ALMA MATER STUDIORUM – UNIVERSITÀ DI BOLOGNA
CAMPUS DI CESENA

Dipartimento di Informatica – Scienza e Ingegneria
Corso di Laurea in Ingegneria e Scienze Informatiche

Titolo dell'elaborato

OPoly: an OpenMP polyhedral compiler

Elaborato in
High-performance Computing

Relatore
Prof. Moreno Marzolla

Presentata da
Giacomo Aloisi

IV Sessione di Laurea
Anno Accademico 2019-2020

Abstract

In this thesis, we introduce polyhedral compilation, a multitude of techniques for representing programs, especially those involving nested loops and arrays, thanks to parametric polyhedrons, and exploit transformations on these objects to automatically analyze and optimize the given programs. In particular, we describe our implementation of a polyhedral compiler: OPoly.

OPoly is a Python application able to perform automatic parallelization of loop nests that can be expressed as a set of uniform recurrent equations. OPoly analyzes loop nests written in pseudocode language and generates parallelizable C code with OpenMP directives, which can replace its original, serial implementation without changing the meaning of the program.

OPoly uses the MiniZinc constraint programming modeling language to model the optimization problems of the polyhedral approach, which are crucial for finding the best possible transformation from the original loop to the parallelizable one.

We describe the architecture of OPoly and give some practical solutions to problems that arise when implementing a polyhedral compiler.

Finally, we compare the performance of the generated parallel code to its original implementation, by studying the application of OPoly on a well-known scientific algorithm implementation.

Sommario

In questa tesi introduciamo la *polyhedral compilation*, una moltitudine di tecniche volte a rappresentare programmi, specialmente quelli che coinvolgono cicli innestati e vettori, rappresentandoli attraverso poliedri parametrici e sfruttando alcune trasformazioni su di essi per analizzare e ottimizzare automaticamente i programmi dati. In particolare, descriviamo la nostra implementazione di un *polyhedral compiler*: OPoly.

OPoly è un'applicazione realizzata in Python in grado di parallelizzare automaticamente cicli innestati che possono essere espressi attraverso un insieme di equazioni ricorsive uniformi. OPoly analizza dei cicli innestati scritti in pseudolinguaggio e genera del codice sorgente parallelizzabile scritto in C e compreso di direttive OpenMP, che può rimpiazzare l'implementazione seriale originale, senza cambiarne il significato.

OPoly sfrutta il linguaggio di modellazione per programmazione a vincoli MiniZinc per modellare il problemi di ottimizzazione dell'approccio poliedrico, che sono cruciali per trovare la trasformazione migliore possibile dal ciclo originale a quello parallelizzabile.

Descriviamo l'architettura di OPoly e diamo alcune soluzioni pratiche a problemi che sorgono implementando un *polyhedral compiler*.

Infine, compariamo le prestazioni del codice parallelizzabile generato da OPoly con quelle della relativa implementazione originale, studiando il caso di un noto algoritmo scientifico.

To my parents, thank you for teaching me about many aspects of life.
To my friends, without whom I would not have withstood
through these dark times.
To my partners in crime, thanks for putting up with my madness
and sharing with me a bit of yours too.
To my professors, my gratitude for letting me thrive intellectually
while instilling inspiration for my future self.
To my university, my thankfulness for giving me the chance to become
a better person and to get to know a lot of crazy people like myself.
To my former colleagues, I hope to give back just a tiny bit of what you gave me while
trying to teach this dumb guy about the real stuff.
To all who supported me, my deepest thanks.

Science is what we understand
well enough to explain to a
computer.
Art is everything else we do.

Foreword to the book "A=B"
DONALD E. KNUTH

Acknowledgements

First of all, I would like to thank my thesis supervisor, professor Moreno Marzolla, who inspired me in pursuing high-performance computing studies, after attending his academic course on the subject. Definitely one of the most interesting courses I attended as an undergraduate student. His great tips and attention to detail also helped me write a better thesis overall.

I found out about polyhedral compilation when I was in Paris for my curricular internship, so big thanks to Huawei Technologies France, my former tutor Zhen Zhang and his colleagues for introducing me to the subject. I have learned a lot from that experience and my interest in high-performance computing grew stronger after that. If it was not for you, this thesis would not have been accomplished.

Last but not least, many thanks to all the members of the polyhedral research community. Over the years, you have done an amazing job in the dissemination of polyhedral compilation. With your numerous contributions, I was able to write this thesis and (hopefully) help to spread the interest in this subject to the academic world. I will definitely continue to learn more about polyhedral compilation, with the hope of becoming a contributor in the future.

Contents

List of Figures

List of Listings

List of Algorithms

Chapter 1

Introduction

1.1 Motivation

"Any program using a significant amount of computer time spends most of that time executing one or more loops": this is the beginning of the introduction of *The Parallel Execution of DO Loops* [Lam74] by Leslie Lamport, which we use throughout the thesis as our main reference for many of the ideas regarding polyhedral compilation. This statement is particularly true for scientific computations, in which many nested loops perform CPU-intensive work, usually by operating on multidimensional arrays to store intermediate results.

With the advent of parallel computing (and parallel architectures in general), much of the existing code needs to be rewritten (partially or completely) to fulfill a certain degree of parallelization necessary to speed up computation, otherwise limited by the single CPU's clock frequency.

However, parallel programming has earned a reputation as one of the most difficult subjects in computer programming, that very few programmers can understand, let alone code it all right. As explained in [McK17], parallel programming difficulties can fall into several categories, such as the historic high cost and relative rarity of parallel systems, or the typical researcher's and practitioner's lack of experience with parallel systems. Fortunately, things are starting to change due to the increase in accessibility of multi-processors systems to end users, ready-to-use APIs for parallel computing, and advancements in automatic parallelization of existing code.

This thesis ventures into the vast world of polyhedral compilation, trying to tackle just a small part of it from an end-user perspective. Broadly speaking, Polyhedral Compilation (PC) is a variety of techniques used to represent programs as some sort of mathematical model, namely polyhedrons (or polyhedra), and to apply transformations on these objects to optimize them in some way;

finally, the resulting model is transformed back into code that is now optimized, without changing the meaning of the original program.

For the intents of this thesis, we want to use PC to generate parallelizable programs given their serial implementations. This is done by using a technique known as the *hyperplane method* [Lam74]. This approach finds a way to rewrite a loop nest such that the statements in the loop body can be concurrently executed for all points lying along a hyperplane of the index space, hence the name hyperplane method. Many hyperplanes are (serially) scanned during computation until no more points in the index space are to be computed. However, each point in the given hyperplane can be computed in parallel, so that a part of the resulting program can now be parallelized.

Generally, PC involves the manipulation of *polyhedra* or *polytopes*, geometric objects having "flat" sides that represent the index space of the loop nest. Each loop iteration of the loop nest can be treated as a point in the lattice of a polyhedron, hence transformations on the polyhedron also transform the order in which the iterations are being executed. By doing proper dependence-preserving transformations, one can convert the initial polyhedron into an equivalent, but optimized accordingly to some optimization goals, *target polyhedron*, which can then be used to generate the transformed loop nest through a process called *polyhedra scanning*. In our case, the optimization goal is to minimize the number of hyperplanes that need to be scanned to compute all the points inside the polyhedron, hence minimizing the serial part of the target program.

With this said, in a reader's mind a question may arise (sorry for the rhyme, and then for the word pun): why isn't polyhedral compilation used everywhere? Simply putting it, there are certain assumptions that the code must fulfill to be optimized with polyhedral compilation. Generally, every technique has its own set of restrictions on what can or cannot be inside the loop body and other properties of the program. The hyperplane method has its assumptions too, perhaps the most restrictive ones since it is one of the first approaches adopted, as well as one of the foundation methods of PC in general.

In this work, we describe OPoly, a polyhedral compiler that implements the hyperplane method to generate parallelizable C code containing OpenMP clauses from a program written in pseudocode language. OPoly can easily be extended to support different languages and other parallel programming paradigms, and, with a bit of effort, different PC techniques.

We also talk about the gains in performance of the generated code with regard to the serial version. To do this, we apply the OPoly transformation to some well-known scientific algorithm implementations, then analyze the speedup and scaling efficiencies of these parallel versions.

1.2 Thesis structure

This thesis is structured into chapters, each one regarding a different aspect of the knowledge behind OPoly. One can choose to skim over chapters to find what they need, or simply read the whole thesis from top to bottom. However, getting the grasp of the main concepts of the hyperplane method may be necessary in order to understand some notions used in other chapters, so it is recommended to read carefully at least section 2.4 (and possibly section 2.1 and section 2.2) of chapter 2 before adventuring into other chapters. Accordingly, the remainder of this thesis is structured as follows.

Chapter 2 introduces notations and definitions used throughout the thesis, describing the polytope model and, more specifically, the hyperplane method. As stated, this is probably the most important (and in our opinion interesting) chapter that should be at least looked at. We also describe the assumptions the code must satisfy in other to be parallelizable with the hyperplane method and the methods used for scanning polyhedra to generate the target program.

In chapter 3 we argue the choices that lead to the design of OPoly, describing the adopted representation of programs and the architecture of the application.

Chapter 4 gives a detailed explanation on how to effectively solve the problems presented in chapter 2, by describing the techniques and algorithms used by OPoly and introducing MiniZinc [Net+07], a modeling language for constraint programming problems. We also describe the pseudocode language used to express programs that can be analyzed by OPoly and the generation of the resulting parallel code.

In chapter 5 we show how to use OPoly in practice by generating parallel code for some implementations of the *Gauss-Seidel* algorithm. We also analyze the performance of the generated code by showing execution times, speedups, and scaling efficiencies.

Finally, chapter 6 concludes this thesis by summarising OPoly's main achievements and outlining a roadmap for future extensions and optimizations of OPoly.

Chapter 2

Background

In this chapter, we introduce the main ideas of polyhedral compilation (PC), focusing on the contribution of [Lam74]. We use concepts and notations from [Fea96] and [Len93] to formalize these ideas.

The basis for the automatic synthesis of nested loops with the polytope model was laid in the 1960s with the paper [KMW67] regarding uniform recurrence equations. In the 1970s, Lamport was the first to apply this approach and develop a method for the automatic parallelization of for loops. After that, the idea picked up and developed further with the birth of systolic arrays, unified in the late 1980s with a paper [RK88] that resulted in a theory for the automatic synthesis of all systolic arrays.

Nowadays, the theory has evolved to include several other optimizations and to progressively remove restrictions on the properties of the programs that can be optimized with these techniques. Many polyhedral libraries [Loe99; WIL00] and compilers [BRS07; GGL12; Bag+19] have been created for dealing with polyhedra and for automatic parallel code generation.

The goal of this thesis is not to present cutting-edge methods for PC since the theory underneath it is so complex that a lot of prior knowledge on the subject would be needed to understand them. Our scope is to give a gentle introduction to PC by getting a gist of a simple method like the hyperplane method, that is comprehensible by anyone who has a bit of confidence with simple programming, linear algebra, and linear programming problems.

We also describe the process of completing the space transformations and rewriting the loop nests to obtain code that is parallelizable straight away, while being computationally equivalent to the original.

We give examples as we explain the concepts, to achieve better clarity and to let the reader follow through with the explanations.

2.1 Notations

In this section, we introduce some of the basic notations used throughout the thesis.

We assume that every numeric value from now on assumes only integer values from the set $\mathbb{Z}$, unless explicitly stated otherwise.

We denote a *scalar* value with a lowercase letter (e.g. $k \in \mathbb{Z}$).

We denote a n-dimensional *vector* with a bold lowercase letter (e.g. $\mathbf{v} \in \mathbb{Z}^n$).

We denote an m-by-n *matrix* with an uppercase letter (e.g. $M \in \mathbb{Z}^{m \times n}$).

We denote with $f : \mathcal{D} \to \mathcal{C}$ a function f with domain $\mathcal{D}$ and codomain $\mathcal{C}$.

We say that two vectors $\mathbf{a}, \mathbf{b} \in \mathbb{Z}^n$ are in *lexicographic order* and we denote it with $\mathbf{a} \ll \mathbf{b}$ if there exists an $1 \leq i \leq n$ such that $a_i < b_i$ and $a_j \leq b_j$ for all $1 \leq j < i$ (e.g. with $n = 3$, $(5, 3, -2) \ll (5, 4, -10)$).

Loop notation
We consider n-nested loops of the form of listing 2.1 (similar to the one in [Lam74]), where the $i_1, \dots, i_n$ are the *loop index variables*, the $\ell_1, \dots, \ell_n$ and the $u_1, \dots, u_n$ are integer-valued expressions called respectively the *lower bounds* and *upper bounds* of the loops. The bounds ℓ_j and u_j of the j-th loop can involve program's parameters, constant values, or *linear combinations* of indexes from previous loops: that is expressions of the form $a_1 i_1 + \dots + a_{j-1} i_{j-1}$ where $\mathbf{a} \in \mathbb{Z}^{j-1}$ (e.g. $u_3 = i_1 + 2i_2 + M - 2$ where $M \in \mathbb{Z}$ is a program's parameter). We allow only unitary increments of the loop indexes.

```
FOR $i_1$ FROM $\ell_1$ TO $u_1$
         ...
FOR $i_n$ FROM $\ell_n$ TO $u_n$
    loop body
```

Listing 2.1: Original loop nest format.

Let VAR be an array variable that appears in the loop body. A variable that appears once or more on the left-hand side of an assignment statement in the loop body is called a *generated* variable.

An occurrence of VAR is any appearance of it in the loop body. If it appears on the left-hand side of an assignment statement, the occurrence is called a *generation*; otherwise, it is called a *use*. Thus, generations modify the values of elements of the array variables, while uses do not.

We make the following assumptions about the loop body:

(A1) It contains no I/O statement.

(A2) It contains no transfer of control to any statement outside the loop.

(A3) It contains no subroutine or function call which can modify data.

(A4) Any occurrence in the loop body of a generated variable VAR is in the form VAR($e_1, \ldots, e_r$) where each e_i is an expression not containing any generated variables.

More assumptions on the statements regarding the loop body are be considered in section 2.4.

OPoly rewrites the loop nest from the form of listing 2.1 to the form of listing 2.2, where the $i'_1, \ldots, i'_n$ are the new loop index variables, the $\lambda_1, \ldots, \lambda_n$ and the $\mu_1, \ldots, \mu_n$ are the new loop bounds. The **FOR CONC** notation express the fact that the loop can be executed concurrently by many processors.

```
FOR $i'_1$ FROM $\lambda_1$ TO $\mu_1$
FOR CONC $i'_2$ FROM $\lambda_2$ TO $\mu_2$
          ...
FOR CONC $i'_n$ FROM $\lambda_n$ TO $\mu_n$
    loop body
```

Listing 2.2: Rewritten loop nest format.

2.2 The geometry of programs

The base intuition behind PC is that for loops iterations can be seen as points in a traditional Euclidean space. Each loop that iterates through a variable, called *index*, is effectively representing a dimension on this space, hence a loop nest of n nested loops define a n-dimensional index space. The index space $\mathcal{IS}$, is the set of all values that are assumed by the indexes during the execution of the loop nest.

To better illustrate this concept, consider the loop nest in alg. 2.1. In the loop body, Q is an array variable (specifically a matrix) that is initialized before the start of the computation, and its elements are updated according to the inner loop statement. The loop nest has two index loop variables, namely i and j, which assume the range of integers between 1 and N inclusively. The parameter N is an integer know at execution time which value does not change during the execution. In this specific case, the value $N + 1$ is assumed to be the size of the matrix Q (with zero-based indexing).

