Faster MPSoC Task Mapping via Symmetry Reduction

Seminar Paper

Timo Nicolai

Advisors: Prof. Jeronimo Castrillon*, Andrés Goens*

*Chair for Compiler Construction, cfaed, TU Dresden

05.06.20

Contents

Abstract

An important problem pertaining to MPSoCs is that of intelligently mapping (i.e. assigning) computational tasks to processing elements, often at runtime.

Whether some mapping is comparatively better than another usually depends on application specific optimality criteria that can only be evaluated by use of computationally expensive simulation. Thus there is a need to reduce the potentially very large mapping search space.

This thesis concerns itself with what we refer to as *symmetry reduction*, a technique that achieves this search space reduction by exploiting symmetries inherent in many MPSoC architecutures and which can be used in conjunction with existing methods that heuristically traverse the search space.

For this purpose we make use of well-established, as well as some recent state of the art methods originated by researchers in the field of *computational group theory (CGT)*, a branch of computational mathematics that concerns itself with the representation and analysis of many of the algebraic structures underlying group theory. We evaluate symmetry reduction results achieved by use of `mpsym` [13], a framework written from the ground up specifically to address symmetry reduction. We demonstrate that `mpsym` is fast, especially for architectures that are hierarchical in nature, and that it outperforms comparable general purpose CGT frameworks like GAP [8].

Chapter 1

Introduction

1.1 Structure

We begin by informally describing the problem this thesis addresses in Section 1.2 and elaborate on what we mean by symmetry reduction in Section 1.3. Chapter 2 then introduces some necessary fundamentals of computational group theory in an accessible manner, focussing on the representation of permutation groups and partial permutation inverse monoids, algebraic structures we will use to capture the symmetries inherent in MPSoC architectures. In Chapter 3 we separately discuss an especially important topic touched on in Chapter 2 in greater detail: construction of a so called *base and strong generating set* for a given permutation group. In Chapter 4 we then outline how to describe common MPSoC architectures mathematically and how to algorithmically transform these descriptions into the symmetry capturing algebraic structures introduced in Chapter 2. Equipped with these fundamentals, we then revisit symmetry reduction in Chapter 5 were we discuss it in a more formal manner and present several algorithms to address it. Finally, in Chapter 6 we present some experimental results obtained by use of `mpsym`.

1.2 Problem Statement

An MPSoC is a circuit that integrates several electronic components required to form a full-fledged computing system, i.e. multiple, often heterogeneous, processing elements as well as memory and I/O components. MPSoCs are often found in embedded applications in domains like multimedia or telecommunication where their specific makeup can meet specialized demands like high throughput, low energy consumption or adaptability thereof.

An important problem pertaining to MPSoCs is that of intelligently mapping (i.e. assigning) computational tasks to processing elements, often at runtime. These computational tasks can be both independent or interoperating programs with potentially very different hardware requirements.

To clarify, consider the abstract MPSoC architecture presented in Figure 1.1, a cluster of processing elements, connected in a regular mesh fashion by a number of communication channels. A processing element could for example represent a single core of some CPU while a communication channel might be a wired or wireless connection or shared memory. For now we assume that all processing elements and communication channels are effectively identical.

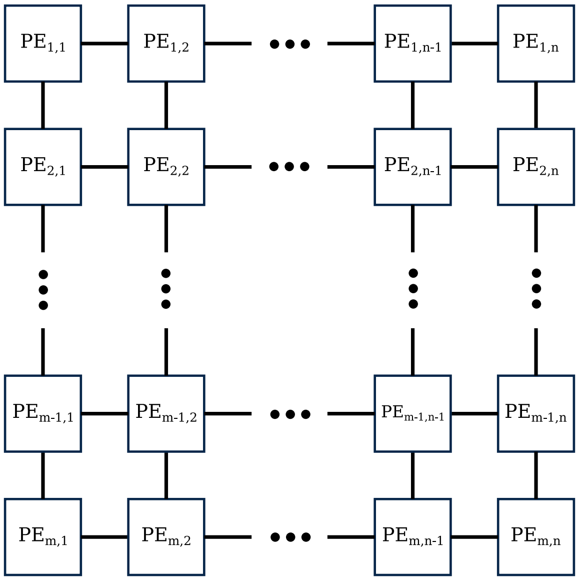


Figure 1.1: $m \times n$ regular mesh abstract MPSoC architecture.

Imagine that we have k distinct computational tasks that we would like to run on this architecture concurrently[1]. For simplicity we will assume that each task can run on any processing element and every processing element can run at most one task.

Which way of mapping these k tasks to the $m \cdot n$ processing elements is "the best" depends on underlying optimality criteria which can vary from application to application. These could e.g. be highest possible throughput, lowest energy consumption, a trade-off between the two or some even more complicated application-specific criteria. Unfortunately, for any such criteria we cannot work with an abstract MPSoC architecture alone because we need to consider "real-world" processing element and communication channel characteristics such as instructions per second, power consumption, latency, bandwidth etc.

In general, finding the optimal task mapping under these conditions requires evaluating our specific criteria for *every possible* task mapping. This evaluation step usually requires time and hardware expensive computer simulation and for our example we would have to perform it $(m \cdot n)!/((m \cdot n) - k)!$ times which quickly becomes unfeasible as m/n and k grow.

Thus, it is of great importance to find a general way to reduce the size of the task mapping "search space". One possible approach to this problem is to traverse the task mapping search space systematically, e.g. by generating "promising" new task mappings based on the performance of previously evaluated ones via a genetic algorithm.

[1] I.e. not necessarily in parallel.

