA balancing system that can dynamically migrate data pages during the program's execution.

These results allow us to provide some answers for our initial research questions:

Answer to RQ1 (To what extent can modern OpenMP directives, such as `ordered` and `depend`, be leveraged by an automated tool?): Modern OpenMP synchronisation directives (e.g., `ordered` and `depend`) can be successfully injected by automated tools to parallelize loops with complex dependencies without changing code layout. However, their high runtime overhead means that they are only practically viable in workloads with high arithmetic intensity. For the benchmark that was evaluated, positive speedups were only observed after synthetically masking this overhead.

Answer to RQ2 (How does conservative, source-level dependence analysis with minimal code transformation compare to full polyhedral optimisation when automatically parallelizing scientific Fortran workloads?): Conservative source-level parallelisation without code transformations cannot match full polyhedral optimisation on arithmetically intensive $O(n^3)$ algorithms that depend on cache-tiling techniques. However, it can be more robust when dealing with irregular, non-affine code structures, and it yields superior performance on some $O(n^2)$ kernels and regular stencils by preserving simple loop geometries and avoiding polyhedral control-flow overhead.

Answer to RQ3 (How do AST-level automated parallelisation strategies interact with underlying physical hardware topologies, specifically Non Uniform Memory Access (NUMA) architectures?): AST-level optimisation strategies interact with NUMA hardware by parallelizing initialisation loops to exploit the First Touch Policy, mitigating memory access latency on regular, static grids. However, at least on a dual-socket architecture, static source-level placement is insufficient for dynamic iteration spaces where thread-to-data affinities shift over time; such workloads can be paired with dynamic, runtime page migration systems.

Chapter 7

Conclusions

This dissertation presented Metafor-AutoPar, an automatic source-to-source parallelisation approach targeting Fortran programs, built on top of the LARA framework and the LLVM Flang frontend. Its current focus is on the insertion of OpenMP worksharing loop directives.

A conservative dependence analysis performed by integrating principles from polyhedral compilation allows us to only insert parallelisation directives on a loop nest when it is mathematically safe (insofar as our polyhedral representation of the loop correctly reflects the semantics of the program). A comparison with Polly, a full polyhedral optimisation approach directly targeting LLVM's intermediate representation, showed that while sophisticated polyhedral compilation can greatly optimize arithmetically dense loop nests, straightforward parallelisation that maintains the original loop nest's structure has its merits for other types of benchmarks, such as some stencils.

Our work also provides some valuable insights into how automatic parallelisation approaches can adapt to a NUMA environment. Parallelizing the data initialisation step can provide a static data placement pattern that fits the algorithm in question via the first touch policy, minimizing memory access costs throughout the execution of the program. However, irregular iteration spaces (e.g., triangular loop nests) can cause the data/thread relation to shift over time, creating memory access bottlenecks across the NUMA domains of the system, especially in write-heavy workloads. To alleviate this, our approach can be paired with a dynamic automatic NUMA balancing system that moves data pages closer to the threads that require them as the program executes, such as the kernel-level system found in Linux.

Additionally, we demonstrate a mechanism for automatically applying OpenMP 4.5's `ordered/depend` "doacross" constructs and clauses to Fortran programs utilizing the information provided by data dependence distance vectors. While this approach proved unsuitable for the PolyBench benchmark it was most directly applicable to (`seidel-2d`) due to the large synchronisation overhead that "doacross" parallelism can introduce, artificially increasing its arithmetic density showed the viability of this type of parallelism. Other practical use cases [26] show that it can match manually parallelised versions of programs.

7.1 Limitations and Future Work

One clear limitation of our current implementation is limited support for several complex code structures. Certain features that are present in several source-to-source optimisers, such as induction variable recognition and substitution, were not prioritised as they were not required to process the selected evaluation benchmarks. Others, such as the lack of inter-procedural analysis, stem from the recent inception of Metafor, which currently lacks the features present in other LARA compilers, such as Clava, that facilitate global AST traversal.