Algorithm 2.1: 2D COMPUTATION EXAMPLE

```
for $i$ from 1 to $N$ do
    for $j$ from 1 to $N$ do
        $Q(i, j) \leftarrow (Q(i-1, j) + Q(i, j-1)) * 0.5$
    end
end
```

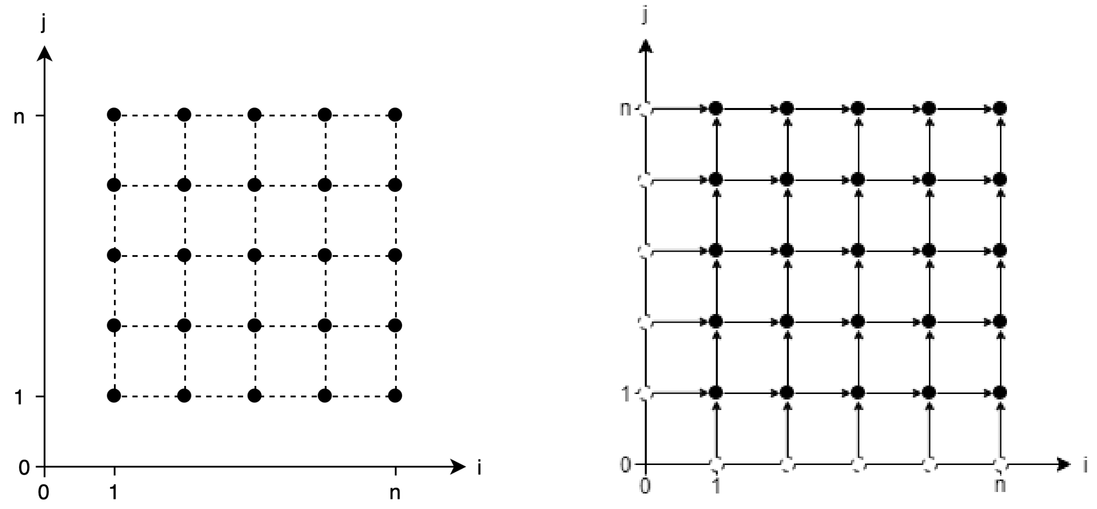
The index space of this loop is:

$$\mathcal{IS} = \{(i, j) \mid 1 \leq i \leq N, 1 \leq j \leq N\}.$$

The index space of this loop nest can be represented in a 2-dimensional space, where each axis represents the values that the corresponding index assumes during execution. A visual representation of the index space of alg. 2.1 is shown in fig. 2.1a.

This loop nest is not parallelizable right away, because there are *loop-carried dependencies* between some elements of Q: for example, $Q(2, 2)$ cannot be computed before computing $Q(1, 2)$ and $Q(2, 1)$. In general, in order to compute $Q(i, j)$ for some $i, j = 1 \ldots N$, we need to wait for the computation of $Q(i-1, j)$ and $Q(i, j-1)$. A representation of the dependencies between different iterations, also known as the *dependence graph*, is shown in fig. 2.1b.

Trying to parallelize the loop nest as-is (for example by diving the workload on multiple processors) will lead to wrong and unpredictable results. We want



(a) Index space. (b) Dependence graph.

Figure 2.1: The index space and dependence graph of alg. 2.1.

to rewrite the loop nest in a way that some points can be safely computed in parallel. Specifically, we find that the loop nest can be rewritten as alg. 2.2, by finding the indexes transformation:

$$
\begin{aligned}
i' &= i + j \\
j' &= j
\end{aligned}
$$

.

Algorithm 2.2: 2D COMPUTATION EXAMPLE (PARALLEL)

```
1 for $i'$ from 2 to $2N$ do
2     for conc $j'$ from $\max(1, i' - N)$ to $\min(N, i' - 1)$ do
3         $Q(i' - j', j') \leftarrow (Q(i' - j' - 1, j') + Q(i' - j', j' - 1)) * 0.5$
4     end
5 end
```

Original index values can be obtained by inverting the transformation:

$$
\begin{aligned}
i &= i' - j' \\
j &= j'
\end{aligned}
$$

The index space $\mathcal{TS}$ of the transformed loop nest, also called the *target space*, is described by:

$$\mathcal{TS} = \{(i', j') \mid 1 \leq i' - j' \leq N, 1 \leq j' \leq N\},$$

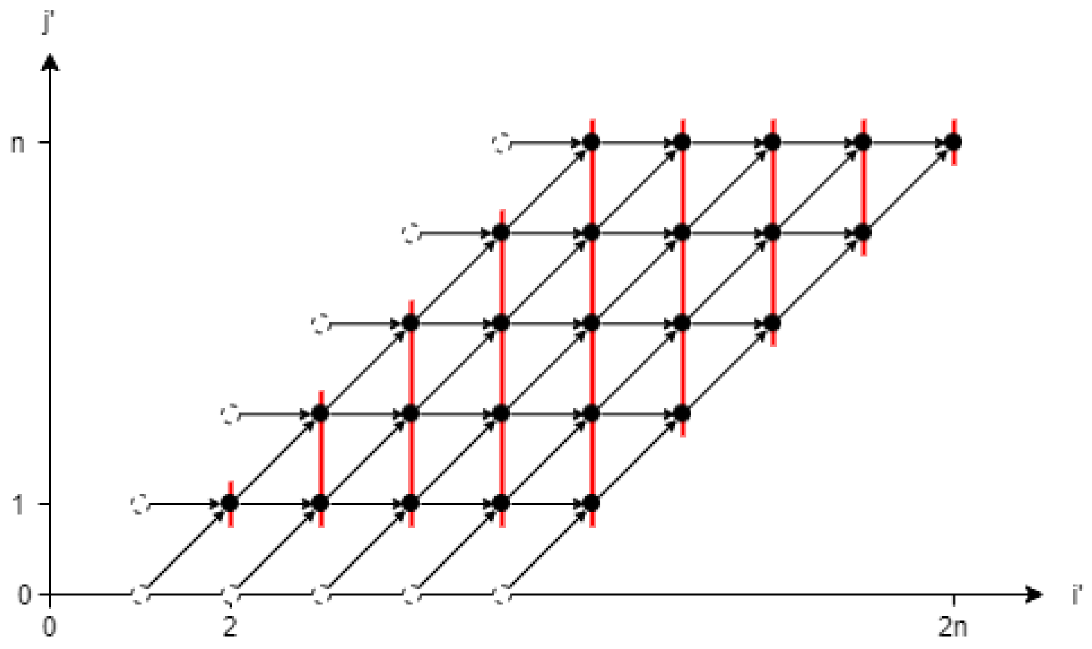


Figure 2.2: The target index space and dependence graph of alg. 2.2. Red lines indicate points that can be executed in parallel.

New lower and upper bounds for the transformed index variables can be found from the target space with a process called *polyhedra scanning*. A visual representation of the target space is shown in fig. 2.2, as well as the original dependencies between points. Points that can be computed in parallel lie on the same red line. Note that the points lying on the same line do not depend on each other, but only on the points lying on the line before.

Each "line" of points can be computed in parallel, but only after computing all the points in the previous lines. This assures that the computation gives correct and predictable results.

The inner loop of alg. 2.2 can now be executed concurrently by many processors, while the outer loop is still executed sequentially. Assuming that we assign a different processor for each iteration of the inner loop, the total number of sequential iterations is reduced from N^2 to $2N - 1$, giving the possibility of a big reduction in execution time.

In the next sections, we formalize these concepts by introducing the polytope model and the hyperplane method.

2.3 The polytope model

This geometric representation of programs can be expressed by mathematical objects called *polyhedra* (plural of polyhedron). In classical Euclidean geometry, the word polyhedron indicates a three-dimensional shape with polygonal faces, and the generalization of a polyhedron for any number of dimensions is usually called *polytope*, or n-polytope if its dimensions are n. With this nomenclature, a polygon is a 2-polytope and a polyhedron is a 3-polytope.

In certain fields of mathematics, the term polyhedron refers to a generic object in any dimensions, whilst the term polytope is used to denote a *bounded* polyhedron, i.e. a polyhedron which does not extend infinitely in some direction.

In this thesis, we use the term polyhedron and polytope indifferently to represent a n-dimensional, bounded, convex geometric object with "flat" sides.

Definition 2.3.1 (Polyhedron)**.** In linear programming, a *polyhedron* is described by a set of m linear inequalities in n variables:

$$A\mathbf{x} \leq \mathbf{b}, \tag{2.3.1}$$

where $A \in \mathbb{R}^{m \times n}$ and $\mathbf{x}, \mathbf{b} \in \mathbb{R}^n$. A polyhedron $\mathcal{P}$ is the set of all $\mathbf{x} \in \mathbb{R}^n$ which satisfy these inequalities:

$$\mathcal{P} = \{\mathbf{x} \mid A\mathbf{x} \leq \mathbf{b}\}. \tag{2.3.2}$$

A polyhedron can be empty (the set of defining inequalities is said to be *infeasible*) or unbounded (it extends infinitely in some direction). The basic property of a polyhedron is *convexity*: if two points $\mathbf{x}$ and $\mathbf{y}$ belong to a polyhedron, then so do all convex combinations $\lambda\mathbf{x} + (1 - \lambda)\mathbf{y}, 0 \leq \lambda \leq 1$.

One can also describe a polyhedron by the set of all convex combinations of a finite set of points, some called *vertices* and other *rays* (the latter if they are at some direction at infinity).

Polyhedra are usually defined in a $\mathbb{R}^n$ space, so, in order to consider integer points only, we need a way to restrict the possible points inside a polyhedron to a so-called *lattice* of integral points.

2.3.1 $\mathbb{Z}$-polyhedron

Definition 2.3.2 ($\mathbb{Z}$-module)**.** Let $\mathbf{v}_1, \ldots, \mathbf{v}_n$ be a set of linearly independent vectors of $\mathbb{Z}^n$. The set:

$$\mathcal{L}(\mathbf{v}_1, \ldots, \mathbf{v}_n) = \{\mu_1\mathbf{v}_1 + \ldots + \mu_n\mathbf{v}_n \mid \mu_i \in \mathbb{Z}\}, \tag{2.3.3}$$

is the $\mathbb{Z}$-module generated by $\mathbf{v}_1, \ldots, \mathbf{v}_n$.

Any $\mathbb{Z}$-module can be characterized by the square matrix V of which the $(\mathbf{v}_1, \ldots, \mathbf{v}_n)$ are the column vectors. $\mathcal{L}(V)$ is also called the *lattice* of points generated by V.

If $V = I$ the identity matrix, then the set of all integral points in $\mathbb{Z}^n$ is the $\mathbb{Z}$-module generated by the canonical basis vectors in V.

However, many different matrices can represent the same $\mathbb{Z}$-module. In this regard, it is useful to define a special type of square, integer matrix, called *unimodular* matrix.

Definition 2.3.3 (Unimodular matrix)**.** A square matrix U is said to be *unimodular* if it is an *integer* matrix ($U \in \mathbb{Z}^{n\times n}$) and $\det(U) = \pm 1$.

It is easy to prove that V and VU generate the same lattice of points (for more information about unimodular matrices see [Ban93]).

We can now combine a polyhedron with a $\mathbb{Z}$-module to represent a polyhedron with only integer points.

Definition 2.3.4 ($\mathbb{Z}$-polyhedron)**.** A $\mathbb{Z}$-polyhedron is the intersection of a $\mathbb{Z}$-module and a polyhedron:

$$F = \{\mathbf{z} \mid \mathbf{z} \in \mathcal{L}(V), A\mathbf{z} \leq \mathbf{b}\}. \tag{2.3.4}$$

If the context is clear, and if $\mathcal{L}(V)$ is the canonical $\mathbb{Z}$-module ($V = I$), it may be omitted in the definition.

Looking back at the loop in alg. 2.1, we can describe its index space $\mathcal{IS}$ with the following (parametric) $\mathbb{Z}$-polyhedron:

$$\begin{bmatrix} -1 & 0 \\ 0 & -1 \\ 1 & 0 \\ 0 & 1 \end{bmatrix} \begin{bmatrix} i \\ j \end{bmatrix} \leq \begin{bmatrix} -1 \\ -1 \\ N \\ N \end{bmatrix}.$$

It is called a parametric $\mathbb{Z}$-polyhedron because the vector $\mathbf{b}$ is made up of constants and parameters (e.g. -1 and N).

Definition 2.3.5 (Index space)**.** In general, we can describe the index space $\mathcal{IS}$ of a n-nested loop with the following (parametric) $\mathbb{Z}$-polyhedron:

$$\mathcal{IS} = \{\mathbf{i} \mid C\mathbf{i} \leq \mathbf{d}\}, \tag{2.3.5}$$

where $C \in \mathbb{Z}^{2n\times n}$, $\mathbf{d} \in \mathbb{Z}^{2n}$ and $\mathbf{i} \in \mathbb{Z}^n$.

The index vector $\mathbf{i} = (i_1, \ldots, i_n)$ represents a specific loop iteration for the values assumed by the index variables $i_1, \ldots, i_n$. Every lower and upper bound of the loop nest can be transformed into an inequality regarding the index variables. The matrix C contains the coefficients of the index variables in these inequalities, while the vector $\mathbf{d}$ contains integer-valued expressions with program's parameters and constants.

2.3.2 Schedule + allocation = transformation matrix

The goal of this approach is to find a correctness-preserving transformation of the initial index space that yields a new index space in which the points are now reordered in a way that some of them can be computed in parallel. To achieve this, we must consider the dependencies between points of the computation and find a valid schedule that maps a point to a certain time in which it can be computed.

Definition 2.3.6 (Schedule)**.** Let $\mathcal{IS}$ be a $\mathbb{Z}$-polyhedron and consider the dependence graph $(\mathcal{IS}, E)$, where E is the edge set defining the dependencies between points in $\mathcal{IS}$. The function $t : \mathcal{IS} \to \mathbb{Z}$ is called a *schedule* if it preserves the dependencies:

$$\forall \mathbf{p}, \mathbf{q} : \mathbf{p}, \mathbf{q} \in \mathcal{IS} \wedge (\mathbf{p}, \mathbf{q}) \in E : t(\mathbf{p}) < t(\mathbf{q}) \tag{2.3.6}$$

That means that the points that are dependent on each other are scheduled at successive times. All points that share the same schedule can be therefore computed in parallel.

Geometrically, the schedule slices the index space into parallel *hyperplanes*, subspaces whose dimensionality is one less than that of the index space (in the case of fig. 2.2, hyperplanes are lines). The requirement for the schedule prescribes that the hyperplanes are not parallel to any edge of the dependence graph.

Definition 2.3.7 (Allocation)**.** Let $\mathcal{IS}$ be a n-dimensional polytope and t a schedule, the function $a : \mathcal{IS} \to \mathbb{Z}^{n-1}$ is called an *allocation* with regard to the schedule t if each process it defines is internally sequential:

$$\forall \mathbf{p}, \mathbf{q} : \mathbf{p}, \mathbf{q} \in \mathcal{IS} : t(\mathbf{p}) = t(\mathbf{q}) \Rightarrow a(\mathbf{p}) \neq a(\mathbf{q}) \tag{2.3.7}$$

Geometrically, the allocation segments the index space into parallel lines. Each line contains the points executed by a fixed processor. The consistency requirement for schedule and allocation prescribes that the lines generated by the allocation are not parallel to the hyperplanes generated by the schedule.

We consider the case in which both t and a are *linear functions*, that is they can be written in the forms:

$$t(\mathbf{x}) = \tau_1 x_1 + \ldots \tau_r x_n, \tag{2.3.8}$$

$$a(\mathbf{x}) = \left(\sum_{j=1}^{n-1} \alpha_{j,1} x_1, \ldots, \sum_{j=1}^{n-1} \alpha_{j,n} x_n \right), \tag{2.3.9}$$

for some $\tau \in \mathbb{Z}^n, A \in \mathbb{Z}^{(n-1)\times n}$:

$$\tau = (\tau_1, \ldots, \tau_n),$$
$$A = \begin{bmatrix} \alpha_{1,1} & \ldots & \alpha_{1,n} \\ \vdots & \ddots & \vdots \\ \alpha_{n-1,1} & \ldots & \alpha_{n-1,n} \end{bmatrix}.$$

Combining τ and A gives use the matrix $T \in \mathbb{Z}^{n\times n}$, also called the *space-time* matrix, that defines the transformation:

$$T = \begin{bmatrix} \tau \\ A \end{bmatrix}. \tag{2.3.10}$$

The requirement on the allocation is that $\det(T) \neq 0$, so that 2.3.7 is fulfilled.

The challenge is to find not only a valid transformation but also a *good quality* one. We dive deeper into the definition and finding of a good quality transformation in section 2.4.