1.3 Symmetry Reduction

The aim of this thesis is to describe and evaluate a very different method of addressing large task mapping search spaces which "collapses" the task mapping search space via *symmetry reduction*, a technique based on the observation that some task mappings are necessarily equivalent *by symmetry* and thus do not need to be simulated separately. This idea is based on previous work presented in [7]. A set of "equivalent" task mappings in this context is a set of task mappings that will necessarily exhibit the same runtime behaviour (apart from minor differences due to e.g. non-identical behaviour of equivalent hardware components elements).

To clarify what we mean when we talk about equivalence by symmetry, consider again the task mapping problem for $k = 4$ for the abstract MPSoC architecture introduced in Figure 1.1, this time for the specific case $m = n = 4$. Figure 1.2 shows four distinct possible mappings of the four tasks to the given abstract MPSoC architecture.

On close inspection we find that task mappings (a) and (b) are not equivalent because tasks two and three might not be interchangeable and thus the "communication structure" among the four tasks in (a) and (b) is different. On the other hand, task mappings (a) and (c) *are* equivalent since this communication structure is preserved.

A more tricky question is whether task mappings (a) and (c) are equivalent to task mapping (d). Technically this is not the case because for task mapping (d) some additional communication paths exist between the tasks. For instance, task one and two can communicate over two paths made up of three consecutive communication channels as opposed to just one, as indicated in red[2]. However, in practice it would most likely make sense to treat task mapping (a), (b) and (d) as equivalent. We say that that task mappings (a) and (b) are equivalent *under symmetry* and both of them are equivalent to task mapping (d) *under partial symmetry*. We will formalize these concepts later.

Symmetry reduction makes use of these task mapping equivalencies by partitioning the task mapping search space into sets of tasks that are equivalent under (partial) symmetry. From then on only one "representative" of each of these sets needs to be considered for simulation. Because the problem of finding these representatives is so central to the symmetry reduction approach, we introduce the term *task mapping orbit representative problem* (short *TMOR problem*) to describe it, the meaning behind which will become apparent in Chapter 5.

The symmetry reduction approach was first proposed in [7]. It can be easily combined with heuristic methods for traversing the task mapping space. As we shall see in Chapter 6, the overhead introduced by symmetry reduction is usually small.

[2]Whether both these communication channels are actually valid of course depends on the concrete routing algorithm employed but we will not consider this here.

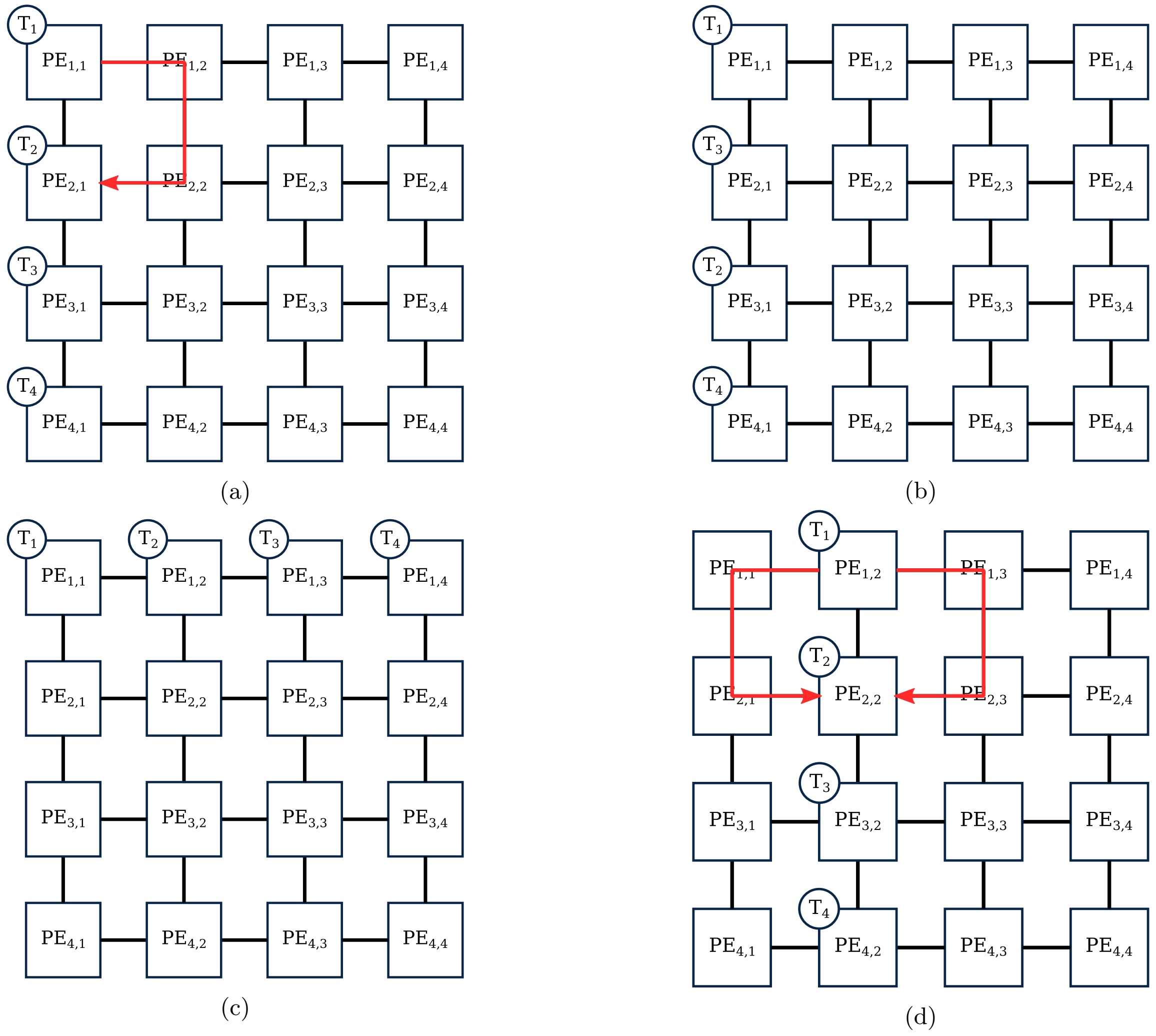


Figure 1.2: Four possible mappings of four tasks to a 4×4 regular mesh MPSoC architecture.