Another avenue for future work involves the integration of syntactic loop transformations into the dependence analysis pipeline. As demonstrated by our brief "ablation study" when exploring the source of the difference in performance between Polly and our approach for the `gemm` benchmark, a simple transformation, such as exchanging two loops, can go a long way in optimizing a program. Our use of the Integer Set Library also means that there could be an attempt to further bridge the gap between our approach and full polyhedral compilation by leaning further into the information that the `isl` can provide for code transformations.

Furthermore, implementing a heuristic-based cost model can be explored to evaluate the profitability of parallelisation. Currently, Metafor-AutoPar operates on an eager parallelisation strategy, injecting OpenMP directives into any loop nest deemed mathematically safe. However, as demonstrated by cases of performance degradation, both in standard and "doacross" parallelism, the overhead of thread synchronisation, fork-join latencies, and distant memory accesses can overshadow parallel gains. Integrating an automated cost model, similarly to what approaches such as Polly have implemented, would allow Metafor-AutoPar to estimate a loop's arithmetic intensity and memory footprint. This would be used to determine when to preserve sequential execution in case the estimated parallel overhead exceeds the "cost" of the loop body.

The module that extracts the polyhedral representation of a loop nest serves a purpose similar to the Polyhedral Extraction Tool [60] for C programs. However, the latter has more extensive support for different programs. For this work, we did not prioritize features that would have no effect on the parallelisation of the benchmarks we evaluated. Some simple aspects, such as recognizing a broader range of operations that the `isl` accepts (such as minimum/maximum operations), are not challenging to implement. Limitations that are not as trivial to address include:

- Support for boolean condition expressions that include negations ("!");
- Support for do-loops with strides that are not 1 and infinite loops;
- Broader support for if/else constructs;
- Broader support for data-dependent expressions;
- Integration of information pertaining to the minimum/maximum values of a given datatype into the analysis.

Finally, because the isolation and analysis of parallelizable loop nests is not inherently tied to any one specific characteristic of the OpenMP `DO` construct, this work could be extended by interchanging it with other constructs, enabling the use of task-based parallelism via the `TASKLOOP` construct or even offloading for heterogeneous architectures via the `TARGET` construct without manual programmer intervention. More specialised offloaded code (e.g., Compute Unified Device Architecture (CUDA) [38] or SYCL [47]) could also be explored, though it may require more in-depth code transformations.

References

[1] Laurence Kedward et al. “The State of Fortran”. In: *Computing in Science & Engineering* (2022). https://arxiv.org/abs/2203.15110.

[2] T. A. Johnson et al. “Experiences in Using Cetus for Source-to-Source Transformations”. In: *Languages and Compilers for High Performance Computing* (2004). https://link.springer.com/content/pdf/10.1007/11532378.pdf#page=10.

[3] Randy Allen and Ken Kennedy. *Optimizing Compilers for Modern Architectures: A Dependence-based Approach*. Morgan Kaufmann, 2001.

[4] Mehdi Amini et al. “Par4All: From Convex Array Regions to Heterogeneous Computing”. In: *Proceedings of the 2nd International Workshop on Polyhedral Compilation Techniques (IMPACT)*. Paris, France, 2012. URL: https://par4all.github.io/documentation.html.

[5] Hamid Arabnejad et al. “Source-to-source compilation targeting OpenMP-based automatic parallelization of C applications”. In: *The Journal of Supercomputing* 76 (Sept. 2020). DOI: 10.1007/s11227-019-03109-9.

[6] David Bailey et al. “The NAS parallel benchmarks”. In: *The International Journal of Supercomputing Applications* (Aug. 1993). DOI: 10.2172/983318.