2.3.3 Target polyhedron

We now need to describe how to apply the transformation matrix T to find the *target space* $\mathcal{TS} = T(\mathcal{IS})$, that is the target polyhedron after applying the transformation to the initial index space $\mathcal{IS}$. The points of the target space are defined by:

$$T(\mathcal{IS}) = \{\mathbf{i}' \mid \exists \mathbf{i} : \mathbf{i}' = T\mathbf{i}, C\mathbf{i} \leq \mathbf{d}\}, \tag{2.3.11}$$

where $C\mathbf{i} \leq \mathbf{d}$ is the system of inequalities describing the index space $\mathcal{IS}$ of the original loop nest. This shows that $\mathbf{i}'$ belongs to the lattice $\mathcal{L}(T)$. Since T is invertible, the set $T(\mathcal{IS})$ can be rewritten as:

$$T(\mathcal{IS}) = \{\mathbf{i}' \mid \mathbf{i}' \in \mathcal{L}(T), CT^{-1}\mathbf{i}' \leq \mathbf{d}\}, \tag{2.3.12}$$

which defines a $\mathbb{Z}$-polyhedron.

Returning to the example described in section 2.2, the space-time matrix that describes the transformation from alg. 2.1 to alg. 2.2 is:

$$T = \begin{bmatrix} 1 & 1 \\ 0 & 1 \end{bmatrix}, T^{-1} = \begin{bmatrix} 1 & -1 \\ 0 & 1 \end{bmatrix},$$

defining the mapping:

$$T\mathbf{i} = \begin{bmatrix} 1 & 1 \\ 0 & 1 \end{bmatrix} \begin{bmatrix} i \\ j \end{bmatrix} = \begin{bmatrix} i+j \\ j \end{bmatrix} = \mathbf{i}',$$

the inverse mapping:

$$T^{-1}\mathbf{i}' = \begin{bmatrix} 1 & -1 \\ 0 & 1 \end{bmatrix} \begin{bmatrix} i' \\ j' \end{bmatrix} = \begin{bmatrix} i' - j' \\ j' \end{bmatrix} = \mathbf{i},$$

and the target space $\mathcal{TS}$ bounded by the system of inequalities:

$$CT^{-1}\mathbf{i}' \leq \mathbf{d}$$

$$\begin{bmatrix} -1 & 0 \\ 0 & -1 \\ 1 & 0 \\ 0 & 1 \end{bmatrix} \begin{bmatrix} 1 & -1 \\ 0 & 1 \end{bmatrix} \begin{bmatrix} i' \\ j' \end{bmatrix} \leq \begin{bmatrix} -1 \\ -1 \\ N \\ N \end{bmatrix}$$

$$\begin{bmatrix} -1 & 1 \\ 0 & -1 \\ 1 & -1 \\ 0 & 1 \end{bmatrix} \begin{bmatrix} i' \\ j' \end{bmatrix} \leq \begin{bmatrix} -1 \\ -1 \\ N \\ N \end{bmatrix}$$

$$\begin{cases} i' - j' \geq 1 \\ j' \geq 1 \\ i' - j' \leq N \\ j' \leq N \end{cases}$$

2.4 The hyperplane method

In this section, we introduce the hyperplane method (described in [Lam74]) that we use to find an optimal schedule for the transformation of loops that can be represented as uniform recurrence equations [KMW67]. To consider only uniform dependencies, we need to add another assumption on the form of the variables in the loop body:

(A5) Each occurrence of a generated variable VAR in the loop body is of the form:

$$\mathsf{VAR}(i_{j_1} + m_1, \ldots, i_{j_r} + m_r), \tag{2.4.1}$$

where $m_1, \ldots, m_k$ are integer constants, and $j_1, \ldots, j_r$ are r **distinct** integers between 1 and n. Moreover, the $j_1, \ldots, j_r$ are the same for any two occurrences of VAR.

This enforces that data dependencies between points in the index space are point-to-point and local.

It is possible to generalize to the case of *affine* recurrence equations, where the occurrences are in the form $\mathsf{VAR}(e_1, \ldots, e_r)$ where the $e_1, \ldots, e_r$ are affine functions of the index variables $i_1, \ldots, i_n$ (i.e. $e_j = \mathbf{a}_j\mathbf{i} + b_j, \mathbf{a}_j \in \mathbb{Z}^n, b_j \in \mathbb{Z}$ for some $1 \le j \le r$), but we consider only uniform functions for simplicity.

The name "hyperplane method" comes from the fact that we want to find a parametric hyperplane such that points lying on it can be executed concurrently. The hyperplane we refer to is defined by the schedule function of definition 2.3.6. At the time the hyperplane method was proposed, the polytope model was not yet formalized, so some nomenclature got changed during the evolution of PC techniques.

Looking back at the example in alg. 2.1, the hyperplane method finds the schedule $t(i, j) = i + j$ which defines an hyperplane of the form $i + j = constant$. The constant is incremented after each iteration of the transformed outer loop (see alg. 2.2) until the loop body has been executed for all points in the index space.

2.4.1 The occurrence mapping

In section 2.3 we used the dependence graph to define a schedule in definition 2.3.6. We can express the dependencies between each variable occurrences in a more compact way by introducing the *occurrence mapping* function and the $\langle f, g\rangle$ sets.

Firstly, we need a function that, given a variable occurrence and a point in $\mathcal{IS}$, gives us the element of the variable referred by that occurrence at that specific iteration.

From now on, we call $q1$ the generation $Q(i, j)$ of the variable Q in alg. 2.1, $q2$ the use $Q(i-1, j)$ and $q3$ the use $Q(i, j-1)$. For example, given the occurrence $q2$ and the point $(2, 1)$, the function gives us the point $(1, 1)$.

In general, given a variable VAR containing r indexes of the n indexes of the loop, we define the *occurrence mapping* function $h_f : \mathcal{IS} \to \mathbb{Z}^r$ with regard to the occurrence f of the variable VAR so that f references the $h_f(\mathbf{p})$ element of VAR during execution of the loop body for $\mathbf{p} \in \mathcal{IS}$. Recalling the example above, $h_{u2}(i, j) = (i-1, j)$. Similarly, $h_{u1}(i, j) = (i, j)$ and $h_{u3}(i, j) = (i, j-1)$.

We are looking for a way to update the definition 2.3.6 to use the new occurrence mapping function instead of the dependence graph.

In our loop nest, the data dependencies may be of one of these three types:

- *data-flow* (or *true dependence*), also known as Read After Write (RAW);
- *anti dependence*, also know as Write After Read (WAR);
- *output dependence*, also known as Write After Write (WAW).

We can see that if a generation f and a use g of the same variable VAR reference the same array element during the execution of the loop, there is a RAW or WAR dependence (depending on the order of execution of the two). If both are generations, there is a WAW dependence. In any case, the order of these references must be preserved by the schedule function.

We can use the occurrence mapping to express this condition, saying that if $h_f(\mathbf{p}) = h_g(\mathbf{q})$ and $\mathbf{p} \ll \mathbf{q}$, then the schedule must preserve the order $t(\mathbf{p}) < t(\mathbf{q})$.

These remarks can be combined in the following condition:

(C1) For every variable and every ordered pair of occurrences f, g of that variable, at least one of which is a generation, for each $\mathbf{p}, \mathbf{q} \in \mathcal{IS}, \mathbf{p} \ll \mathbf{q}$ such that $h_f(\mathbf{p}) = h_g(\mathbf{q})$, the schedule t must satisfy the relation $t(\mathbf{p}) < t(\mathbf{q})$.

2.4.2 The $\langle f, g \rangle$ sets

We still have some trouble defining proper constraints for the schedule function t since the condition (C1) requires us to consider many points $\mathbf{p}, \mathbf{q} \in \mathcal{IS}$. We would like to express rule (C1) more compactly, such that constraints on the values $\tau_i, \dots, \tau_n$ that define t are independent of the index space $\mathcal{IS}$. Before continuing, consider the loop in alg. 2.3.

This loop differs from the one in alg. 2.1 because the outer loop index i is missing in the occurrences of the array variable U. In fact, we can say that $r = 2$ for the occurrences of the variable U, as defined in eq. (2.4.1). We assign names to the occurrences of the variable U in alg. 2.3 from left to right as they appear

Algorithm 2.3: Simplified 2D relaxation

```
for $i$ from 1 to $L$ do
    for $j$ from 2 to $M$ do
        for $k$ from 2 to $N$ do
            $U(j,k) \leftarrow (U(j-1,k) + U(j,k-1) + U(j+1,k) + U(j,k+1)) * 0.25$
        end
    end
end
```

Sets	Elements $\gg 0$	Constraints
$\langle q1, q1\rangle = (0,0)$	-	-
$\langle q1, q2\rangle = (1,0)$	$(1,0)$	$\tau_1 > 0$
$\langle q2, q1\rangle = (-1,0)$	-	-
$\langle q1, q3\rangle = (0,1)$	$(0,1)$	$\tau_2 > 0$
$\langle q3, q1\rangle = (0,-1)$	-	-

Table 2.1: Table showing the $\langle f, g\rangle$ sets, elements greater than 0, and schedule constraints of alg. 2.1

in the statement, starting from $U(i, j)$ which we call $u1$, and so on. With the previous definition of occurrence mapping, we can see that $h_{u1}(\mathbf{p}) = h_{u2}(\mathbf{q})$ only if $\mathbf{q} = \mathbf{p} + (*, 1, 0)$ where $*$ denotes any integer. For example, if $\mathbf{p} = (5, 3, 7)$, then all possible values for $\mathbf{q}$ that satisfy $h_{u1}(\mathbf{p}) = h_{u2}(\mathbf{q})$ are $\mathbf{q} = (x, 4, 7)$ where x is any integer.

This suggests the following definition:

Definition 2.4.1 ($\langle f, g\rangle$ set)**.** For any pair of occurrences f, g of a generated variable in the loop, define the $\langle f, g\rangle$ set as:

$$\langle f, g\rangle = \{\mathbf{x} \mid \mathbf{x} \in \mathbb{Z}^n, \exists\, \mathbf{p} \in \mathbb{Z}^n : h_f(\mathbf{p}) = h_g(\mathbf{p} + \mathbf{x})\}. \tag{2.4.2}$$

We observe that the $\langle f, g\rangle$ set is independent of the index space $\mathcal{IS}$. In our previous example, $\langle u1, u2\rangle = \{(x, 1, 0) \mid x \in \mathbb{Z}\}$. All the $\langle f, g\rangle$ sets for alg. 2.1 and alg. 2.3 are listed in table 2.1 and table 2.2 respectively.

We now recall that $t(\mathbf{p} + \mathbf{x}) = t(\mathbf{p}) + t(\mathbf{x})$, since we have assumed t to be a linear function (eq. (2.3.8)). Also, $\mathbf{p} \ll \mathbf{p} + \mathbf{x}$ if and only if $\mathbf{x} \gg 0$. We can then substitute $\mathbf{p} + \mathbf{x}$ for $\mathbf{q}$ in condition (C1) and obtain the following condition:

Sets	**Elements $\gg 0$**	**Constraints**
$\langle u1, u1\rangle = (*, 0, 0)$	$(+, 0, 0)$	$\tau_1 > 0$
$\langle u1, u2\rangle = (*, 1, 0)$	$(+, 1, 0)$ $(0, 1, 0)$	$\tau_1 + \tau_2 > 0$ $\tau_2 > 0$
$\langle u2, u1\rangle = (*, -1, 0)$	$(+, -1, 0)$	$\tau_1 - \tau_2 > 0$
$\langle u1, u3\rangle = (*, 0, 1)$	$(+, 0, 1)$ $(0, 0, 1)$	$\tau_1 + \tau_3 > 0$ $\tau_3 > 0$
$\langle u3, u1\rangle = (*, 0, -1)$	$(+, 0, -1)$	$\tau_1 - \tau_3 > 0$
$\langle u1, u4\rangle$	same as $\langle u2, u1\rangle$	
$\langle u4, u1\rangle$	same as $\langle u1, u2\rangle$	
$\langle u1, u5\rangle$	same as $\langle u3, u1\rangle$	
$\langle u5, u1\rangle$	same as $\langle u1, u3\rangle$	

Table 2.2: Table showing the $\langle f, g\rangle$ sets, elements greater than 0, and schedule constraints of alg. 2.3

(C1′) For every variable and every ordered pair of occurrences f, g of that variable, at least one of which is a generation: for each $\mathbf{p}, \mathbf{p}+\mathbf{x} \in \mathcal{IS}$ with $\mathbf{x} \gg 0$ such that $h_f(\mathbf{p}) = h_g(\mathbf{p}+\mathbf{x})$, the schedule t must satisfy the relation $t(\mathbf{x}) > 0$.

Substituting the clause "for each $\mathbf{p}, \mathbf{p}+\mathbf{x} \in \mathcal{IS}$ such that $h_f(\mathbf{p}) = h_g(\mathbf{p}+\mathbf{x})$" with "for each $\mathbf{x} \in \langle f, g\rangle$" from (C1′) gives us a the following stronger condition for t to satisfy:

(C2) For every variable and every ordered pair of occurrences f, g of that variable, at least one of which is a generation: for each $\mathbf{x} \in \langle f, g\rangle$ with $\mathbf{x} \gg 0$, the schedule t must satisfy the relation $t(\mathbf{x}) > 0$.

Finding a schedule that satisfies (C2) gives us a valid schedule as in definition 2.3.6. Moreover, (C2) is independent of the index space $\mathcal{IS}$. Each condition $t(\mathbf{x}) > 0$ gives us a constraint on the values $\tau_1, \ldots, \tau_n$ in the form:

$$t(\mathbf{x}) = \tau_1 x_1 + \ldots + \tau_n x_n > 0\ . \tag{2.4.3}$$

Computing the $\langle f, g\rangle$ sets

Recalling assumption (A5), say that an occurrence f of a generated variable is of the form eq. (2.4.1) and another occurrence g of the same variable is of the similar form $\mathsf{VAR}(i_{j_1} + l_1, \ldots, i_{j_r} + l_r)$. Then $h_f(p_1, \ldots, p_n) = (p_1 + m_1, \ldots, p_r + m_r)$, and $h_g(p_1, \ldots, p_n) = (p_1 + l_1, \ldots, p_r + l_r)$. It is easy to see from definition 2.4.1 that $\langle f, g \rangle$ is the set of all elements of $\mathbb{Z}^n$ whose j_kth coordinate is $m_k - l_k$ for $k = 1, \ldots, r$ and whose remaining $n - r$ coordinates are any integers. This gives us a way to compute and describe the $\langle f, g \rangle$ set with the tuple $(x_1, \ldots, x_n)$ where x_{j_k} is either the integer $m_k - l_k$ or $*$ denoting any possible integer.

The index variable i_j is said to be *missing* from the variable VAR if i_j is not one of the i_{j_k} in eq. (2.4.1). In other words, i_j is missing from VAR if the $\langle f, g \rangle$ set has an $*$ in the jth coordinate, for any pair of occurrences f, g of VAR.

We call i_j a *missing index* if it is missing from some generated variable in the loop.

2.4.3 The hyperplane theorem

The hyperplane theorem was introduced by Lamport in [Lam74] and proves the existence of a valid schedule for a loop in the form of listing 2.1 satisfying the assumptions (A1)-(A5). We use a special case of the theorem in which the following assumption is added:

(A6) None of the index variables $i_2, \ldots, i_n$ in the loop is a missing index.

In this way, each of the $\langle f, g \rangle$ sets can be described either by $(*, x_2, \ldots, x_n)$ or $(x_1, \ldots, x_n)$ where the x_j are integers.

We can split the descriptor $(*, x_2, \ldots, x_n)$ in two by considering the values of $x_1 > 0$ and $x_1 = 0$, using the notation $(+, x_2, \ldots, x_n)$ for the former case and $(0, x_2, \ldots, x_n)$ for the latter. This can be done since we are only interested in descriptors that are lexicographically greater than 0. As show in the theorem, we can substitute the "+" with a 1, because $t(1, x_2, \ldots, x_n) > 0$ implies $t(x_1, x_2, \ldots, x_n)$ for all $x_1 > 0$ if we assume that the $\tau_1, \ldots, \tau_n$ representing the schedule are all nonnegative integers. These descriptors $\mathbf{x}$ are called *dependence vectors*.