Chapter 2

Theoretical and Computational Foundations

In the following sections we will introduce some mathematical notation and definitions and present important theorems and algorithms from the field of computational group theory. In each of these sections we will first introduce some mathematical object via a series of definitions and then discuss algorithms and data structures with which we can construct and represent instances of this object in a computer program. Hereby we lay the foundation we need to formalize the TMOR problem in Section 5.1 and to understand the algorithms presented in Section 5.2 that address it.

Sections 2.1 and 2.2 will introduce permutations and permutation groups with which we can represent symmetries inherent in MPSoC architectures. Sections 2.3 and 2.4 will introduce partial permutations and partial permutation inverse monoids with which we can represent partial symmetries as well, making them potentially more powerful.

For general literature on group theory including permutations and permutation groups, refer to [15]. For an extensive treatment of inverse monoids refer to [11] and for a comprehensive overview of computational group theory refer to [10], [2] and [17].

2.1 Permutations

2.1.1 Motivation

Permutations are a useful mathematical tool when it comes to describing symmetries of systems. To illustrate this point, let us consider *isomorphisms* and *automorphisms* of undirected[1] graphs and how we can describe them mathematically.

[1]While we only consider undirected graphs here for simplicity, the following concepts can easily be extended to directed graphs as well.

Definition 2.1: Undirected Graph

An undirected graph is a tuple $G = (V, E)$ where $V = \{1, 2, \dots, n \mid n \in \mathbb{N}\}$ is a set of vertex indices and $E \subseteq \{\{x, y\} \mid i, j \in V\}$ with $\{i, j\} \in E \Leftrightarrow$ a graph edge exists between the ith and jth vertex.

Example 2.1: Undirected Graph

Consider the undirected graph $G = (V, E)$, with $V = \{1, 2, 3, 4\}$ and $E = \{\{1, 2\}, \{2, 3\}, \{3, 4\}, \{4, 1\}\}$ visualized in Figure 2.1.

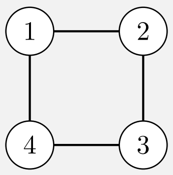


Figure 2.1: An undirected graph with four vertices and edges.

Definition 2.2: Graph Isomorphism

Given two undirected graphs $G = (V_G, E_G)$ and $H = (V_H, E_H)$, an isomorphism is a bijective function $\text{isom} : V_G \to V_H$ with $\{i, j\} \in E_G \Leftrightarrow \{\text{isom}(i), \text{isom}(j)\} \in E_H$. If such a function exists we also say that G and H are isomorphic.

Example 2.2: Graph Isomorphism

Consider the two undirected graphs $G = (V_G, E_G)$ and $H = (V_H, E_H)$ visualized in Figure 2.2.

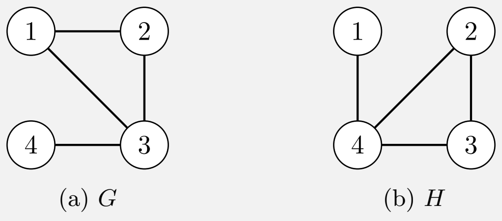


Figure 2.2: Two isomorphic graphs.

The function $\text{isom} : V_G \to V_H$ with:

$$\text{isom}(1) = 2,\ \text{isom}(2) = 3,\ \text{isom}(3) = 4,\ \text{isom}(4) = 1$$

is an isomorphism from G to H. We can easily see that this is true by verifying that $\{\{\text{isom}(i), \text{isom}(j)\} \mid \{i,j\} \in E_G\} = E_H$:

$$E_G = \{\{1,2\},\{1,3\},\{2,3\},\{3,4\}\}$$
$$\Downarrow$$
$$E_H = \{\{2,3\},\{2,4\},\{3,4\},\{4,1\}\}$$

Note that isomorphisms are *structure preserving*, i.e. G and H are the same graph aside from the differently numbered vertices.

Definition 2.3: Graph Automorphism

Given an undirected graph $G = (V, E)$, an automorphism of G autom : $V \to V$ is an isomorphism from G to itself.

Example 2.3: Graph Automorphism

Consider again the undirected graph G from Example 2.1 and the function autom : $V \to V$ with:

$$\text{autom}(1) = 2,\ \text{autom}(2) = 3,\ \text{autom}(3) = 4,\ \text{autom}(4) = 1$$

Renumbering the vertices of G according to autom results in the graph G' visualized in Figure 2.3b.

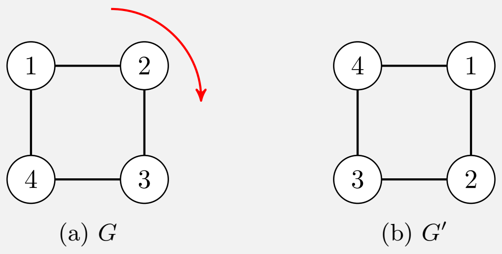


Figure 2.3: A graph automorphism.