[7] P. Barakhshan and R. Eigenmann. “iCetus: A Semi-automatic Parallel Programming Assistant”. In: Lecture Notes in Computer Science. Vol. 13181 LNCS. 2022, pp. 18–32. DOI: 10.1007/978-3-030-99372-6_2. URL: https://www.scopus.com/inward/record.uri?eid=2-s2.0-85127621348&doi=10.1007%2F978-3-030-99372-6_2&partnerID=40&md5=03f031cc476682c2751d2c2616cc15c5.

[8] Guillaume Bertholon et al. “Interactive Source-to-Source Optimizations Validated using Static Resource Analysis”. In: *Proceedings of the 13th ACM SIGPLAN International Workshop on the State Of the Art in Program Analysis*. SOAP 2024. event-place: Copenhagen, Denmark. New York, NY, USA: Association for Computing Machinery, 2024, pp. 26–34. ISBN: 979-8-4007-0621-9. DOI: 10.1145/3652588.3663320. URL: https://doi.org/10.1145/3652588.3663320.

[9] A. Bhosale et al. “Automatic and Interactive Program Parallelization Using the Cetus Source to Source Compiler Infrastructure v2.0”. In: *Electronics (Switzerland)* 11.5 (2022). DOI: 10.3390/electronics11050809. URL: https://www.scopus.com/inward/record.uri?eid=2-s2.0-85125996772&doi=10.3390%2Felectronics11050809&partnerID=40&md5=a8b5013b2c5f0dd2f368cce1f9b3cbf3.

[10] J. Bispo and J.M.P. Cardoso. “Clava: C/C++ source-to-source compilation using LARA”. In: *SoftwareX* 12 (2020). DOI: 10.1016/j.softx.2020.100565. URL: https://www.scopus.com/inward/record.uri?eid=2-s2.0-85088268271&doi=10.1016%2Fj.softx.2020.100565&partnerID=40.

[11] J. Bispo, N. Paulino, and L.M. Sousa. "Challenges and Opportunities in C/C++ Source-To-Source Compilation". In: OpenAccess Series in Informatics. Vol. 107. 2023. DOI: 10.4230/OASIcs.PARMA-DITAM.2023.2. URL: https://www.scopus.com/inward/record.uri?eid=2-s2.0-85150358915&doi=10.4230%2FOASIcs.PARMA-DITAM.2023.2&partnerID=40.

[12] G. Cordasco et al. "Toward a domain-specific language for scientific workflow-based applications on multicloud system". In: *Concurrency and Computation: Practice and Experience* 33.18 (2021). DOI: 10.1002/cpe.5802. URL: https://www.scopus.com/inward/record.uri?eid=2-s2.0-85084203343&doi=10.1002%2Fcpe.5802&partnerID=40&md5=12c4170fc1756fb3d0a4a045674ab9fd.

[13] T. Cramer et al. "OpenMP target device offloading for the SX-Aurora TSUBASA vector engine". In: Lecture Notes in Computer Science. Vol. 12043 LNCS. 2020, pp. 237–249. DOI: 10.1007/978-3-030-43229-4_21. URL: https://www.scopus.com/inward/record.uri?eid=2-s2.0-85083982767&doi=10.1007%2F978-3-030-43229-4_21&partnerID=40&md5=bddac3c7940f4613843661aa05e73d2b.

[14] Chirag Dave et al. "Cetus: A Source-to-Source Compiler Infrastructure for Multicores". In: *Computer* 42.12 (2009), pp. 36–42. DOI: 10.1109/MC.2009.385.

[15] Andi Drebes et al. "Scalable task parallelism for NUMA: A uniform abstraction for coordinated scheduling and memory management". In: *2016 International Conference on Parallel Architecture and Compilation Techniques (PACT)*. 2016, pp. 125–137. DOI: 10.1145/2967938.2967946.