Say that there are r dependence vectors $\mathbf{x}^k \gg 0$ in all the $\langle f, g \rangle$ sets described in definition 2.4.1. Then the schedule t must satisfy all the constraints in the form:

$$t(\mathbf{x}^k) = \tau_1 x_1^k + \ldots + \tau_n x_n^k > 0, \tag{2.4.4}$$

for each $k = 1, \ldots, r$. A schedule that satisfies these constraints also satisfies the condition (C2), thus being a valid schedule. We call $D \in \mathbb{Z}^{r \times n}$ the *dependence matrix* composed by the dependence vectors $\mathbf{x}^1, \ldots, \mathbf{x}^r$:

$$D = \begin{bmatrix} x_1^1 & \ldots & x_n^1 \\ \vdots & \ddots & \vdots \\ x_1^r & \ldots & x_n^r \end{bmatrix} \tag{2.4.5}$$

All the elements $\gg 0$ and the constraints on the $\tau_1, \dots, \tau_n$ for alg. 2.1 and alg. 2.3 are listed in table 2.1 and table 2.2 respectively.

Note that the choice of $\tau_1 = \tau_2 = 1$ for alg. 2.1 satisfies all the constraints in table 2.1. We will later choose $\tau_1 = 2, \tau_2 = \tau_3 = 1$ for alg. 2.3, that also satisfy all the constraints in table 2.2.

Finding an optimal schedule

Lamport also considers the problem of finding an optimal schedule, specifically the schedule that *minimizes* the number of iterations in the sequential outer loop of listing 2.2. If a sufficiently large number of processors is available, this gives the maximum amount of concurrent computation. This means that we need to minimize the value $\lambda_1 - \mu_1$. Since λ_1 and μ_1 are the lower and upper bounds of the target polyhedron in the first dimension, it is easy to see that:

$$\mu_1 - \lambda_1 = (u_1 - \ell_1)|\tau_1| + \dots + (u_n - \ell_n)|\tau_n|, \tag{2.4.6}$$

where the $\ell_1, \dots, \ell_n$ and $u_1, \dots, u_n$ are the original lower and upper bounds of the starting loop as in listing 2.1. This substitution leads us to the following minimization objective:

$$\mathit{minimize} \sum_{i=1}^{n} (u_i - \ell_i)|\tau_i|. \tag{2.4.7}$$

Since the lower and upper bounds of the starting loop may not be known at compile time, another reasonable objective is to minimize the sum of the absolute value of the coefficients of the schedule t:

$$\mathit{minimize} \sum_{i=1}^{n} |\tau_i|. \tag{2.4.8}$$

Intuitively, minimizing this value also minimizes the maximum value that the schedule function can reach, which is also known as the *latency* of the schedule.

If we assume that the coefficients are all nonnegative integers, then the problem of finding a valid, minimal latency schedule can be expressed by the following *integer linear programming* problem:

$$\begin{aligned} \mathit{minimize} \quad & \sum_{i=1}^{n} \tau_i \\ \mathit{subject\ to} \quad & \tau \in \mathbb{Z}^n, \tau_j \geq 0, \\ & D\tau > 0, \end{aligned} \tag{2.4.9}$$

where D is the dependence matrix of the form in eq. (2.4.5). Integer linear programming (ILP) problems can be efficiently solved by iterative algorithms like the Simplex method, combined with techniques like the *cutting plane* method or the *branch and bound* method for finding integer solutions. In chapter 4 we discuss the tools we used to solve these kinds of problems.

For example, consider the problem of finding a schedule for alg. 2.1. Given the constraints in table 2.1, the schedule which minimizes the optimization objective is $\lambda = \begin{bmatrix}1 & 1\end{bmatrix}$. Similarly, considering alg. 2.3 and its constraints in table 2.2, the optimal schedule is $\lambda = \begin{bmatrix}2 & 1 & 1\end{bmatrix}$.

2.4.4 Completing the transformation

We now need to complete the transformation by choosing a matching allocation for the given schedule. In section 2.3.2 we stated the minimal requirement for a valid allocation, specifically that the time-space matrix T has $\det(T) \neq 0$. We restrict this requirement by choosing an allocation a such that T is unimodular (see definition 2.3.3). Unimodular matrices have the property of defining *one-to-one mappings*, such that each point in the initial $\mathbb{Z}$-polyhedron can be mapped to a point in the target one. This greatly simplifies the task of scanning the polyhedron (see section 2.5) since the matrix T^{-1} has only integer elements and there are no "holes" to be taken care of inside the target polyhedron.

There are ways to deal with non-unimodular matrices, usually by building its Hermite normal form $H = TU$ where U is unimodular. More details on how to deal with non-unimodularity are presented in [Len93] and [Fea96].

A classical number theoretic calculation described in [Mor69, p.31], gives us a completion of the schedule vector $\tau = (\tau_1 \ldots, \tau_n)$ into a unimodular matrix. We do not go into the details of this calculation, but it can be proven that we can always find a unimodular matrix completion given *relatively prime* integer coefficients of the first row. It is easy to see that the $\tau_1, \ldots, \tau_n$ resulting from the solution of eq. (2.4.9) are relatively prime, i.e. their greatest common divisor is 1, because we can always divide the τ_i by their g.c.d., giving new values of τ_i satisfying the constraints for a smaller value of the minimization objective.

Since there are multiple allocation matrices A that complete the schedule vector τ into a unimodular matrix, we also consider the problem of finding an *efficient* allocation. There are numerous optimization objectives one can impose on the construction of A, such as maximizing spatial locality and many others. We considered the objective of finding the allocation minimizing the sum of all non-zero elements of A and their distance to the diagonal of T. Trying to make T sparse and diagonal also helps T^{-1} being more sparse and diagonal, hence reducing the number of non-zero coefficients of the inequality matrix of the target polyhedron. Simpler inequalities lead to simpler bounds for the target loop, hence fewer computations to be made during code execution.

This optimization objective, as well as the constraint that T is unimodular, can be represented by the following *constraint optimization* problem:

$$\begin{aligned} &\textit{minimize} && \sum_{i=1}^{n-1}\sum_{j=1}^{n} \alpha_{i,j}(1+|i-j|) \\ &\textit{subject to} && A \in \mathbb{Z}^{(n-1)\times n}, \alpha_{i,j} \geq 0, \\ & && |\det(T)| = 1, \end{aligned} \tag{2.4.10}$$

where $T = \begin{bmatrix} \tau \\ A \end{bmatrix}$. Constraint optimization problems are a generalization of the more classic constraint satisfaction problems, which can be solved by a multitude of techniques. In chapter 4 we show how we used a solver that efficiently solves these kinds of problems for relatively small inputs.

For example, consider the problem of finding an allocation for alg. 2.1 given the optimal schedule $\lambda = \begin{bmatrix} 1 & 1 \end{bmatrix}$. The allocation which minimizes the optimization objective is $A = \begin{bmatrix} 0 & 1 \end{bmatrix}$, giving the transformation matrix $T = \begin{bmatrix} 1 & 1 \\ 0 & 1 \end{bmatrix}$. Similarly, considering alg. 2.3 and its optimal schedule $\lambda = \begin{bmatrix} 2 & 1 & 1 \end{bmatrix}$, the optimal allocation is $A = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 0 & 1 \end{bmatrix}$, giving the transformation matrix $T = \begin{bmatrix} 2 & 1 & 1 \\ 1 & 0 & 0 \\ 0 & 0 & 1 \end{bmatrix}$.

2.5 Scanning polyhedra

Once we found a valid transformation, we then need to compute the bounds for the new index variables for the target loop in the form of listing 2.2. This process is also known as *polyhedral scanning*, i.e. enumerating the points in the polyhedron by "scanning" the dimensions one at a time.

Specifically, we want to represent each lower and upper bound of the jth nested loop as a function of the previous $j-1$ index variables. Recall the construction of the target polyhedron in eq. (2.3.12). The target polyhedron is bounded by the system of inequalities $CT^{-1}\mathbf{i}' \leq \mathbf{d}$, where $C\mathbf{i} \leq \mathbf{d}$ are the boundaries of the starting polyhedron. We must transform this polyhedron into an equivalent one of the form $C'\mathbf{i}' \leq \mathbf{d}'$, whose defining inequalities refer only to the indices of enclosing loops, i.e. the two halves of C' are lower triangular.

Several algorithms for calculating C' and $\mathbf{d}'$ have been proposed and we opted for the simpler Fourier-Motzkin elimination algorithm. The Fourier-Motzkin elimination algorithm is a method for iteratively eliminating a variable from a system of linear inequalities. The elimination of a variable from the system refers to the creation of another system, but without the eliminated variable, such that both systems have the same solutions over the remaining variables. A detailed explanation of the Fourier-Motzkin elimination is given in the next section.

2.5.1 Fourier-Motzkin elimination

Consider a system $\mathcal{S}$ of n inequalities with r variables $x_1, \ldots, x_r$, with the variable x_r to be eliminated, described by $A\mathbf{x} \leq \mathbf{b}$. Each of the linear inequalities in the system can be classified in one of these three groups, depending on the sign of the coefficient $a_{i,r}$ for the i-th inequality:

1. inequalities that are of the form
$$x_r \geq \frac{b_i - \sum_{k=1}^{r-1} a_{i,k}x_k}{a_{i,r}},$$
denoting them by $x_r \geq L_j(x_1, \ldots, x_{r-1}) = L_j$ for each $1 \leq j \leq n_L$ where n_L is the number of such inequalities;

2. inequalities that are of the form
$$x_r \leq \frac{b_i - \sum_{k=1}^{r-1} a_{i,k}x_k}{a_{i,r}},$$
denoting them by $x_r \leq U_j(x_1, \ldots, x_{r-1}) = U_j$ for each $1 \leq j \leq n_U$ where n_U is the number of such inequalities;

3. inequalities in which x_r plays no role (i.e. $a_{i,r} = 0$), denoting them by the system $\mathcal{E}$ of $n - n_L - n_U$ inequalities and grouping them into a single conjunction $\varnothing$.

The original system is thus equivalent to:

$$\max(L_1, \ldots, L_{n_L}) \leq x_r \leq \max(U_1, \ldots, U_{n_L}) \wedge \varnothing. \tag{2.5.1}$$

Elimination consists in producing a system equivalent to $\exists x_r\, \mathcal{S}$. This formula is equivalent to:

$$\max(L_1, \ldots, L_{n_L}) \leq \max(U_1, \ldots, U_{n_L}) \wedge \varnothing, \tag{2.5.2}$$

where the inequality:

$$\max(L_1, \ldots, L_{n_L}) \leq \max(U_1, \ldots, U_{n_L}), \tag{2.5.3}$$

is equivalent to $n_L n_U$ inequalities $L_i \leq U_j$ for $1 \leq i \leq n_L, 1 \leq j \leq n_U$.

We have therefore transformed the original system into another system where x_r is eliminated.

Note that the output system has $(n - n_L - n_U) + n_L n_U$ inequalities.

2.5.2 Bounds extraction

We can apply Fourier-Motzkin elimination iteratively to eliminate each index variable, from the innermost loop to the outermost one, and extract the lower and upper bounds of that variable at each iteration. The algorithm is described by alg. 2.4.

Algorithm 2.4: Fourier-Motzkin elimination

```
Input  : A system $\mathcal{S}$ of $n$ inequalities in $r$ variables $x_1, \ldots, x_r$.
Output: The integer bounds $\lambda_i$ and $\mu_i$ of each variable $x_i$, such that they
        depend only on the previous variables $x_1, \ldots, x_{i-1}$.
1 $\mathcal{S}_r \leftarrow \mathcal{S}$
2 for $i$ from $r$ to $1$ do
3     Divide the inequalities in $\mathcal{S}_i$ regarding $x_i$ into the $L_j$, $U_j$ and $\mathcal{E}$
4     $\lambda_i \leftarrow \lceil \max(L_1, \ldots, L_{n_L}) \rceil$
5     $\mu_i \leftarrow \lfloor \min(U_1, \ldots, U_{n_U}) \rfloor$
6     $\mathcal{S}_{i-1} \leftarrow \{L_j \leq U_k \mid 1 \leq j \leq n_L, 1 \leq k \leq n_U\} \cup \mathcal{E}$
7     Eliminate inequalities in $\mathcal{S}_{i-1}$ that have no index variable
8 end
9 return $\lambda, \mu$
```

Note that the L_j and U_j can be rational if fractions are present in the expressions, so we apply *ceiling* and *floor* functions to get integer bounds.

Since running an elimination step on a system with n inequalities yields at most $n^2/4$ inequalities, running r successive steps can result in at most $4(n/4)^{2^r}$ inequalities, a double exponential complexity in the order of $O(n^{2^r})$. This is because the Fourier-Motzkin algorithm produces unnecessary constraints, i.e. constraints that are implied by other constraints. There exist more sophisticated algorithms that permit the elimination of unnecessary constraints, but they are beyond the scope of this thesis. For practical purposes, we can eliminate inequalities where no index variable is present since we know that they are true if each lower bound of the starting loop nest is less than or equal to its corresponding upper bound.

Consider the problem of finding the bounds for the transformation $T = \begin{bmatrix} 1 & 1 \\ 0 & 1 \end{bmatrix}$, $T^{-1} = \begin{bmatrix} 1 & -1 \\ 0 & 1 \end{bmatrix}$ of alg. 2.1. We rename i'_1 to i' and i'_2 to j'. The target polyhedron can be described by the system of inequalities:

$$\begin{cases} i' - j' \geq 1 \\ j' \geq 1 \\ i' - j' \leq N \\ j' \leq N \end{cases}$$

In the first step, we isolate the index variable j', obtaining the system:

$$\begin{cases} j' \geq 1 \\ j' \geq i' - N \\ j' \leq i' - 1 \\ j' \leq N \end{cases}$$

where the first two inequalities are of the group L_j and the second one of the group U_j. The lower bound for j' is $\lambda_2 = \max(1, i' - N)$, while the upper bound is $\mu_2 = \min(i' - 1, N)$. Note that ceiling or floor functions are not needed here since there are no fractions in the bounds. We then eliminate the variable j' by combining each couple of the L_j and U_j in the system:

$$\begin{cases} 1 \leq i' - 1 \\ 1 \leq N \\ i' - N \leq i' - 1 \\ i' - N \leq N \end{cases}$$

and after removing inequalities where no index variable is present and simplifying:

$$\begin{cases} i' \geq 2 \\ i' \leq 2N \end{cases}$$

Note that the variable j' is gone, and the resulting system represent the bounds for the last index variable i', namely $\lambda_1 = 2$ and $\mu_1 = 2N$.

Another example is considering the problem of finding the bounds for the transformation $T = \begin{bmatrix} 2 & 1 & 1 \\ 1 & 0 & 0 \\ 0 & 0 & 1 \end{bmatrix}$, $T^{-1} = \begin{bmatrix} 0 & 1 & 0 \\ 1 & -2 & -1 \\ 0 & 0 & 1 \end{bmatrix}$ of alg. 2.3. We rename i'_1 to i', i'_2 to j' and i'_3 to k'. The target polyhedron can be described by the system of inequalities:

$$\begin{cases} j' \geq 1 \\ i' - 2j' - k' \geq 2 \\ k' \geq 2 \\ j' \leq L \\ i' - 2j' - k' \leq M \\ k' \leq N \end{cases}$$

In the first step, we isolate the index variable k', obtaining the system:

$$\begin{cases} k' \geq i' - 2j' - M \\ k' \geq 2 \\ k' \leq i' - 2j' - 2 \\ k' \leq N \\ j' \geq 1 \\ j' \leq L \end{cases}$$

where the first two inequalities are of the group L_j, the third and the fourth are of the group U_j and the last two are of the system $\mathcal{E}$. The lower bound for k' is $\lambda_3 = \max(2, i' - 2j' - M)$, while the upper bound is $\mu_3 = \min(N, i' - 2j' - 2)$. We then eliminate the variable k' by combining each couple of the L_j and U_j and adding the remaining ones of the system $\mathcal{E}$:

$$\begin{cases} i' - 2j' - M \le i' - 2j' - 2 \\ i' - 2j' - M \le N \\ 2 \le i' - 2j' - 2 \\ 2 \le N \\ j' \ge 1 \\ j' \le L \end{cases}$$

and after removing inequalities where no index variable is present and simplifying, we get:

$$\begin{cases} i' - 2j' \ge 4 \\ i' - 2j' \le N + M \\ j' \ge 1 \\ j' \le L \end{cases}$$

We can apply the second step by isolating the variable j', obtaining the system:

$$\begin{cases} j' \ge \frac{i'-N-M}{2} \\ j' \ge 1 \\ j' \le \frac{i'-4}{2} \\ j' \le L \end{cases}$$

Again, we compute the lower bound $\lambda_2 = \left\lceil \max(1, \frac{i'-N-M}{2}) \right\rceil$ and the upper bound $\mu_2 = \left\lfloor \min(L, \frac{i'-4}{2}) \right\rfloor$ of the variable j' and we eliminate it, obtaining (after removals and simplifications):

$$\begin{cases} i' \ge 6 \\ i' \le 2L + N + M \end{cases}$$

which are the bounds of the last index variable i'.