Notice that this graph and G are not only isomorphic but *identical*, making autom an automorphism of G. Geometrically, if we interpret the vertices of G as the edges of a square, we can view autom as a 90° right rotation of that square about its center (as indicated by the red arrow in Figure 2.3a).

Graph automorphisms can thus express symmetries, e.g. rotational symmetry in Example 2.3, inherent in a graph. As per Definition 2.3, every automorphism is a bijective function from a set to itself. Such functions are exactly the permutations that this section is concerned with.

2.1.2 Definition

Definition 2.4: Permutation

A permutation is a bijection from a set Ω to itself.

We say that a permutation g stabilizes some $x \in \Omega$ if $g(x) = x$ and denote the *identity permutation* that stabilizes all $x \in \Omega$ by 1.

We usually denote permutations with the letters g and h and notate them in either *two-line notation* or *cycle notation*. The former displays x and $g(x)$ on top of each other for each $x \in \Omega$ in a $2 \times |\Omega|$ matrix. The latter denotes a permutation as a sequence of disjoint cycles where each element in a cycle is mapped to the next one by g (in a cyclic fashion) and cycles of length one are omitted. We denote the identity permutation as () in cycle notation.

In the context of this thesis we are mostly concerned with the case $\Omega \subseteq \mathbb{N}_+$, or more specifically $\Omega = \Omega_n = \{1, 2, \ldots, n\}$ where we refer to n as the *degree* of permutations of Ω_n. Unless otherwise noted, it is assumed that Ω has this form and we also write $\deg(g)$ instead of n.

Example 2.4: Permutation

Let $g : \Omega_5 \to \Omega_5$ such that:

$$g(1) = 2,\ g(2) = 1,\ g(3) = 3,\ g(4) = 5,\ g(5) = 4$$

Then we can more compactly write g as:

$$g = \begin{pmatrix} 1 & 2 & 3 & 4 & 5 \\ 2 & 1 & 3 & 5 & 4 \end{pmatrix}$$

or:

$$g = (1\ 2)(4\ 5)$$

Two more important concepts are the *composition* of two permutations and the *inverse* of a permutation:

Definition 2.5: Permutation Composition

The composition of two permutations $g, h : \Omega \to \Omega$ is another permutation $gh : \Omega \to \Omega$ such that $\forall x \in \Omega : gh(x) = g(h(x))$. Note that this operation is associative but not commutative.

Definition 2.6: Permutation Inverse

The inverse of a permutation $g : \Omega \to \Omega$ is another permutation $g^{-1} : \Omega \to \Omega$ such that $gg^{-1} = g^{-1}g = 1$.

Example 2.5: Permutation Inverse

Let g be as in Example 2.4 and $h = (1\ 2\ 3\ 4)$ Then we have:

$$gh = \begin{pmatrix} 1 & 2 & 3 & 4 & 5 \\ 3 & 2 & 4 & 5 & 1 \end{pmatrix} = (1\ 3\ 4\ 5)$$
$$hg = \begin{pmatrix} 1 & 2 & 3 & 4 & 5 \\ 1 & 3 & 5 & 2 & 4 \end{pmatrix} = (2\ 3\ 5\ 4)$$

And:

$$g^{-1} = g$$
$$h^{-1} = \begin{pmatrix} 1 & 2 & 3 & 4 & 5 \\ 4 & 1 & 2 & 3 & 5 \end{pmatrix} = (1\ 4\ 3\ 2)$$

2.1.3 Representing Permutations on a Computer

When thinking about how to most efficiently represent permutations in a computer program we have to consider which operations we want to perform on them. The most important ones are the following:

- Evaluate $g(x)$ for any $x \in \Omega$ (1)
- Obtain the composition gh of g and h (2)
- Obtain the inverse g^{-1} of g (3)

Intuitively, if $\Omega = \Omega_n$ we can represent a permutation g by using an array `arr` of n unsigned integer entries such that (assuming one-based-indicing) $\forall x \in \Omega : g(x) = y \Leftrightarrow \texttt{arr}[x] = y$. This representation allows (1) to be evaluated in $O(1)$ time, and (2), (3) to be obtained in $O(n)$ time.

Another possibility is representing g as a *permutation word*, i.e. several permutations $g_1, g_2, \ldots, g_m$, each of which is in turn represented by an array as above, such that $g = g_1 g_2 \cdots g_m$. With this representation, evaluating (1) takes $O(m)$ time and obtaining (3) takes $O(mn)$ time. However, obtaining (2) is now possible in $O(1)$ time because composing two permutation words simply requires concatenating their respective representations. This representation can thus be beneficial in contexts where permutations are evaluated infrequently but composed frequently[2]. To prevent permutation words from growing too large they can be *nomalized* when appropriate, i.e. their representation can be collapsed into a single array in $O(mn)$ time.

[2]`mpsym` does not make use of permutation words since preliminary experiments found that using them were most appropriate did not result in a notable performance increase.

2.2 Permutation Groups

2.2.1 Motivation

We have seen in Section 2.1.1 how we can use permutations to describe graph automorphisms which capture symmetries inherent in graphs. An obvious question now is how we can determine and describe the set of *all* such symmetries for a given graph. A first step in this direction is the observation that both "composition" and "inversion" of symmetries again result in symmetries.