[16] L.G. Faé, R.B. Hoffmann, and D. Griebler. "Source-to-Source Code Transformation on Rust for High-Level Stream Parallelism". In: 2023, pp. 41–49. DOI: 10.1145/3624309.3624320. URL: https://www.scopus.com/inward/record.uri?eid=2-s2.0-85180131541&doi=10.1145%2F3624309.3624320&partnerID=40&md5=03a2c9ab40319430af2666ed82d4202b.

[17] P. Flynn, X. Yi, and Y. Yan. "Exploring Source-to-Source Compiler Transformation of OpenMP SIMD Constructs for Intel AVX and Arm SVE Vector Architectures". In: 2022, pp. 11–20. DOI: 10.1145/3528425.3529100. URL: https://www.scopus.com/inward/record.uri?eid=2-s2.0-85129653685&doi=10.1145%2F3528425.3529100&partnerID=40&md5=94c4af9057392f0263003a9218f49382.

[18] A. Freitas, T. Baptista, and P.R. Rangel Henriques. "Bridging Language Barriers: A Comparative Review and Empirical Evaluation of Source-To-Source Transpilers". In: OpenAccess Series in Informatics. Vol. 135. 2025. DOI: 10.4230/OASIcs.SLATE.2025.11. URL: https://www.scopus.com/inward/record.uri?eid=2-s2.0-105013130072&doi=10.4230%2FOASIcs.SLATE.2025.11&partnerID=40&md5=bfbf362372daac50dc546de1fd2f8512.

[19] Y. Fridman, Y. Goren, and G. Oren. "From OpenACC to OpenMP5 GPU Offloading: Performance Evaluation on NAS Parallel Benchmarks". In: 2025, pp. 10–18. DOI: 10.1145/3720555.3721989. URL: https://www.scopus.com/inward/record.uri?eid=2-s2.0-105007287141&doi=10.1145%2F3720555.3721989&partnerID=40&md5=f97587f2d36703d3d439d5bb8e936e04.

[20] Tobias Grosser, Armin Groesslinger, and Christian Lengauer. "Polly – Performing polyhedral optimizations on a low-level intermediate representation". In: *Parallel Processing Letters* 22.04 (Dec. 2012). Publisher: World Scientific Publishing Company. DOI: 10.1142/S012962641250010X. URL: https://doi.org/10.1142/S012962641250010X.

[21] R. Harel, Y. Pinter, and G. Oren. "POSTER: Learning to Parallelize in a Shared-Memory Environment with Transformers". In: 2023, pp. 450–452. DOI: 10.1145/3572848.3582565. URL: https://www.scopus.com/inward/record.uri?eid=2-s2.0-85149297649&doi=10.1145%2F3572848.3582565&partnerID=40&md5=d335237fb9d900c93473d3f73ce6a7fb.

[22] R. Harel et al. "Source-to-Source Parallelization Compilers for Scientific Shared-Memory Multi-core and Accelerated Multiprocessing: Analysis, Pitfalls, Enhancement and Potential". In: *International Journal of Parallel Programming* 48.1 (2020), pp. 1–31. DOI: 10.1007/s10766-019-00640-3. URL: https://www.scopus.com/inward/record.uri?eid=2-s2.0-85070320501&doi=10.1007%2Fs10766-019-00640-3&partnerID=40&md5=25affdd8fc71c248c90ff1858536c13c.

[23] Miguel Henriques, João Bispo, and Nuno Paulino. "Using Source-to-Source to Target RISC-V Custom Extensions: UVE Case-Study". In: *Proceedings of the 16th Workshop on Rapid Simulation and Performance Evaluation for Design*. RAPIDO '24. Munich, Germany: Association for Computing Machinery, 2024, pp. 42–50. ISBN: 9798400717918. DOI: 10.1145/3642921.3642930. URL: https://doi.org/10.1145/3642921.3642930.