2.5.3 Code generation

The last step is to generate the loop body of the transformed loop as in listing 2.2. We can substitute each occurrence of the index variable i_j by applying the relative transformation row of the space-time matrix T such that:

$$i_j = T_j^{-1}\mathbf{i}', \qquad (2.5.4)$$

for each $1 \leq j \leq n$, where T_j^{-1} is the jth row of the inverted transformation matrix. In a more compact form:

$$\mathbf{i} = T^{-1}\mathbf{i}'. \tag{2.5.5}$$

For example, consider the transformation matrix for alg. 2.3 $T = \begin{bmatrix} 2 & 1 & 1 \\ 1 & 0 & 0 \\ 0 & 0 & 1 \end{bmatrix}$, $T^{-1} = \begin{bmatrix} 0 & 1 & 0 \\ 1 & -2 & -1 \\ 0 & 0 & 1 \end{bmatrix}$, the indexes substitution is:

$$\mathbf{i} = T^{-1}\mathbf{i}' = \begin{bmatrix} i'_2 \\ i'_1 - 2i'_2 + i'_3 \\ i'_3 \end{bmatrix} = \begin{bmatrix} j' \\ i' - 2j' + k' \\ k' \end{bmatrix}$$

Putting together the bounds calculated in the previous section, we obtain the rewritten loop in alg. 2.5.

Algorithm 2.5: SIMPLIFIED 2D RELAXATION (PARALLEL)

```
1 for $i'$ from 6 to $2L + N + M$ do
2     for conc $j'$ from $\left\lceil \max(1, \frac{i'-N-M}{2}) \right\rceil$ to $\left\lfloor \min(L, \frac{i'-4}{2}) \right\rfloor$ do
3         for conc $k'$ from $\max(2, i' - 2j' - M)$ to $\min(N, i' - 2j' - 2)$ do
4             $U(i' - 2j' - k', k') \leftarrow (U(i' - 2j' - k' - 1, k') + U(i' - 2j' - k', k' -$
              $1) + U(i' - 2j' - k' + 1, k') + U(i' - 2j' - k', k' + 1)) * 0.25$
5         end
6     end
7 end
```

Chapter 3

Design

In this chapter, we discuss the ideas that lead to the design of OPoly. While designing OPoly, we wanted to achieve the following quality requirements:

- *generality*: to be free from unnatural restrictions or limitations that are not constraints of the problem itself;
- *modularity*: to separate the program's functionalities into independent, interchangeable modules, that can be replaced or updated without affecting the whole program;
- *extensibility*: to be able to extend or add program's functionalities most effortlessly.

Two main issues arose from the analysis:

- how to effectively *represent programs*, especially for loops, in a general, but effective way, such that OPoly can analyze them and generate new ones from scratch;
- how to organize the *architecture* of OPoly to achieve the desired quality requirements.

In the following sections, we provide the solutions we adopted for solving these issues: in section 3.1 we show how OPoly can represent for loops using an *object-oriented programming* representation; in section 3.2 we illustrate the OPoly architecture and how we modularized it.

3.1 Program representation

The first main issue we needed to address is how to represent programs, more precisely for loops and assignment statements. To achieve generality and extensibility, we wanted to be able to express any for loop in the form of listing 2.1 without adding limitations to its structure or content.

We choose to represent a for loop by composing the following parts:

- an *index variable*, the iteration dimension of the loop;
- the lower and upper *bounds*, the iteration space of the loop;
- an (optional) *step*, the increment of the index variable at each iteration;
- a list of *statements*, the loop body.

We introduce two concepts that we use to model the loop's components:

- *statements*, a program's syntactic unit that expresses some action to be carried out. These can be *simple*, if they cannot contain other statements (e.g. assignments, declarations, etc), or *compound* if they can contain other statements (e.g. loops, if-statements, etc);
- *expressions*, a program's syntactic entity that can be evaluated to determine its value. It combines one or more *constants*, *variables* or *functions*, with *operators* describing the relations between them.

A for loop is some kind of statement, specifically a compound statement, since it has a loop body that contains other statements.

Statements that assign or update a value of some variables in the program are called *assignment* statements. Assignments are composed of two parts: the left part, which is the variable that is being assigned (or one of its elements if it is an array variable), and the right part, which is an expression, usually involving multiple constants and variables.

By composing for loop statements and assignments, we can represent any loop in the form of listing 2.1 and some other cases of not perfectly nested loops, which can be treated in PC with some more advanced techniques, thus enabling our representation to be used by possible extensions of OPoly. In our current implementation, we do not define other useful statements, like *if-then-else* statements, but they can easily be implemented by reusing or extending existent statements and expressions.

From now on, we use the camel case `ClassName` to denote a class and the lowercase with underscores `attr_name` to denote an attribute of a class.

3.1.1 Expressions

Expressions are the building blocks for representing complex formulas within a program. The main concept is that an `Expression` is made up by other expressions (*terms*) concatenated by *operators*. That is, if an expression has n terms, $n-1$ operators are required to complete the expression, represented by the following form:

$$\texttt{term}_1\ \texttt{op}_1\ \ldots\ \texttt{op}_{n-1}\ \texttt{term}_n\ .$$

For our specific use case, operators are just strings. One can argue that operators should be modeled as a different entity with its own set of rules, but we decided to simplify this by letting any string be an operator, delegating the responsibility of checking its correct format to the specific implementation of language-dependent modules (see section 3.2).

An `Expression` can (not required) be of one of the following three types:

- `SingleExpression`, an expression having only one term (itself);
- `UnaryExpression`, an expression with an additional unary operator in front of itself;
- `GroupingExpression`, an expression enclosed by two parenthesis.

Furthermore, a `SingleExpression` can be categorized as:

- `VariableExpression`, an expression representing a program variable. It may have a list of expressions, each one representing the value of the array index;
- `ConstantExpression`, an expression which has a constant value (integer or floating point);
- `FunctionExpression`, an expression representing a function with a list of arguments which are also expressions.

In fig. 3.1 is depicted the class diagram representing the relations between different types of expressions.

3.1.2 Statements

Statements express some actions that need to be carried out by the program, for example assigning a value to a variable. In real program syntax, there can be a lot of different statements, but we decided to model only the ones required to represent the loop format in listing 2.1 and also the rewritten loop format in

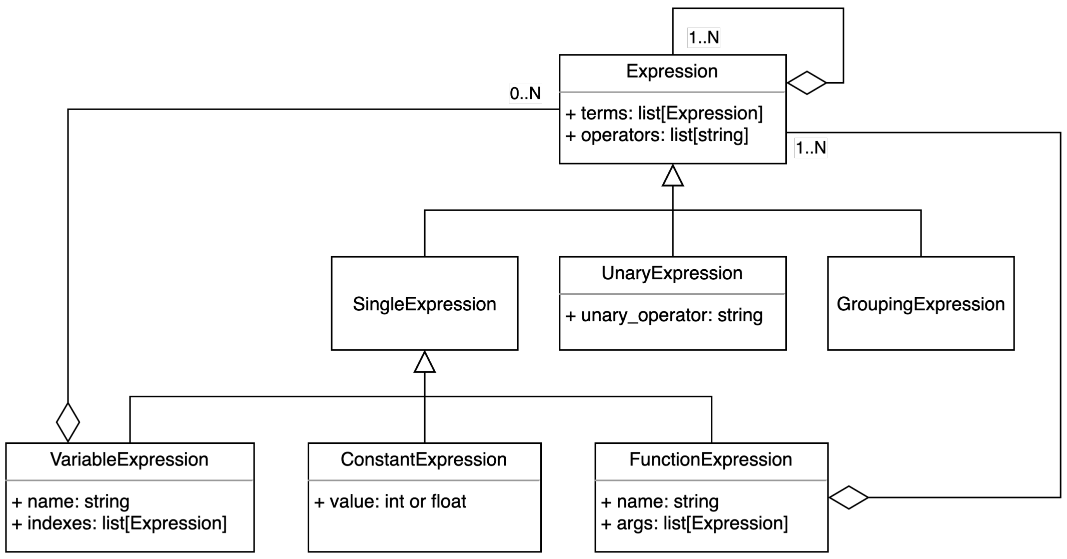


Figure 3.1: Expressions class diagram.

listing 2.2, which essentially needs no more different types of statements from the former, but we define one more statement (the `DeclarationStatement`) which is useful for rewriting the loop.

We basically need two (plus one) types of `Statement`:

- `AssignmentStatement`, a statement representing a variable assignment. We consider the following form:

  `left_term` = `right_term`,

  where a `left_term` is a `VariableExpression`, and `right_term` is a general `Expression`;

- `ForLoopStatement`, a statement representing a for loop. It derives from the class `CompoundStatement`, meaning it contains other statements, specifically the ones in the loop `body`. A for loop has also an `index`, which is a `VariableExpression`; the `lowerbound`, `upperbound` and `step` that define the loop iteration space (each of them is a general `Expression`) and a flag `is_parallel` that denotes if the loop can be executed concurrently or not.

- `DeclarationStatement`, a statement representing a variable declaration. We consider the following form:

  `var_type` `variable` = `initialization`,

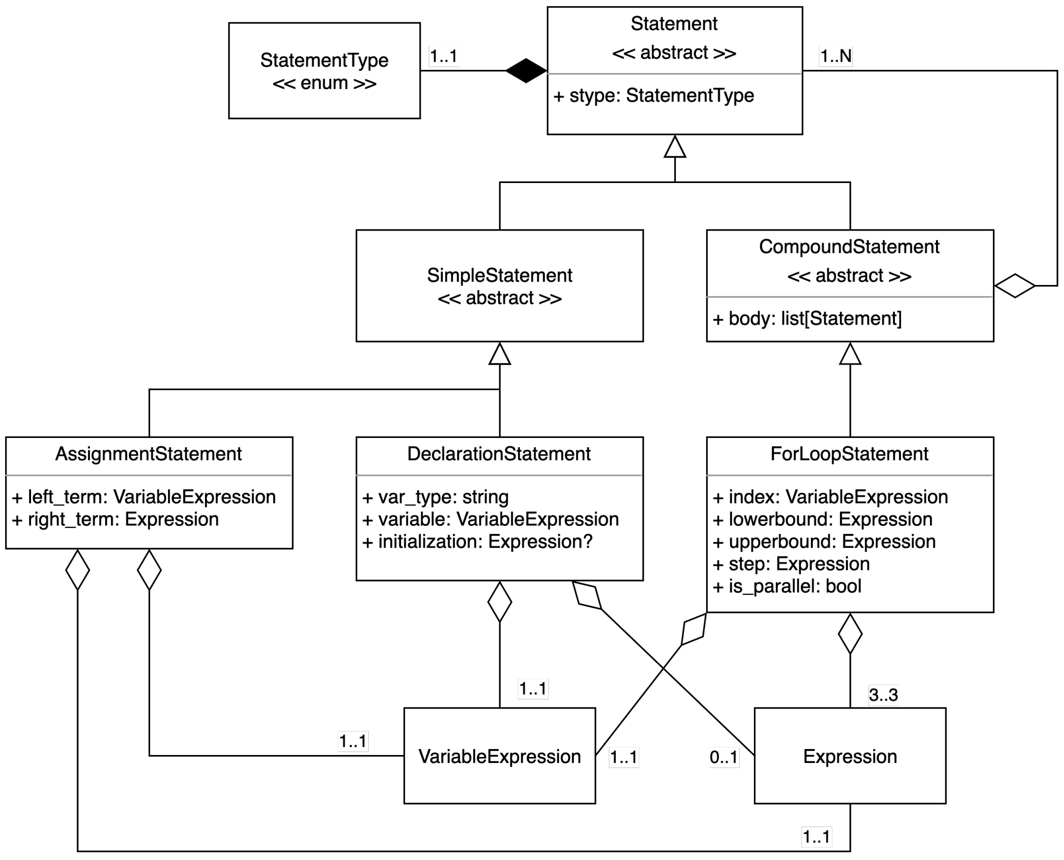


Figure 3.2: Statements class diagram.

where `var_type` is a string representing the type of the declared variable, `variable` is a `VariableExpression`, and `initialization` is an optional `Expression` representing the initial value of `variable`.

Since `Statement` is an abstract class, the only classes that can be instantiated are the ones described above. It makes sense to also have a reference to the type `StatementType` that a `Statement` must have. In our case, is an *enumeration* with three possible values: `FOR_LOOP`, `DECLARATION` and `ASSIGNMENT`.

We also modeled the abstract classes `SimpleStatement` and `CompoundStatement` for future classes extensions.

In fig. 3.2 is depicted the class diagram representing the relations between different types of statements and some expressions that are parts of statements.

3.2 Architecture

The architecture of OPoly involves a *pipeline* of *modules* that takes as input the source code with the loop to optimize and outputs the generated parallel code. In fig. 3.3 is depicted the structure of the pipeline. From now on, we use the small caps Module to denote a module.

Each module is designed to solve a particular problem, for example, the Parser module has the task of parsing the input source code to generate the ForLoopStatement object described in section 3.1.2. In this way, each module is independent and fully replaceable, given that it maintains the same interface with the previous and successive modules.

The Parser and Generator modules are the only *language dependent* modules, meaning that they depend on the language of, respectively, the source code and the generated code. All other modules use the classes described in section 3.1 to perform their tasks, so they are independent of the source and target language. These properties further increase the modularity of OPoly and reduce the burden for the implementation of future extensions.

In the next paragraphs, we describe each module in detail, from the first in the pipeline (Parser) to the last one (Generator).

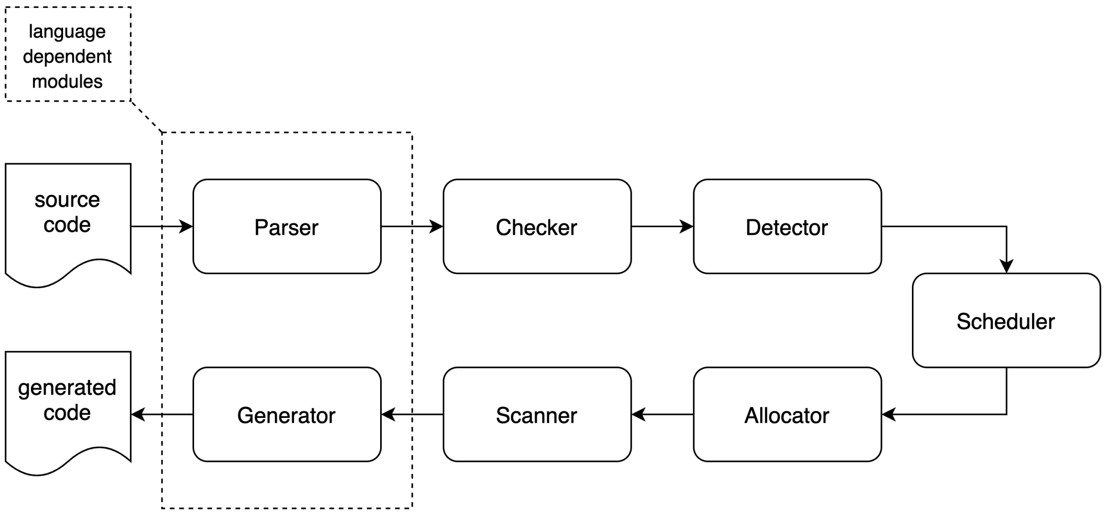


Figure 3.3: OPoly pipeline diagram.