Thinking back to Example 2.3, we interpreted the presented automorphism as a 90° right rotation about the center of a square. Let us refer to this automorphism as autom_1. Another automorphism, autom_2 of G which exchanges vertices 2 and 4 corresponds to a reflection across the squares upper left to lower right diagonal. Notice that we can apply autom_1 and autom_2 in arbitrary order over and over again without ever producing a graph that is not identical to G. The same is true when also considering the inverse of autom_1, i.e. a 90° left rotation (autom_2 is its own inverse).

We can generalize this observation and state that given any two automorphisms of some undirected graph G represented as permutations g and h, the combination gh as well as the inverses g^{-1} and h^{-1} also represent automorphisms of G. This implies that the set of all automorphisms of G is *closed under composition and inversion* which leads directly to the concept of a *permutation group*.

2.2.2 Definition

Definition 2.7: Group

A (finite) group is a tuple $(G, \circ)$ where G is a set and $\circ : G \times G \to G$ is a binary operator. Instead of $(G, \circ)$ we usually just write G if the context is clear and instead of $\circ((g_1, g_2))$ we write $g_1 \circ g_2$ or simply $g_1 g_2$. We denote the *order* of a group, i.e. the number of elements it contains by $|G|$. Any group must satisfy the following criteria:

- $\forall g_1, g_2 \in G : g_1 g_2 \in G$ (closedness under $\circ$)
- $\forall g_1, g_2, g_3 \in G : g_1(g_2 g_3) = (g_1 g_2) g_3$ (associativity of $\circ$)
- $\exists e \in G : \forall g \in G : eg = ge = g$ (existence of identity)
- $\forall g \in G : \exists g^{-1} \in G : gg^{-1} = g^{-1}g = e$ (existence of inverse)

From Definition 2.7 it can easily be derived that the identity element e and the inverse of any $g \in G$ must be unique. Any group of order one must necessarily only contain this identity element. We call such a group *trivial* and denote it by 1.

Example 2.6: Group

The set of all permutations on a set Ω, denoted $\text{Sym}(\Omega)$ (if $\Omega = \Omega_n$ we also write S_n), together with the composition of permutations forms a group as can be easily verified.

An important related concept is that of *subgroups*:

Definition 2.8: Subgroup

If $H \subseteq G$ and $(H, \circ|_H)$ is a group (where $\circ|_H$ is $\circ$ restricted to the elements of H), we say that H is a *subgroup* of G. We also write $H \leq G$.

We call any $G \leq \mathrm{Sym}(\Omega)$ a *permutation group* and say that G *acts on* Ω.

Example 2.7: Subgroup

The set $\{(),\ (1\ 2\ 3), (1\ 3\ 2)\}$ together with the usual permutation composition operator forms a subgroup of S_3. More specifically, this group is the *cyclic group* of degree three, also denoted C_3.

We can characterise groups in general and permutation groups in particular via a *generating set*:

Definition 2.9: Generating Set

Let $(G, \circ)$ be a group and $S \subseteq G$, then $\langle S \rangle$ is the intersection of all subgroups of $(G, \circ)$ that contain S. If $\langle S \rangle = H$ we say that S *generates* H.

Intuitively this means that every element of $\langle S \rangle$ can be obtained by composing (possibly inverted) elements of S via the $\circ$ operator. Groups can have many different generating sets and some generating sets may be redundant, i.e. removing an element from the generating set does not change the generated group. Very large groups can often be represented by comparatively small generating sets.

Example 2.8: Generating Set

$S_n = \langle \{(1\ 2), (1\ 2\ \dots\ n)\} \rangle$ with $|S_n| = n!$.

Another method of characterising groups are *presentations* which generalize generating sets. Informally, a presentation $\langle S \mid R \rangle$ consists of a set of generators S and a set of *relations* R.

Example 2.9: Presentation

$S_n = \langle \{i\ i{+}1) \mid 1 \leq i < n\} \rangle$.

As we shall see, both these representations, while often convenient for constructing mathematical proofs are not well suited to represent permutation groups in a computer program.

2.2.3 Orbits

A concept related to permutation groups that will be essential when formalizing the TMOR problem in Chapter 5 is that of *permutation group orbits*:

Definition 2.10: Permutation Group Orbit

Given a permutation group G acting on a set Ω, the permutation group orbit x^G of some $x \in \Omega$ is the set $\{g(x) \mid g \in G\}$.

Example 2.10: Permutation Group Orbit

Consider the permutation group $G = \langle\{(1\ 2), (2\ 3), (4\ 5)\}\rangle$ acting on Ω_5, then we have:

$$\begin{aligned} 1^G = 2^G = 3^G &= \{1,2,3\} \\ 4^G = 5^G &= \{4,5\} \end{aligned}$$

It follows directly from Definition 2.10 and the fact that every $g \in G$ has an inverse that:

Lemma 2.1: Disjointness of Permutation Group Orbits

For two $x_1, x_2 \in \Omega$, x_1^G and x_1^G are either identical or disjoint.

Based on this, we can formulate the following equivalence relation:

Definition 2.11: $\sim_G$

Let $\sim_G \subseteq \Omega \times \Omega$ with $(x_1, x_2) \in \sim_G \Leftrightarrow x_1^G = x_2^G$. Instead of $(x_1, x_2) \in \sim_G$ we write $x_1 \sim_G x_2$. For each $x \in \Omega$ we denote the equivalence class of x by $[x]_{\sim_G}$ and note that the equivalence classes partition Ω into the quotient set $\Omega/{\sim_G} = \{[x]_{\sim_G} \mid x \in \Omega\}$. Given some (usually obvious) total strict order $<$ on Ω we can define a canonical representative for every equivalence class as $\mathrm{repr}([x]_{\sim_G}) = \min_<([x]_{\sim_G})$.