[24] R.B. Hoffmann et al. "OpenMP as runtime for providing high-level stream parallelism on multi-cores". In: *Journal of Supercomputing* 78.6 (2022), pp. 7655–7676. DOI: 10.1007/s11227-021-04182-9. URL: https://www.scopus.com/inward/record.uri?eid=2-s2.0-85122139066&doi=10.1007%2Fs11227-021-04182-9&partnerID=40&md5=a90bd91c573a8a2f8b492fcc66fa0d9b.

[25] Jos de Jong and contributors. *math.js*. Version 15.2.0. 2026. URL: https://mathjs.org/.

[26] Gabriele Jost and Henry Jin. *OpenMP Doacross Loops Case Study*. SC17 OpenMP Booth Talks, Denver, CO. Presentation slides. Nov. 2017. URL: https://www.openmp.org/wp-content/uploads/SC17-Jost-OpenMP-booth-doacross.pdf.

[27] W. Kelly et al. *The Omega library*. Tech. rep. University of Maryland, Nov. 1996.

[28] Chunhua Liao et al. "Semantic-Aware Automatic Parallelization of Modern Applications Using High-Level Abstractions". In: *International Journal of Parallel Programming* 38 (Oct. 2010), pp. 361–378. DOI: 10.1007/s10766-010-0139-0.

[29] Amanda Liu et al. "Verified tensor-program optimization via high-level scheduling rewrites". In: *Proc. ACM Program. Lang.* 6 (POPL Jan. 2022). Place: New York, NY, USA Publisher: Association for Computing Machinery. DOI: 10.1145/3498717. URL: https://doi.org/10.1145/3498717.

[30] Ramakrishna Manchana. "Java Virtual Machine (JVM): Architecture, Goals, and Tuning Options". In: *International Journal of Scientific Research and Engineering Trends* 1 (May 2015), pp. 42–52. DOI: 10.61137/ijsret.vol.1.issue3.42.

[31] J.N. Matos, J. Bispo, and L.M. Sousa. "A C Subset for Ergonomic Source-to-Source Analyses and Transformations". In: 2024, pp. 1–8. DOI: 10.1145/3642921.3642922. URL: https://www.scopus.com/inward/record.uri?eid=2-s2.0-85190954833&doi=10.1145%2F3642921.3642922&partnerID=40&md5=51a85951c86f030f9f0edbab4626ee74.

[32] Collin McCurdy and Jeffrey Vetter. "Memphis: Finding and fixing NUMA-related performance problems on multi-core platforms". In: Apr. 2010, pp. 87–96. DOI: 10.1109/ISPASS.2010.5452060.

[33] Elliott Mendelson. *Introduction to Mathematical Logic*. 6th. CRC Press, 2015.

[34] R. Milewicz et al. "Negative Perceptions About the Applicability of Source-to-Source Compilers in HPC: A Literature Review". In: Lecture Notes in Computer Science. Vol. 12761 LNCS. 2021, pp. 233–246. DOI: 10.1007/978-3-030-90539-2_16. URL: https://www.scopus.com/inward/record.uri?eid=2-s2.0-85119838083&doi=10.1007%2F978-3-030-90539-2_16&partnerID=40.

[35] A. Mishra, A.M. Malik, and B. Chapman. "Data transfer and reuse analysis tool for gpu-offloading using openmp". In: Lecture Notes in Computer Science. Vol. 12295 LNCS. 2020, pp. 280–294. DOI: 10.1007/978-3-030-58144-2_18. URL: https://www.scopus.com/inward/record.uri?eid=2-s2.0-85091285486&doi=10.1007%2F978-3-030-58144-2_18&partnerID=40.

[36] I. Mosseri et al. "Compar: Optimized multi-compiler for automatic openmp s2s parallelization". In: Lecture Notes in Computer Science. Vol. 12295 LNCS. 2020, pp. 247–262. DOI: 10.1007/978-3-030-58144-2_16. URL: https://www.scopus.com/inward/record.uri?eid=2-s2.0-85091271404&doi=10.1007%2F978-3-030-58144-2_16&partnerID=40&md5=74838a55dcdf6906cc84c878bd439deb.