3.2.1 Parser

The Parser module has the task of parsing the source code to generate one or more ForLoopStatement described in section 3.1. Since the ForLoopStatement is intrinsically recursive, a nested loop in the form of listing 2.1 can be represented by a single ForLoopStatement with (possibly) other ones nested within it.

At the time of writing, only one implementation of this module exists, namely the PseudocodeForLoopParser, which parses a single (nested) loop written in the OPoly pseudocode language later described in section 4.3. Parsers supporting other languages may be integrated into the pipeline almost effortlessly, given that they produce as output a ForLoopStatement object.

If the source code has syntax errors, the Parser returns an error, interrupting the execution of the pipeline and generating a parser error that is shown by OPoly.

3.2.2 Checker

The Checker module has the task of checking that the ForLoopStatement produced by the Parser module is compliant with some assumptions or restrictions.

For our purpose, the assumptions (A1)-(A6) described in section 2.1 and section 2.4 are checked by the module's implementation LamportForLoopChecker.

If the assumptions are not met, the Checker returns an error, interrupting the execution of the pipeline and generating a checker error that is shown by OPoly.

3.2.3 Detector

The Detector module has the task of detecting the dependencies among the loop's generated variables. In particular, it generates a number of dependencies of the form eq. (2.4.3) from a ForLoopStatement. These dependencies constitute the dependence matrix (see eq. (2.4.5)) needed to find a valid schedule.

The Detector module assumes that the ForLoopStatement has been checked by a particular Checker before generating the dependencies, so no errors should be returned by this module. This means that the Checker and Detector modules are somewhat coupled; in fact, one might argue they can be merged in one single module. We decided not to do that since the two tasks are quite different from each other.

For our purpose, we implemented the LamportLoopDependenciesDetector, which generates the dependencies using the notion of $\langle f, g \rangle$ sets (see definition 2.4.1).

3.2.4 Scheduler

The Scheduler module has the task of finding an optimal, valid schedule (see definition 2.3.6 and section 2.4.3) from the loop's dependencies. This means solving the integer linear programming problem described by eq. (2.4.9). x The Scheduler takes as input the dependence matrix and returns the coefficients of the found schedule, i.e. the schedule vector.

We implemented the LamportCPScheduler, which uses a solver for *constraint programming* problems to solve the integer programming problem of finding an optimal schedule. For more information about the actual implementation, see section 4.2

3.2.5 Allocator

The Allocator module has the task of finding the allocation (see definition 2.3.7) associated with the schedule found by the Scheduler module. This means solving the constraint programming problem described by eq. (2.4.10).

The Allocator takes as input the schedule vector and returns the full transformation matrix, where the first row is the schedule vector and the rest of the matrix is the allocation.

We implemented the LamportCPScheduler, which uses a solver for the aforementioned constraint programming problem. For more information about the actual implementation, see section 4.2.

3.2.6 Scanner

The Scanner module has the task of scanning the target polyhedron to extract the lower and upper bounds for the transformed index variables of the target loop, and to transform the source ForLoopStatement into an equivalent, but parallel, one in the form of listing 2.2.

The Scanner takes as input the source ForLoopStatement and the transformation matrix, and returns the target ForLoopStatement.

We implemented the FourierMotzkinScanner, that applies the Fourier-Motzkin elimination algorithm (see section 2.5) to the target polyhedron constructed as eq. (2.3.12), and generates the target loop by creating the nested ForLoopStatement with the new index variables, bounds, and declarations of the old index variables in the body loop by applying the inverse transformation to the new indexes. This is where the DeclarationStatement comes in handy because we just need to declare the old index variables instead of substituting them with new index variables expressions for every occurrence in the statements of the loop body.

3.2.7 Generator

The Generator module has the task of generating the code for the target loop.

The GENERATOR module takes as input the target ForLoopStatement and returns a string of code representing the generated loop.

At the time of writing, two implementations of this module are available: the PSEUDOCODEGENERATOR, which generates the target loop in OPoly pseudocode language, and the CCODEGENERATOR, which generates the target loop in C language with correctly placed `#pragma omp parallel for` OpenMP directives where needed. For more information on the actual implementation of the CCODEGENERATOR, see section 4.4.

Chapter 4

Implementation

In this chapter, we discuss the implementation details of OPoly, by introducing the tools used for its implementations: the Python programming language and the MiniZinc constraint programming modeling language. At the time of writing, the implementation of OPoly can be found on GitHub [1], released under the GNU General Public License v3.0 [Fre07].

We also define the OPoly pseudocode syntax that is used to write programs that can be parsed by OPoly. Moreover, we describe how the resulting, parallel C code with OpenMP directives is generated.

The main goal of OPoly is to give an accessible, easy-to-use, and simple implementation of a polyhedral compiler that can be adopted to automatically rewrite perfectly nested loops with uniform dependencies among the statements in the loop body. These are perhaps the most restrictive assumptions for which a polyhedral compiler may be used, but nevertheless a very common case in high-performance computations.

We show in chapter 5 that OPoly can lead to substantial speedups in computational time, even in the case of shallow loops like the one in alg. 2.1, by applying the OPoly transformation to various implementations of the *Gauss-Seidel* algorithm.

[1]`https://github.com/GiackAloZ/OPoly`

4.1 Python

For implementing OPoly, we mainly used the Python [VD95] programming language. Python is a high-level, interpreted, general-purpose programming language. We opted for Python because of its easiness of use and increasing popularity across all fields of computer science. We used the most recent version of Python at the time of writing, which was Python 3.9.

Python also has a rich ecosystem of libraries that are useful for developing all kinds of applications. We now describe the non-standard libraries we used for implementing OPoly.

4.1.1 NumPy and SymPy

We used two python libraries to help us with algebra and symbolic math computations, namely NumPy [Har+20] and SymPy [Meu+17]. Both of them are part of SciPy [2], a Python-based ecosystem of open-source software for mathematics, science, and engineering.

NumPy is a Python library for dealing with multi-dimensional arrays. We used it for matrix computations, specifically for multiplications between matrices and inverse matrix calculations.

SymPy is a Python library for symbolic mathematics representations and computations. We used it to represent parametric polyhedrons as systems of inequalities with symbolic variables, in order to simplify the implementation of the Fourier-Motzkin algorithm (see section 2.5). SymPy can isolate variables and simplify symbolic expressions, as well as perform translations between these expressions and their equivalents in many common programming languages, among which the C programming language.

NumPy and SymPy are well integrated, so it is easy to use them in conjunction.

4.1.2 PyMzn

PyMzn [3] is a python library that wraps the MiniZinc tool for constraint programming modeling. We used PyMzn instead of the more standard MiniZinc Python package, because of its better interfaces and easiness of use.

PyMzn lets us call the `minizinc` command-line tool directly from Python: it feeds the model's parameters and returns the solution(s) found using standard

[2]For more information on SciPy, visit `https://www.scipy.org`

[3]For more information on PyMzn, visit its GitHub repository at `https://github.com/paolodragone/pymzn`

Python dictionaries.

4.1.3 Pytest

Pytest is one of the most used Python testing frameworks. We used Pytest to perform modules' unit tests and integration tests between each module. We also used the Pytest plugin `pytest-cov` that produces coverage reports of the tested code, so we could be aware of the percentage of the written code that was actually being tested [4].

[4]For more information on `pytest-cov`, visit `http://pytest-cov.rtfd.org/`

4.2 MiniZinc

MiniZinc [Net+07] is a free and open-source modeling language for constraint programming problems. MiniZinc can be used to model different constraint programming problems, specifically satisfaction or optimization problems, without worrying about the actual methods used for solving them. MiniZinc compiles the model into FlatZinc, a solver input language that is understood by a wide range of solvers, such as constraint programming (CP), mixed integer linear programming (MIP), or boolean satisfiability (SAT) solvers. A MiniZinc model does not dictate *how* to solve the problem, but rather specify the various constraints on the *decision variables* (i.e. the values we want to find) using a high-level syntax that is closely related to the mathematical way of representing constraint programming problems.

MiniZinc models are usually *parametric*, i.e. they describe a whole class of problems rather than an individual problem instance. In this way, we can feed the values of the parameters only when we need to solve an actual problem instance.

We use MiniZinc to model the problems of finding an optimal schedule (eq. (2.4.9)) and its matching allocation (eq. (2.4.10)). We use the Chuffed [5] solver for solving our models. Chuffed is based on *lazy clause generation* [Chu+11], which is a hybrid approach to constraint solving that combines features of finite domain propagation and boolean satisfiability. We experimentally chose Chuffed after trying many solvers, since it has given the best performance on solving our particular problems.

4.2.1 Minizinc model structure

A MiniZinc model is usually structured in four main parts:

- *parameters* declarations: constant values given as input to the model for solving a particular instance of it. Parameters must be declared with a type (e.g. `int`). They can be assigned directly in the model or by giving a separate file containing their value. Their value cannot change during the execution of the model, effectively representing the model's constants.

- *decision variables* declarations: these are the variables for which we want the problem to be solved for. Unlike the variables in a standard programming language, the modeler does not need to give them a value. Rather the value of a decision variable is unknown and it is only when the model is

[5] `https://github.com/chuffed/chuffed`

executed that the solver determines if the decision variable can be assigned a value that satisfies the constraints in the model and if so what this is.

Decision variables must be declared with a type or a *domain*, representing the set of possible values that the variable can take. The type of the variable is then inferred from the type of values in the given domain.

- *constraints* definitions: boolean expressions that the decision variables must satisfy to be valid solutions. These expressions can involve a combination of decision variables and parameters.

- *type of problem*: what kind of problem we want to solve, i.e. a satisfaction problem or an optimization problem. In the latter case, we need to also specify an optimization *objective*, which can be any kind of arithmetical expression involving decision variables and parameters.

MiniZinc gives us the syntax for defining constraints in a way that is similar to the mathematical formulation of the problem. We can use MiniZinc built-in functions to define the constraints on the decision variables (e.g. the `forall` constraint) or declare our custom functions and constraints using the appropriate syntax and use a mix of both.

We now explain the two models we wrote for solving the constraint programming problems of OPoly.

4.2.2 Schedule model

Recall the scheduling problem in the form of eq. (2.4.9). This is a basic integer linear programming problem that can be easily modeled with MiniZinc.

The schedule model's input parameters are:

- the number of nested loops $n \in \mathbb{N}$ (i.e. the polyhedron dimensions, or the number of loop index variables);

- the number of dependence vectors $r \in \mathbb{N}$;

- the dependence matrix $D \in \mathbb{Z}^{r \times n}$.

The decision variable is the schedule vector $\tau \in \mathbb{Z}^n$. We constrain the values of τ to be nonnegative integers (line 14) and each row of the dependence matrix multiplied by τ to be strictly positive (line 17). The optimization objective is to minimize the *latency* of the schedule (see eq. (2.4.8)), so we minimize the sum of all coefficients of τ, since we know that they can only be nonnegative (line 20). The complete schedule model is shown in listing 4.1.

```
% Parameters definitions
par int: n;                         % Number of indexes
par int: r;                         % Number of dependencies
set of int: N = 1..n;
set of int: R = 1..r;
array[R,N] of par int: D;           % Dependency matrix
% ---------------------
% Variables definitions
array[N] of var int: tau;           % Schedule vector
% ----------------------
% Constraints definitions
% Valid schedule constraint
constraint forall(j in R)(
    sum(i in N)(D[j,i] * tau[i]) > 0
);
% Nonnegative coefficients
constraint forall(i in N)(tau[i] >= 0);
% ----------------------
% Minimization objective (minimal latency schedule)
solve minimize sum(i in N)(tau[i]);
```

Listing 4.1: MiniZinc schedule model.

4.2.3 Allocation model

Recall the scheduling problem in the form of eq. (2.4.10). This is a general constraint programming optimization problem that can be modeled with MiniZinc by defining some custom functions that compute the determinant of a matrix.

MiniZinc supports recursive functions, so we define a function `determinant` that uses the matrix *Laplace expansion* formula to compute its determinant. In linear algebra, the Laplace expansion states that the determinant of an n-by-n matrix A can be computed with the following recursive formula:

$$\det(A) = \sum_{j=1}^{n} (-1)^{i+j}\, a_{i,j}\ \det(C_{i,j}),\ \forall i \in \{1, \ldots, n\}\,, \tag{4.2.1}$$

where $a_{i,j}$ is the element of the ith row and jth column of A, and $C_{i,j}$ is the submatrix of A formed by deleting the ith row and the jth column of A. The term $(-1)^{i+j}\ \det(C_{i,j})$ is also called the *cofactor* or *first minor* of the element $a_{i,j}$. Equation (4.2.1) is true for every row i of the matrix A we choose, so in particular it is true for the first row $i = 1$.

The implementation of the Laplace expansion in MiniZinc is show in listing 4.2. The `submatrix` function extracts the $C_{i,j}$ submatrix, the `cofactor` function computes the cofactor of the element $a_{i,j}$ and the `determinant` function calculates the determinant of a matrix A by using the Laplace expansion on its first row.

The allocation model's parameters are:

- the number of nested loops $n \in \mathbb{N}$;
- the schedule vector $\tau \in \mathbb{Z}^n$.

The decision variable is the complete transformation matrix $T \in \mathbb{Z}^{n \times n}$ made up by the schedule vector and the allocation matrix (see eq. (2.3.10)).

We can include the module `determinant.mnz` (listing 4.2) to the allocation model and use the function `determinant` to constrain the absolute value of the determinant of the transformation matrix to be equal to one, giving us a unimodular matrix (line 19). Other constraints are defined such that the first row of the transformation matrix is equal to the schedule vector (line 15) and the coefficients of the allocation matrix are nonnegative (line 17). The optimization objective is to minimize the values of the coefficients of the allocation matrix and their distance to the diagonal (line 22). The complete allocation model is shown in listing 4.3.

```
% Extracts the submatrix C_{i,j} from the matrix M
function array[int,int] of var int: submatrix(int: i, int: j,
   int: dim, array[int,int] of var int: M) =
    array2d(1..(dim-1), 1..(dim-1), [M[a, b] | a in (1..dim
       diff i..i), b in (1..dim diff j..j)]);

% Computes the cofactor of the element (i,j) of the matrix M
function var int: cofactor(int: i, int: j, int: dim, array[int,
   int] of var int: M) =
    pow(-1, i+j) * determinant(dim-1, submatrix(i, j, dim, M));

% Calculates the determinant of a (dim)x(dim) matrix M using
   Laplace expansion
function var int: determinant(int: dim, array[int,int] of var
   int: M) =
    if dim = 1 then M[1,1] else (
        sum(j in 1..dim)(M[1,j] * cofactor(1, j, dim, M))
    ) endif;
```

Listing 4.2: MiniZinc determinant function using the Laplace expansion.

```
% Libraries inclusions
include "determinant.mzn";
% ----------------------
% Parameters definitions
par int: n;                   % Number of indexes
set of int: N = 1..n;
set of int: A = 2..n;         % Allocation matrix set of indexes
array[N] of par int: tau;     % Schedule vector
% ----------------------
% Variables definitions
array[N,N] of var int: T;     % Transformation matrix
% ------------------------
% Constraints definitions
% First row schedule vector constraint
constraint forall(j in N)(T[1,j] = tau[j]);
% Nonnegative coefficients constraint
constraint forall(i in A, j in N)(T[i,j] >= 0);
% Unimodular transformation matrix constraint
constraint abs(determinant(n, T)) = 1;
% ------------------------
% Minimization objective (values and distance to the diagonal)
solve minimize sum(i in A, j in N)(T[i,j] * (1 + abs(i-j)));
```

Listing 4.3: MiniZinc allocation model.

4.3 Pseudocode

Our current implementation of OPoly does not parse C code directly. Instead, we invented a *pseudocode syntax* for representing loops in the form of listing 2.1. We chose this approach because our pseudocode syntax is much more restricted and easy to parse than real C code would be. However, future implementations of the Parser module (see section 3.2.1) can extend OPoly's capability of parsing different languages.

In this section, we present the main constructs of OPoly pseudocode and some examples of loop nests that have been rewritten in this syntax.