Example 2.11: $\sim_G$

Let G be as in Example 2.10. Then we have:

$$\begin{aligned} [1] = [2] = [3] &= \{1,2,3\},\ \mathrm{repr}(\{1,2,3\}) = 1 \\ [4] = [5] &= \{4,5\},\ \mathrm{repr}(\{4,5\}) = 4 \end{aligned}$$

And $\Omega = \{1,2,3\} \sqcup \{4,5\}$.

It is interesting to ask how many such equivalence classes exist for a given G acting on Ω. An answer is provided by the following lemma, proved e.g. in [15] Chapter 2:

Lemma 2.2: Cauchy-Frobenius Lemma

For all $g \in G$, let $\Omega^g = \{x \in \Omega \mid g(x) = x\}$, then it holds that $|\Omega /{\sim_G}| = \frac{1}{|G|} \sum_{g \in G} |\Omega^g|$, i.e. the number of equivalence classes of $\sim_G$ is the average number of elements of Ω stabilized by the elements of G.

Example 2.12: Cauchy-Frobenius Lemma

Let G be as in Example 2.10. As can be readily verified $|G| = 12$. We can explicitly determine $|\Omega^g|$ for all $g \in G$:

$$\begin{aligned}
&|\Omega^{()}| = |\{1,2,3,4,5\}| = 5 && |\Omega^{(1,3)(4,5)}| = |\{2\}| = 1 \\
&|\Omega^{(1,2)}| = |\{3,4,5\}| = 3 && |\Omega^{(1,3,2)}| = |\{4,5\}| = 2 \\
&|\Omega^{(1,2)(4,5)}| = |\{3\}| = 1 && |\Omega^{(1,3,2)(4,5)}| = |\emptyset| = 0 \\
&|\Omega^{(1,2,3)}| = |\{4,5\}| = 2 && |\Omega^{(2,3)}| = |\{1,4,5\}| = 3 \\
&|\Omega^{(1,2,3)(4,5)}| = |\emptyset| = 0 && |\Omega^{(2,3)(4,5)}| = |\{1\}| = 1 \\
&|\Omega^{(1,3)}| = |\{2,4,5\}| = 3 && |\Omega^{(4,5)}| = |\{1,2,3\}| = 3
\end{aligned}$$

Averaging these set sizes we obtain $\frac{1}{12}(5+3+1+2+0+3+1+2+0+3+1+3) = 2$ which is, as expected, equal to the number of permutation group orbits we determined in Example 2.10.

Another concept we will encounter again later is that of *transitive* permutation groups:

Definition 2.12: Transitive Permutation Group

If $|\Omega /{\sim_G}| = 1$ we say that G is transitive.

Example 2.13: Transitive Permutation Group

The permutation group $G = \langle\{(1\ 2), (2\ 3), (3\ 4), (4\ 5)\}\rangle$ is transitive.

2.2.4 Direct and Wreath Product

To conclude our theoretical discussion of permutation groups we will now define two binary operators that combine permutation groups, namely the *direct product* and the *wreath product*. We will need these concepts in Section 4.3 where we use them to decompose the automorphism groups of certain separable and hierarchical graphs.

We will first introduce the relatively straightforward direct product for the special case of permutation groups acting on Ω_n:

Definition 2.13: Direct Product

Given two permutation groups $G = \langle S_G \rangle$ and $H = \langle S_H \rangle$ with G acting on Ω_{n_1} and H acting on Ω_{n_2}, the direct product of G and H, denoted $G \times H$ is a permutation group acting on $\Omega_{n_1+n_2}$ such that $G \times H = \langle S_G \cup S'_H \rangle$ where S'_H is S_H "shifted upwards" such that it acts on $\{n_1 + 1, \ldots, n_1 + n_2\}$.

The direct product is somewhat analogous to the cartesian product of sets. In particular, we have:

Lemma 2.3: Direct Product Order

Given two permutation groups G and H it holds that $|G \times H| = |G| \cdot |H|$.

Example 2.14: Direct Product

Given:

$$G = \langle \{(1\ 2), (1\ 3), (2\ 3)\} \rangle$$
$$H = \langle \{(1\ 3), (2\ 4), (1\ 2)(3\ 4)\} \rangle$$

acting on Ω_3 and Ω_4 respectively, we have:

$$G \times H = \langle \{(1\ 2), (1\ 3), (2\ 3), (4\ 6), (5\ 7), (4\ 5)(6\ 7)\} \rangle$$

and:

$$|G \times H| = |G| \cdot |H| = 6 \cdot 8 = 48$$

The wreath product is slightly more involved, instead of a formal definition we will try to give an intuitive understanding of it by means of what we shall call undirected *hypergraphs*[3]. Given a "proto" undirected graph $G_{\text{proto}} = (V_{\text{proto}}, E_{\text{proto}})$ and a "super" undirected graph $G_{\text{super}} = (V_{\text{super}}, E_{\text{super}})$ we construct such an undirected hypergraph by fully connecting $|V_{\text{super}}|$ "instances" of G_{proto} according to E_{super}. More formally:

Definition 2.14: Hypergraph

Given two undirected graphs $G_{\text{proto}} = (V_{\text{proto}}, E_{\text{proto}})$ and $G_{\text{super}} = (V_{\text{super}}, E_{\text{super}})$, we define their undirected hypergraph $G_{\text{hyper}} = (V_{\text{hyper}}, E_{\text{hyper}})$ as follows:

$$V_{\text{hyper}} = \{1, 2, \ldots, |V_{\text{proto}}| \cdot |V_{\text{super}}|\}$$
$$E_{\text{hyper}} = \{\{i, j\} \mid \{p(i), p(j)\} \in E_{\text{proto}} \lor \{s(i), s(j)\} \in E_{\text{super}}, i, j \in V_{\text{hyper}}\}$$