[37] MPI Forum. *Message Passing Interface (MPI) Forum*. Accessed: 2026-01-13. 2026. URL: https://www.mpi-forum.org/.

[38] John Nickolls et al. "Scalable parallel programming with CUDA". In: *ACM Queue* 6.2 (2008), pp. 40–53.

[39] Robert W. Numrich and John Reid. "Co-array Fortran for parallel programming". In: *SIGPLAN Fortran Forum* 17.2 (Aug. 1998), pp. 1–31. ISSN: 1061-7264. DOI: 10.1145/289918.289920. URL: https://doi.org/10.1145/289918.289920.

[40] OpenMP Architecture Review Board. *OpenMP: The Standard for Parallel Programming*. Accessed: 2026-01-13. 2026. URL: https://www.openmp.org/.

[41] M. Pałkowski and W. Bielecki. "Parallel Tiled Cache and Energy Efficient Code for Zuker's RNA Folding". English. In: vol. 12044 LNCS. 2020, pp. 25–34. DOI: 10.1007/978-3-030-43222-5_3. URL: https://www.scopus.com/inward/record.uri?eid=2-s2.0-85083985626&doi=10.1007%2F978-3-030-43222-5_3&partnerID=40&md5=82b0a83fa6b06ac22cca4645ac54dd68.

[42] P. Pinto et al. "Pegasus: Performance Engineering for Software Applications Targeting HPC Systems". In: *IEEE Transactions on Software Engineering* 48.3 (2022), pp. 732–754. DOI: 10.1109/TSE.2020.3001257. URL: https://www.scopus.com/inward/record.uri?eid=2-s2.0-85086722258&doi=10.1109%2FTSE.2020.3001257&partnerID=40&md5=40fec7e05cb840086bb1031d5f80557f.

[43] Pedro Pinto et al. "Aspect composition for multiple target languages using LARA". In: *Computer Languages, Systems & Structures* 53 (2018), pp. 1–26. ISSN: 1477-8424. DOI: https://doi.org/10.1016/j.cl.2017.12.003. URL: https://www.sciencedirect.com/science/article/pii/S147784241730115X.

[44] Louis-Noël Pouchet. *PolyBench/Fortran: The Polyhedral Benchmark suite*. https://www.cs.colostate.edu/~pouchet/software/polybench/polybench-fortran.html. Accessed: 2026-06-04.

[45] Louis-Noël Pouchet. *Polyopt, a polyhedral optimizer for the rose compiler*. 2011.

[46] Dan Quinlan and Chunhua Liao. "The ROSE source-to-source compiler infrastructure". In: *Cetus users and compiler infrastructure workshop, in conjunction with PACT*. Vol. 2011. Citeseer. 2011, p. 1.

[47] Ruymán Reyes and Victor Lomüller. "SYCL: Single-source C++ accelerator programming". In: *International Conference on Parallel Computing*. 2015. URL: https://api.semanticscholar.org/CorpusID:6979118.

[48] M. Sato and M. Tsuji. "OpenACC Execution Models for Manycore Processor with ARM SVE". In: ACM International Conference Proceeding Series. 2023, pp. 73–77. DOI: 10.1145/3581576.3581617. URL: https://www.scopus.com/inward/record.uri?eid=2-s2.0-85147732331&doi=10.1145%2F3581576.3581617&partnerID=40&md5=24ca2c9e33b9f03e778c8c8a8233b5db.

[49] A. Schmitz et al. "Parallel Pattern Compiler for Automatic Global Optimizations". In: *Parallel Computing* 122 (2024). DOI: 10.1016/j.parco.2024.103112. URL: https://www.scopus.com/inward/record.uri?eid=2-s2.0-85204871481&doi=10.1016%2Fj.parco.2024.103112&partnerID=40.