4.3.1 Definitions

We start by giving some definitions of the terms that we will later use to describe the pseudocode syntax and its current restrictions. First of all, we define two simple, self-contained symbols that can be used as building blocks for more complicated expressions:

- *constants*: numeric symbols that represent their value (e.g. 1, 2, 0.5, etc...). They can assume integer or decimal values;

- *variables*: symbols denoted by a name. As variables in a standard programming language, they represent some value that can change during the execution of the program.

  Variables can be *simple* if they store a single value, or *indexed* if they store multiple values that can be accessed by a list of indexes. Indexed variables are the equivalent of multi-dimensional arrays in a standard programming language.

We can combine constants and variables to form *expressions*. Each expression can be imagined as a list of constants or variables, where each pair of successive elements is connected with an *operator* that describes what operation is to be performed between the elements.

Two statements are supported by the parser at the time of writing:

- *assignments*: statements representing a change of a variable's value. Assignments are composed of two terms: the *left* term is a variable, and the *right* term is an expression. This means that the variable in the left term assumes a new value equal to the value of the expression in the right term.

- *for loops*: statements representing loops that iterate over a simple variable, called *index* variable, assuming a range of integer values.

The initial value of the index variable is called *lower* bound, while the last value is called *upper* bound. One can also specify the *step* for which the index variable is incremented (or decremented) at each iteration. This for loop formulation is very much similar to that of a Fortran "DO" loop.

There is also one more statement that OPoly pseudocode can represent, which is the *declaration* statement. It is very much similar to an assignment statement, but the right term is optional (*initialization*). This statement is **only** used in the generated code to retrieve the values of the original index variables.

In the next section, we describe the syntax of each one of the concepts we just introduced.

4.3.2 Syntax

In the syntax definitions, we use the following syntactic notations:

- bold text (**`boldface`**) when the text is to be used as-is;
- italic text (*`italic`*) when the text represents some replaceable arguments;
- square brackets (`[ ]`) surrounding optional arguments;
- ellipses (`...`) for repeatable arguments.
- pipes (|) for denoting a choice between two arguments.

Moreover, every time we mention a *name* (like variable names or function names), it is assumed to be composed of alphanumeric characters and underscores (_), where the first character must always be alphabetic.

Expressions

The syntax for generic expressions is:

```
expr [op other_expr]...
```

where *`expr`* and *`other_expr`* are constants, variables or general expressions, and `op` is an operator. Supported operators at the time of writing are: "+" addition, "-" subtraction, "*" multiplication, and "/" division.

Expressions can be preceded by an *unary* operator to indicate unary expressions, with the following syntax:

```
unary_op expr
```

where *unary_op* is an unary operator. At the time of writing, the only supported unary operator is "-", denoting a negative expression.

Expressions can be enclosed by a set of parenthesis to represent a *grouping* expression, with the following syntax:

```
(expr)
```

There can also be *function* expressions, representing named functions with a list of expressions as their arguments, with the following syntax:

```
func_name(arg1,...,argn)
```

where *func_name* is the name of the function and the arguments (*arg1*, ..., *argn*) are expressions.

Variables

The syntax for variables is:

```
var_name[[index]...]
```

where *var_name* is the name of the variable and *index* is an expression representing the index value. If a variable does not have indexes it is a simple variable. Otherwise, it is an indexed variable.

For example, the pseudocode `foo` indicates a simple variable named "foo", while `bar[i+1][2]` indicates an indexed variable named "bar" with two index expressions `i+1` and `2`.

Assignments

The syntax for assignments is:

```
STM left_term = right_term;
```

where *left_term* is a variable and *right_term* is an expression. Please note the **STM** and at the start and the **;** at the end of the statement.

Declarations

The syntax for declarations is:

```
VAR var_name [= initialization];
```

where *var_name* is the name of the declared variable and *initialization* is an expression. Please note the **VAR** at the start and the **;** at the end of the statement.

For loops

The for loop statement syntax is:

```
FOR [CONC] index FROM lb TO ub [STEP step] { loop body }
```

where `index` is a simple variable denoting the loop index, `lb` and `ub` are expressions denoting the lower and upper bounds of the loop, `step` is an expression denoting the step of the loop's index variable, and finally `loop body` are multiple other statements, potentially loop statements in the case of loop nests. If the step is omitted, a unitary step is assumed. The **CONC** keyword is used **only** in the generated pseudocode to express that the for loop can be executed concurrently.

Even though OPoly is capable of parsing all loop statements in this form, in order to parallelize them, the assumptions (A1)-(A6) (see section 2.1 and section 2.4.3) must be fulfilled. Furthermore, we currently do not support non-unitary steps or loop bounds that are not expressions in the form:

```
var|const [+|- const]
```

where `var` is a simple variable that can be an index of the previous loops or a variable which value never changes (*parameter*), and `const` is a constant value.

4.3.3 Examples

As examples, we present the algorithms alg. 2.1 and alg. 2.3 that have been rewritten in pseudocode syntax, respectively in listing 4.4 and listing 4.5.

We are not restricted to have only one assignment or only one variable in the loop body. We can write a loop nest like the one in listing 4.6 which can also be parallelized by OPoly.

```
FOR i FROM 1 TO N {
    FOR j FROM 1 TO N {
        STM Q[i][j] = (Q[i-1][j] + Q[i][j-1]) * 0.5;
    }
}
```

Listing 4.4: Algorithm 2.1 rewritten in OPoly pseudocode.

```
FOR i FROM 1 TO L {
    FOR j FROM 2 TO M {
        FOR k FROM 2 TO N {
            STM U[j][k] = (U[j+1][k] + U[j][k+1] + U[j-1][k] +
               U[j][k-1]) * 0.25;
        }
    }
}
```

Listing 4.5: Algorithm 2.3 rewritten in OPoly pseudocode.

```
FOR k FROM 1 TO q {
    FOR i FROM 1 TO n-2 {
        STM a[i] = b[i+1] + c[i];
        STM a[i-1] = 1.0 / a[i];
        STM b[i] = a[i];
    }
}
```

Listing 4.6: Multiple assignments loop nest in OPoly pseudocode.

4.4 C code generation

At the time of writing, OPoly can generate code in two different languages: OPoly pseudocode language, and C language with OpenMP directives for parallel execution.

The OpenMP API [DM98] is a specification for a set of compiler directives that can be used to specify shared-memory parallelism in Fortran and C/C++ programs. We chose OpenMP because of its wide use and support, and also for its relatively easy way to write parallel for loops.

We adopted the C99 syntax [ISO99] for generating the transformed loops, which allows us to declare index variables in the loop's initialization. Also, we used the OpenMP 4.0 API specifications [Ope13].

As an example, we show the code generated by OPoly from the serial implementation of alg. 2.1 rewritten in pseudocode in listing 4.4. The generated pseudocode is shown in listing 4.7, while the C code is show in listing 4.8.

The transformed index variables are renamed from "i" and "j" to "new_i" and "new_j". The new loop bounds replace the old ones: in the pseudocode version, the new bounds are simply placed in the for loop statement; meanwhile, in the C version, the new bounds are declared as separate variables just before the for loop statement, and then used within it as usual.

In the pseudocode version, the loops that can be executed in parallel are tagged with the keyword **`CONC`**, while in the C version, a `#pragma omp parallel for` OpenMP directive is used, but only for the first loop that can be computed in parallel (i.e. the second one). We cannot use the `collapse` OpenMP clause, because the loops are not rectangular, i.e. the loop bounds are not constant, but they depend on the values of the previous loops indexes. We could try to use multiple, nested OpenMP parallel for clauses, by exploiting the so-called *nested* parallelism feature of OpenMP. Unfortunately, due to the overhead of handling nested parallel regions, this approach does not lead to better performance.

Collapsing of non-rectangular for loops is a more recent advancement of the polyhedral compilation techniques, based on Ehrhart polynomials [Cla96; CAK17] for counting the number of integer points inside a polyhedron. This is one of the many enhancements that can be implemented in OPoly for future work.

```
FOR new_i FROM 2 TO 2 * N STEP 1 {
    FOR CONC new_j FROM fmax(1, -N + new_i) TO fmin(N, new_i -
       1) STEP 1 {
        VAR i = new_i - new_j;
        VAR j = new_j;
        STM Q[i][j] = (Q[i - 1][j] + Q[i][j - 1]) * 0.5;
    }
}
```

Listing 4.7: Pseudocode generated by OPoly of alg. 2.1 from the starting implementation in listing 4.4.

```
for(int new_i = 2; new_i <= 2 * N; new_i++) {
    int new_j_lb = fmax(1, -N + new_i);
    int new_j_ub = fmin(N, new_i - 1);
    #pragma omp parallel for
    for(int new_j = new_j_lb; new_j <= new_j_ub; new_j++) {
        int i = new_i - new_j;
        int j = new_j;
        Q[i][j] = (Q[i - 1][j] + Q[i][j - 1]) * 0.5;
    }
}
```

Listing 4.8: C code generated by OPoly of alg. 2.1 from the starting implementation in listing 4.4.

Chapter 5

Benchmarks

In this chapter, we show how OPoly generates parallelizable code from a serial implementation of an algorithm that is used in practice. We discuss the performance of the generated code, by comparing it with the original, serial implementation. We measure execution times of both and analyze the speedup and strong scaling efficiencies of the generated code.

We considered the problem of generating a parallelized implementation of the *Gauss-Seidel* algorithm, which is a classical numerical method for solving systems of linear equations but can also be used as a smoother component in multigrid methods [HY73; Bar+94]. The Gauss-Seidel algorithm can be generalized to be implemented in n dimensions, so we compare the implementations for one, two, and three dimensions. The three algorithms are show respectively in alg. 5.1, alg. 5.2 and alg. 5.3. The indexes of the array variable ϕ are assumed to be zero-based.

Algorithm 5.1: 1D Gauss-Seidel algorithm

```
for $q$ from 1 to $niter$ do
    for $i$ from 1 to $M - 2$ do
        $\phi(i) \leftarrow \frac{1}{2}(\phi(i-1) + \phi(i+1))$
    end
end
```

We took the implementation in three dimensions from [HW10, p.159], in which the Gauss-Seidel algorithm is parallelized with a technique called *wavefront* parallelization [MA96].

As mentioned in [HW10, p.301], wavefront parallelization is to be preferred for this particular problem if we use cache-based processors for the computation, because they suffer from the erratic access patterns generated by the hyperplane

Algorithm 5.2: 2D Gauss-Seidel algorithm

```
for $q$ from 1 to $niter$ do
    for $j$ from 1 to $M - 2$ do
        for $i$ from 1 to $N - 2$ do
            $\phi(i, j) \leftarrow \frac{1}{4}(\phi(i-1, j) + \phi(i+1, j) + \phi(i, j-1) + \phi(i, j+1))$
        end
    end
end
```

Algorithm 5.3: 3D Gauss-Seidel algorithm

```
for $q$ from 1 to $niter$ do
    for $k$ from 1 to $L - 2$ do
        for $j$ from 1 to $M - 2$ do
            for $i$ from 1 to $N - 2$ do
                $\phi(i, j, k) \leftarrow \frac{1}{6}(\phi(i-1, j, k) + \phi(i+1, j, k) + \phi(i, j-1, k) +$
                  $\phi(i, j+1, k) + \phi(i, j, k-1) + \phi(i, j, k+1))$
            end
        end
    end
end
```

```
FOR q FROM 1 TO niter {
    FOR j FROM 1 TO M-2 {
        FOR i FROM 1 TO N-2 {
            STM phi[i][j] = ( phi[i-1][j] + phi[i+1][j]
                            + phi[i][j-1] + phi[i][j+1] ) * (1
                               / 4.0);
        }
    }
}
```

Listing 5.1: 2D Gauss-Seidel algorithm (alg. 5.2) rewritten in OPoly pseudocode.

method, and thus failing to exploit spatial locality. However, the hyperplane method allows for more concurrent execution than wavefront parallelization does.

Firstly, we need to rewrite the algorithms in OPoly pseudocode syntax. An example of such rewriting for the 2D version is shown in listing 5.1. Note that the 2D version is very much similar to the alg. 2.3 used as example in section 2.4.

Then, we use OPoly to generate the parallel version in C syntax with an `#pragma omp parallel for` OMP directive on the second loop, resulting in the code shown in listing 5.2.

We successively wrote the remaining parts of the programs that read the input vector `phi`, execute the generated for loop, and outputs the resulting vector. The programs start a timer before the algorithm's for loop and stop it after exiting it, to measure only the loop's execution time. The outputs of the original serial code are compared with their parallelized counterpart, to verify the correctness of the computation.

We also generated the input files containing the initial values of the elements of `phi`, selecting them from a uniform distribution at random. We chose the input size to be roughly the same for every version of the algorithm, by varying the algorithm parameters: the number of iterations `niter` and the size of the dimensions N, M and L of the array variable `phi`.

For running the benchmarks, we used a machine with two Intel© Xeon© E5-2603 v4 processors with 6 cores each, for a total of 12 cores and 64 GB of RAM.

Parameters for the different versions and execution times (in seconds) with different numbers of processors used (p) for the parallel versions are shown in table 5.1.

```c
for(int new_q = 4; new_q <= M + N + 2*niter - 4; new_q++) {
    int new_j_lb = ceil(fmax(1, (1.0 / 2.0) * (-M - N + new_q +
        4)));
    int new_j_ub = floor(fmin(niter, (1.0 / 2.0) * (new_q - 2))
       );
    #pragma omp parallel for
    for(int new_j = new_j_lb; new_j <= new_j_ub; new_j++) {
        int new_i_lb = fmax(1, -M - 2*new_j + new_q + 2);
        int new_i_ub = fmin(N-2, -2*new_j + new_q - 1);
        for(int new_i = new_i_lb; new_i <= new_i_ub; new_i++) {
            int q = new_j;
            int j = -new_i - 2*new_j + new_q;
            int i = new_i;
            phi[i][j] = (phi[i-1][j] + phi[i+1][j] + phi[i][j
               -1] + phi[i][j+1]) * (1 / 4.0);
        }
    }
}
```

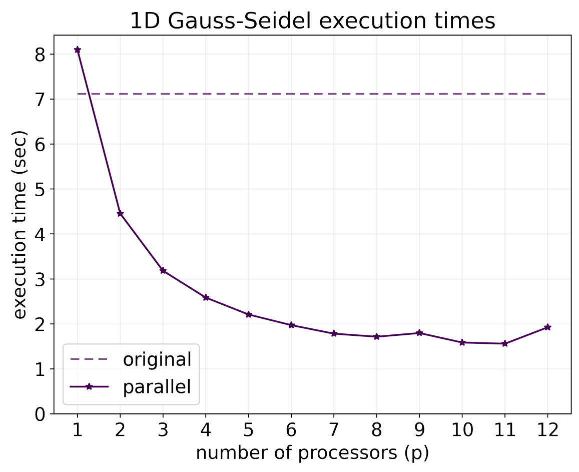
Listing 5.2: C code generated by OPoly from the 2D Gauss-Seidel implementation in pseudocode syntax (listing 5.1).

versions			execution times					
dims	params		serial	parallel				
	niter	$N = M = L$		$p = 1$	$p = 4$	$p = 8$	$p = 10$	$p = 12$
1D	10k	100k	7.113	8.095	2.585	1.715	1.585	1.924
2D	1k	1k	13.50	18.65	7.930	4.436	3.689	3.680
3D	100	200	31.46	40.46	15.26	8.586	7.345	7.398

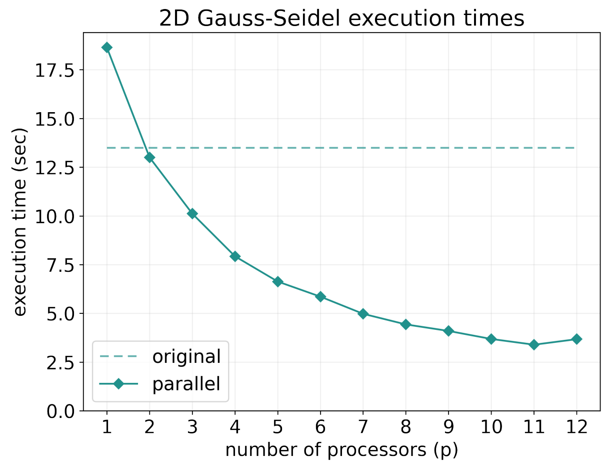
Table 5.1: Execution times of the serial and parallel versions of Gauss-Seidel algorithms in various dimensions. Times are reported in seconds. The number of iterations and the array sizes are reported for each version. Execution times for the parallel versions are reported with regard to p: the number of processors used in the computation.