[3]Note that the definition we give here deviates slightly from what is commonly understood by the term.

where:

$$p : V_{\text{hyper}} \to V_{\text{proto}},\ p(i) = ((i-1) \bmod |V_{\text{proto}}|) + 1$$
$$s : V_{\text{hyper}} \to V_{\text{super}},\ s(i) = \left\lfloor \frac{i-1}{|V_{\text{proto}}|} \right\rfloor + 1$$

It can be shown that the wreath product the automorphism groups of G_{proto} and G_{super}, denoted $G_{\text{proto}} \wr G_{\text{super}}$, is a subgroup of, and often identical to, the automorphism group of G_{hyper}. The conditions under which identity holds are known but highly technical, refer to [9]. We will for now always assume identity, at the risk of "missing out on" a number of automorphisms when decomposing automorphism groups into wreath products in Section 4.3.

Example 2.15: Wreath Product

Given two undirected graphs which have automorphism groups G and H from Example 2.14 respectively, their undirected hypergraph has automorphism group $G \wr H$ with:

$$\begin{aligned} G \wr H = \langle\{&(1\ 2), (1\ 3), (2\ 3), (4\ 10)(5\ 11)(6\ 12), \\ &(1\ 4)(2\ 5)(3\ 6)(7\ 10)(8\ 11)(9\ 12)\}\rangle \end{aligned}$$

and:

$$|G \wr H| = |G|^4 \cdot |H| = 6^4 \cdot 8 = 10368$$

As with the direct product, we can easily determine $|G \wr H|$ from $|G|$ and $|H|$:

Theorem 2.1: Wreath Product Order

Let G and H be permutation groups acting on Ω_{n_1} and Ω_{n_2} respectively and assume that H does not stabilize n_2, then it holds that $|G \wr H| = |G|^{n_2} \cdot |H|$.

2.2.5 Representing Permutation Groups on a Computer

While a generating set completely defines a permutation group, it is not a suitable representation when considering which common operations we would like to be able to efficiently perform on permutation groups. In particular, several algorithms we will encounter in later chapters will require us to be able to do the following given a permutation group G acting on a set Ω:

- Iterate over all $g \in G$ (1)
- Test whether $g \in G$ for any $g \in \text{Sym}(\Omega)$ (2)

In principle, a generating set allows for both of these operations. From the definition of a generating set it directly follows that every element of a permutation group G can be obtained via the simple fixed point Algorithm 2.1. However, this algorithm is inefficient in two ways: For one it requires us to somehow store every element of G (or a hash thereof) since otherwise we would have no way of

Algorithm 2.1 Naive group enumeration.

```
procedure ENUMERATE_GROUP_NAIVE($G = \langle S \rangle$)
    $S' \leftarrow S \cup \{s^{-1} \mid s \in S\}$

    elements $\leftarrow \{\}$

    while elements is changing do
        for $g \in$ elements, $h \in S'$ do
            elements $\leftarrow$ elements $\cup \{gh\}$
        end for
    end while

    return elements
end procedure
```

knowing if a newly generated element has already been generated previously. Additionally, some elements of G may be generated multiple times. We can then perform membership testing for some permutation h by comparing h to every $g \in G$.

Ideally we would like to use an alternative permutation group representation that allows us to uniquely enumerate all group elements in a deterministic order without the need to store all of them and to perform membership testing with time and space complexity independent of $|G|$. Luckily, such a representation does indeed exist and takes the form of a *base and strong generating set* (short *BSGS*). The rest of this section defines what a BSGS is and how we can utilize it for efficient group enumeration and membership testing. It is based on the information presented in [10] Chapter 4. Constructing a BSGS from a given generating set is such an important topic that we will discuss it separately in Chapter 3.

We begin with a preliminary definition:

Definition 2.15: Stabilizer Subgroup

Given a permutation group G acting on a set Ω and a sequence $X = (x_1, x_2, \ldots, x_k),\ x_i \in \Omega,\ \forall 1 \leq i \leq k$ the *stabilizer subgroup* $G_{x_1,x_2,\ldots,x_k} \leq G$ is the largest subgroup of G whose elements stabilize every element of X, i.e. $\forall g \in G_{x_1,x_2,\ldots,x_k} : X^g = X$.

We can now define:

Definition 2.16: BSGS

A BSGS for a permutation group G acting on a set Ω is a tuple (B, S). The *base* B is a sequence $B = (\beta_1, \beta_2, \ldots, \beta_k),\ \beta_i \in \Omega, \forall 1 \leq i \leq k$ which is not stabilized by any element of G, i.e. $\forall g \in G, g \neq 1 : \exists \beta \in B : g(\beta) \neq \beta$. The *strong generating set* S is a set of permutations that generates G (i.e. $G = \langle S \rangle$) and additionally contains generators for every *basic stabilizer* $G^{(i)} = G_{\beta_1,\beta_2,\ldots,\beta_{i-1}},\ \forall 1 \leq i \leq k+1$. We introduce the notation $S^{(i)} = S \cap G^{(i)}$ and it must hold that $G^{(i)} = \left\langle S^{(i)} \right\rangle$. It is easy to see that for the basic stabilizers it holds that $1 = G^{(k+1)} \leq G^{(k)} \leq \cdots \leq G^{(1)} = G$.