[50] Changqing Shi et al. "TransCL: An Automatic CUDA-to-OpenCL Programs Transformation Framework". In: *ACM Trans. Archit. Code Optim.* 22.2 (June 2025). Place: New York, NY, USA Publisher: Association for Computing Machinery. ISSN: 1544-3566. DOI: 10.1145/3718987. URL: https://doi.org/10.1145/3718987.

[51] Jun Shirako, Louis-Noël Pouchet, and Vivek Sarkar. "Oil and Water Can Mix: An Integration of Polyhedral and AST-Based Transformations". In: *SC '14: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*. 2014, pp. 287–298. DOI: 10.1109/SC.2014.29.

[52] Jun Shirako et al. "Expressing DOACROSS Loop Dependences in OpenMP". In: *OpenMP in the Era of Low Power Devices and Accelerators*. Ed. by Alistair P. Rendell, Barbara M. Chapman, and Matthias S. Müller. Berlin, Heidelberg: Springer Berlin Heidelberg, 2013, pp. 30–44. ISBN: 978-3-642-40698-0.

[53] S. Siso, A.R. Porter, and R.W. Ford. "Transforming Fortran weather and climate applications to OpenCL using PSyclone". In: 2023. DOI: 10.1145/3585341.3585360. URL: https://www.scopus.com/inward/record.uri?eid=2-s2.0-85153401773&doi=10.1145%2F3585341.3585360&partnerID=40&md5=7837cf92a8fe4928eeea1afa79fa53cf.

[54] P. Skotnicki. "TC: Optimizing compiler". In: *SoftwareX* 32 (2025). DOI: 10.1016/j.softx.2025.102351. URL: https://www.scopus.com/inward/record.uri?eid=2-s2.0-105016462890&doi=10.1016%2Fj.softx.2025.102351&partnerID=40&md5=5b99e3500fe0298be1de233a1406132c.

[55] S. Tang and R.O. Sinnott. "Towards the Translation of OpenMP to Hybrid MPI/OpenMP". In: 2025, pp. 66–73. DOI: 10.1145/3727166.3727186. URL: https://www.scopus.com/inward/record.uri?eid=2-s2.0-105012252447&doi=10.1145%2F3727166.3727186&partnerID=40.

[56] A. Thangamani, V. Loechner, and S. Genaud. “Extending Polygeist to Generate OpenMP SIMD and GPU MLIR Code”. In: Lecture Notes in Computer Science. Vol. 15386 LNCS. 2025, pp. 323–328. DOI: 10.1007/978-3-031-90203-1_36. URL: https://www.scopus.com/inward/record.uri?eid=2-s2.0-105008654526&doi=10.1007%2F978-3-031-90203-1_36&partnerID=40.

[57] W. Vanderbauwhede. “Making legacy Fortran code type safe through automated program transformation”. In: *Journal of Supercomputing* 78.2 (2022), pp. 2988–3028. DOI: 10.1007/s11227-021-03839-9. URL: https://www.scopus.com/inward/record.uri?eid=2-s2.0-85123495660&doi=10.1007%2Fs11227-021-03839-9&partnerID=40&md5=f8ad14f574f1603d50adbf51e35fda46.

[58] Sven Verdoolaege. “isl: An Integer Set Library for the Polyhedral Model”. In: *Mathematical Software – ICMS 2010*. Ed. by Komei Fukuda et al. Berlin, Heidelberg: Springer Berlin Heidelberg, 2010, pp. 299–302. ISBN: 978-3-642-15582-6.

[59] Sven Verdoolaege. *Presburger Formulas and Polyhedral Compilation*. Version v0.02-13-g53cb23d. Polly Labs and KU Leuven. Aug. 2021.

[60] Sven Verdoolaege and Tobias Grosser. “Polyhedral Extraction Tool”. In: *Second Int. Workshop on Polyhedral Compilation Techniques (IMPACT’12)*. Paris, France, Jan. 2012.