Execution times plots of the various versions, with regard to the number of processors used, are shown in fig. 5.1. Execution times of the original serial implementations are shown with a dotted horizontal line.

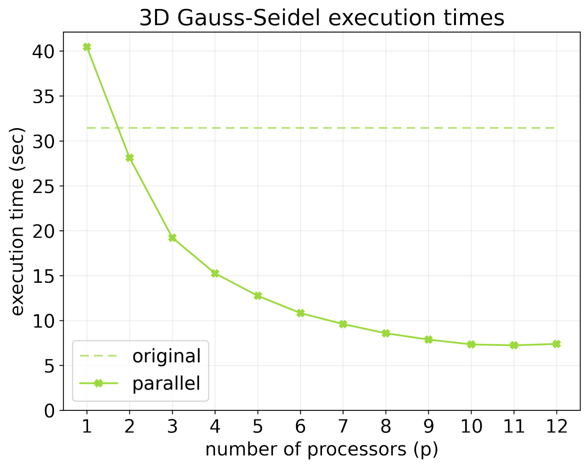
As expected, the parallel versions with $p = 1$ are a bit slower than the original serial implementation, since there are extra computations to be performed (loop bounds) and spatial locality is much worse.



(a) Execution times of the 1D Gauss-Seidel algorithm.

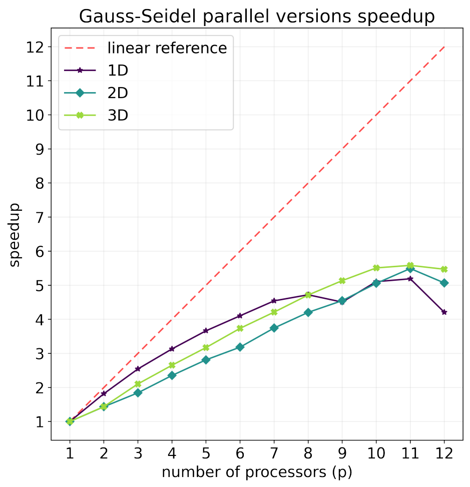


(b) Execution times of the 2D Gauss-Seidel algorithm.

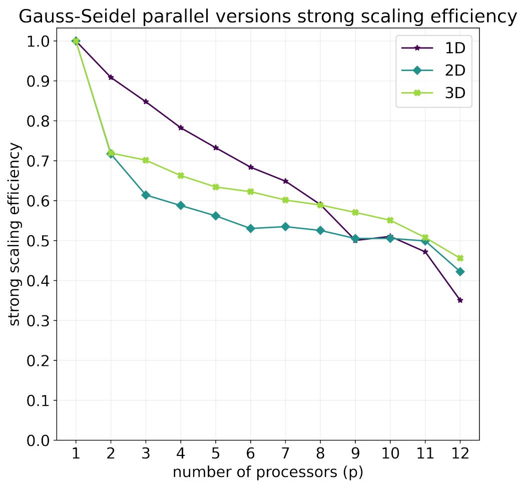


(c) Execution times of the 3D Gauss-Seidel algorithm.

Figure 5.1: Execution times of the Gauss-Seidel implementations in various dimensions. The original, serial version is show as reference with a dashed line.



(a) Speedup plot.



(b) Strong scaling efficiency plot.

Figure 5.2: Speedup and strong scaling efficiency plots of the parallel Gauss-Seidel versions in different dimensions.

As the number of processors grows, the execution time gets lower, until p approaches the number of available cores.

The *speedup* plot of the parallel versions is shown in fig. 5.2a. The speedup $S(p)$ of the parallel version with p processors over the same one with one processor (in the case that the workload is equal between the two) is defined as the execution time $T_{\text{parallel}}(1)$ with one processor divided by the execution time $T_{\text{parallel}}(p)$ with p processors:

$$S(p) = \frac{T_{\text{parallel}}(1)}{T_{\text{parallel}}(p)}. \tag{5.0.1}$$

Note that we use the execution time $T_{\text{parallel}}(1)$ of the parallel version with only one processor as reference for computing $S(p)$.

The best case is that $S(p) = p$, also known as *linear speedup*. We show the linear speedup case as a reference in fig. 5.2a. Usually, $S(p)$ is more close to p with lower values of p, since the overhead costs for handling multiple processes are lower.

Also, the speedup is constrained by the fraction α of the total execution time spent on the serial portion of the program. By Amdahl's law [Amd67] we have that:

$$S(p) = \frac{1}{\alpha + \frac{1-\alpha}{p}}. \tag{5.0.2}$$

In our case, there is a substantial part of the program which is executed in serial, specifically the outer loop. For this reason, we observe that the speedup of our generated implementations is sub-linear and tends to grow less and less for bigger values of p. For low values of p, we observe that the speedup is almost linear.

We can measure the speedup growth by computing the *strong scaling efficiency*. The strong scaling efficiency $E(p)$ is defined as the speedup $S(p)$ divided by the number of processors p:

$$E(p) = \frac{S(p)}{p} = \frac{T_{\text{parallel}}(1)}{pT_{\text{parallel}}(p)}. \tag{5.0.3}$$

In the best case (linear speedup), $E(p) = 1$. The strong scaling efficiency plot is shown in fig. 5.2b.

We observe that the speedup and strong scaling efficiency of the 1D version are better than the other ones for lower values of p. On the contrary, the 2D and 3D versions begin to scale better than the 1D version as the number of processors p increases. This is because the computations are better distributed among the processors as p increases in the 2D and 3D versions.

All things considered, we achieve a good improvement in execution time with regard to the original serial implementations of the algorithms, even with a small number of processors.

Chapter 6

Conclusions

This thesis aimed at exploring and understanding the basic concepts of the polytope model and polyhedral compilation, while giving the tools for implementing a simple polyhedral compiler that can perform automatic code generation.

We used this knowledge to implement OPoly and to show that polyhedral compilation can be successfully used to speed up the computation for perfectly nested for loops that have uniform dependencies.

Although many libraries for handling polyhedra and much more complex polyhedral compilers exist, OPoly gives us a simple, but effective way to exploit polyhedral compilation for automatic parallel code generation.

While we have successfully shown the increased performance of the generated code, there are a lot of enhancements that are possible to achieve with the most recent techniques in polyhedral compilation, such as the analysis of spatial locality for better use of the cache, or specific code generation for massively-parallel architectures like GPUs. Also, the assumptions on the shape of the loops and the dependencies in the loop body can be relaxed by generalizing the theory for affine dependencies and transformations.

Future work aims on diving deeper into the recent techniques for polyhedral compilation and implementing them in OPoly, in order to achieve better performances and include more classes of problems to be optimized and parallelized automatically with it. In the future, OPoly can be extended to directly parse complete programs written in various languages, and to also generate code for different languages and parallel programming models.

Another interesting future project is to give a visual representation of the original and transformed loops' index spaces in at most three dimensions. That would be most useful for explaining visually the geometric representation of programs for didactic purposes.

Bibliography

[Amd67] Gene M. Amdahl. “Validity of the Single Processor Approach to Achieving Large Scale Computing Capabilities”. In: *Proceedings of the April 18-20, 1967, Spring Joint Computer Conference*. AFIPS ’67 (Spring). Atlantic City, New Jersey: Association for Computing Machinery, 1967, pp. 483–485. ISBN: 9781450378956. DOI: 10.1145/1465482.1465560 (cited on page 62).

[KMW67] Richard M. Karp, Raymond E. Miller, and Shmuel Winograd. “The Organization of Computations for Uniform Recurrence Equations”. In: *J. ACM* 14.3 (1967), pp. 563–590. DOI: 10.1145/321406.321418 (cited on pages 5, 16).

[Mor69] L.J. Mordell. *Diophantine Equations*. 1st. Vol. 30. Pure and Applied Mathematics. Academic P.,U.S., 1969. ISBN: 978-0-125-06250-3 (cited on page 22).

[HY73] L. Hageman and David Young. “Iterative Solution of Large Linear Systems”. In: *The American Mathematical Monthly* 80 (Oct. 1973), p. 92. DOI: 10.2307/2319285 (cited on page 57).

[Lam74] Leslie Lamport. “The Parallel Execution of DO Loops”. In: *Commun. ACM* 17.2 (1974), pp. 83–93. DOI: 10.1145/360827.360844 (cited on pages 1, 2, 5, 6, 16, 20).

[RK88] S. K. Rao and T. Kailath. “Regular iterative algorithms and their implementation on processor arrays”. In: *Proceedings of the IEEE* 76.3 (1988), pp. 259–269. DOI: 10.1109/5.4402 (cited on page 5).

[Ban93] Utpal Banerjee. “Unimodular Matrices”. In: *Loop Transformations for Restructuring Compilers: The Foundations*. Boston, MA: Springer US, 1993, pp. 21–48. ISBN: 978-0-585-28004-2. DOI: 10.1007/978-0-585-28004-2_2 (cited on page 12).

[Len93] Christian Lengauer. "Loop Parallelization in the Polytope Model". In: *CONCUR '93, 4th International Conference on Concurrency Theory, Hildesheim, Germany, August 23-26, 1993, Proceedings*. Ed. by Eike Best. Vol. 715. Lecture Notes in Computer Science. Springer, 1993, pp. 398–416. DOI: `10.1007/3-540-57208-2\_28` (cited on pages 5, 22).

[Bar+94] Richard F. Barrett et al. *Templates for the Solution of Linear Systems: Building Blocks for Iterative Methods*. Other Titles in Applied Mathematics. SIAM, 1994. ISBN: 978-0-89871-328-2. DOI: `10.1137/1.9781611971538` (cited on page 57).

[VD95] Guido Van Rossum and Fred L Drake Jr. *Python tutorial*. Centrum voor Wiskunde en Informatica Amsterdam, The Netherlands, 1995 (cited on page 42).

[Cla96] Philippe Clauss. "Counting Solutions to Linear and Nonlinear Constraints Through Ehrhart Polynomials: Applications to Analyze and Transform Scientific Programs". In: *Proceedings of the 10th international conference on Supercomputing, ICS 1996, Philadelphia, PA, USA, May 25-28, 1996*. Ed. by Pen-Chung Yew. ACM, 1996, pp. 278–285. DOI: `10.1145/237578.237617` (cited on page 55).

[Fea96] Paul Feautrier. "Automatic Parallelization in the Polytope Model". In: *The Data Parallel Programming Model: Foundations, HPF Realization, and Scientific Applications*. Ed. by Guy-René Perrin and Alain Darte. Vol. 1132. Lecture Notes in Computer Science. Springer, 1996, pp. 79–103. DOI: `10.1007/3-540-61736-1\_44` (cited on pages 5, 22).

[MA96] Naraig Manjikian and Tarek S. Abdelrahman. "Scheduling of Wavefront Parallelism on Scalable Shared-memory Multiprocessors". In: *Proceedings of the 1996 International Conference on Parallel Processing, ICCP 1996, Bloomingdale, IL, USA, August 12-16, 1996. Volume 3: Software*. Ed. by Keshav Pingali. IEEE Computer Society, 1996, pp. 122–131. DOI: `10.1109/ICPP.1996.538567` (cited on page 57).

[DM98] Leonardo Dagum and Ramesh Menon. "OpenMP: an industry standard API for shared-memory programming". In: *Computational Science & Engineering, IEEE* 5.1 (1998), pp. 46–55 (cited on page 55).

[ISO99] ISO. *ISO C Standard 1999*. Tech. rep. ISO/IEC 9899:1999 draft. 1999. URL: `http://www.open-std.org/jtc1/sc22/wg14/www/docs/n1124.pdf` (cited on page 55).

[Loe99] Vincent Loechner. *PolyLib: A library for manipulating parameterized polyhedra*. 1999. URL: `https://repo.or.cz/polylib.git/blob_plain/HEAD:/doc/parampoly-doc.ps.gz` (cited on page 5).

[WIL00] DORAN K. WILDE. "A LIBRARY FOR DOING POLYHEDRAL OPERATIONS". In: *Parallel Algorithms and Applications* 15.3-4 (Dec. 2000), pp. 137–166. DOI: `10.1080/01495730008947354` (cited on page 5).

[BRS07] Uday Bondhugula, J. Ramanujam, and P. Sadayappan. *PLuTo: A Practical and Fully Automatic Polyhedral Parallelizer and Locality Optimizer*. Tech. rep. OSU-CISRC-10/07-TR70. The Ohio State University, Oct. 2007 (cited on page 5).

[Fre07] Free Software Foundation. *GNU General Public License Version 3 (GPL-3.0)*. Accessed 4 March 2021. June 2007 (cited on page 41).

[Net+07] Nicholas Nethercote et al. "MiniZinc: Towards a Standard CP Modelling Language". In: *Principles and Practice of Constraint Programming - CP 2007, 13th International Conference, CP 2007, Providence, RI, USA, September 23-27, 2007, Proceedings*. Ed. by Christian Bessiere. Vol. 4741. Lecture Notes in Computer Science. Springer, 2007, pp. 529–543. DOI: `10.1007/978-3-540-74970-7\_38` (cited on pages 3, 44).

[HW10] Georg Hager and Gerhard Wellein. *Introduction to High Performance Computing for Scientists and Engineers*. 1st. USA: CRC Press, Inc., 2010. ISBN: 143981192X (cited on page 57).

[Chu+11] Geoffrey Chu et al. "Symmetries and Lazy Clause Generation". In: *IJCAI 2011, Proceedings of the 22nd International Joint Conference on Artificial Intelligence, Barcelona, Catalonia, Spain, July 16-22, 2011*. Ed. by Toby Walsh. IJCAI/AAAI, 2011, pp. 516–521. DOI: `10.5591/978-1-57735-516-8/IJCAI11-094` (cited on page 44).

[GGL12] Tobias Grosser, Armin Größlinger, and Christian Lengauer. "Polly - Performing Polyhedral Optimizations on a Low-Level Intermediate Representation". In: *Parallel Process. Lett.* 22.4 (2012). DOI: `10.1142/S0129626412500107` (cited on page 5).

[Ope13] OpenMP Architecture Review Board. *OpenMP Application Program Interface Version 4.0*. July 2013. URL: `https://www.openmp.org/wp-content/uploads/OpenMP4.0.0.pdf` (cited on page 55).

[CAK17] Philippe Clauss, Ervin Altintas, and Matthieu Kuhn. "Automatic Collapsing of Non-Rectangular Loops". In: *2017 IEEE International Parallel and Distributed Processing Symposium, IPDPS 2017, Orlando, FL, USA, May 29 - June 2, 2017*. IEEE Computer Society, 2017, pp. 778–787. DOI: `10.1109/IPDPS.2017.34` (cited on page 55).

[McK17] Paul E. McKenney. "Is Parallel Programming Hard, And, If So, What Can You Do About It? (v2017.01.02a)". In: *CoRR* abs/1701.00854 (2017). arXiv: `1701.00854`. URL: `http://arxiv.org/abs/1701.00854` (cited on page 1).

[Meu+17] Aaron Meurer et al. "SymPy: symbolic computing in Python". In: *PeerJ Computer Science* 3 (Jan. 2017), e103. ISSN: 2376-5992. DOI: `10.7717/peerj-cs.103` (cited on page 42).

[Bag+19] Riyadh Baghdadi et al. "Tiramisu: A Polyhedral Compiler for Expressing Fast and Portable Code". In: *Proceedings of the 2019 IEEE/ACM International Symposium on Code Generation and Optimization*. CGO 2019. Washington, DC, USA: IEEE Press, 2019, pp. 193–205. ISBN: 978-1-7281-1436-1. URL: `https://arxiv.org/pdf/1804.10694.pdf` (cited on page 5).

[Har+20] Charles R. Harris et al. "Array programming with NumPy". In: *Nature* 585.7825 (Sept. 2020), pp. 357–362. DOI: `10.1038/s41586-020-2649-2` (cited on page 42).