Example 2.16: BSGS

Let $G = S_5$, then one possible BSGS of G is:

$$\begin{aligned} B &= (1, 2, 3, 4) \\ S &= \{(4\ 5), (3\ 5), (2\ 5), (1\ 5)\} \end{aligned}$$

and we have:

$$\begin{aligned} &S^{(1)} = S && G^{(1)} = G \\ &S^{(2)} = \{(4\ 5), (3\ 5), (2\ 5)\} && G^{(2)} = \langle S^{(2)} \rangle = G_1 \\ &S^{(3)} = \{(4\ 5), (3\ 5)\} && G^{(3)} = \langle S^{(3)} \rangle = G_{1,2} \\ &S^{(4)} = \{(4\ 5)\} && G^{(4)} = \langle S^{(4)} \rangle = G_{1,2,3} \\ &S^{(5)} = \{\} && G^{(5)} = 1 = G_{1,2,3,4} \end{aligned}$$

Note that while every permutation group can be represented by a BSGS, a permutation groups BSGS is not unique[4].

To make full use of the BSGS representation, we additionally have to augment (B, S) with the following data structures:

Definition 2.17: Basic Orbits and Transversals

Given a BSGS (B, S) with $B = (\beta_1, \beta_2, \ldots, \beta_k)$, for every $1 \leq i \leq k$ we define an associated *basic orbit* $\Delta^{(i)} = \beta_i^{G^{(i)}}$ and a *transversal* $U^{(i)} = \{u_x^{(i)} \in G \mid x \in \Delta^{(i)}\}$ where $u_x^{(i)}$ is some permutation such that $u_x^{(i)}(\beta_i) = x$. We further define $\Delta = (\Delta_1, \Delta_2, \ldots, \Delta_k)$ and $U = (U^{(1)}, U^{(2)}, \ldots, U^{(k)})$. From now on we write (B, S, Δ, U) to explicitly denote a BSGS plus associated basic orbits and transversals.

Assuming that we can efficiently obtain B, S, Δ and U for some group G, the following theorem provides us with the means to efficiently enumerate G:

Theorem 2.2: Transversal Normal Form

Given a BSGS (B, S, Δ, U) for a permutation group G with a base of size k, every $g \in G$ has a unique representation of the form $g = u_k u_{k-1} \ldots u_1,\ u_i \in U^{(i)}, \forall 1 \leq i \leq k$.

[4]Except for the trivial permutation group 1.

Thus we can systematically enumerate G by composing all combinations of elements from $U^{(k)}, \dots, U^{(1)}$. We can then also easily calculate $|G| = |U^{(k)}| \cdot |U^{(k-1)}| \cdot \dots \cdot |U^{(1)}|$.

Example 2.17: Transversal Normal Form

Let $G = A_4$, the *alternating group* of degree four, also denoted A_4. One possible BSGS for A_4 is $(B = (1, 2),\ S = \{(2\ 3\ 4), (1\ 3\ 4)\})$ such that the basic stabilizers, basic orbits and transversals are as follows:

$$\begin{array}{lll} S^{(1)} = S & \Delta^{(1)} = 1^S = \{1, 2, 3, 4\} & U^{(1)} = \{(), (1\ 2\ 4), (1\ 3\ 4), (1\ 4\ 3)\} \\ S^{(2)} = \{(2\ 3\ 4)\} & \Delta^{(2)} = 2^{\{(2\ 3\ 4)\}} = \{2, 3, 4\} & U^{(2)} = \{(), (2\ 3\ 4), (2\ 4\ 3)\} \end{array}$$

It is well known that $|A_n| = n!/2$, so in this case we should have $|G| = |A_4| = 4!/2 = 12$ and indeed we find that $|U^{(2)}| \cdot |U^{(1)}| = 3 \cdot 4 = 12$. Iterating over all combinations of elements $u_2 \in U^{(2)}, u_1 \in U^{(1)}$ we can obtain all elements of G:

$$\begin{array}{ll} u_{2,1}u_{1,1} = ()() = () & u_{2,2}u_{1,3} = (2\ 3\ 4)(1\ 3\ 4) = (1\ 3)(2\ 4) \\ u_{2,1}u_{1,2} = ()(1\ 2\ 4) = (1\ 2\ 4) & u_{2,2}u_{1,4} = (2\ 3\ 4)(1\ 4\ 3) = (1\ 4\ 2) \\ u_{2,1}u_{1,3} = ()(1\ 3\ 4) = (1\ 3\ 4) & u_{2,3}u_{1,1} = (2\ 4\ 3)() = (2\ 4\ 3) \\ u_{2,1}u_{1,4} = ()(1\ 4\ 3) = (1\ 4\ 3) & u_{2,3}u_{1,2} = (2\ 4\ 3)(1\ 2\ 4) = (1\ 2)(3\ 4) \\ u_{2,2}u_{1,1} = (2\ 3\ 4)() = (2\ 3\ 4) & u_{2,3}u_{1,3} = (2\ 4\ 3)(1\ 3\ 4) = (1\ 3\ 2) \\ u_{2,2}u_{1,2} = (2\ 3\ 4)(1\ 2\ 4) = (1\ 2\ 3) & u_{2,3}u_{1,4} = (2\ 4\ 3)(1\ 4\ 3) = (1\ 4)(2\ 3) \end{array}$$

Algorithm 2.2 Test permutation group membership.

```
procedure IS_MEMBER($g, (B, S, \Delta, U)$)
    $g' \leftarrow g$

    for $i = 1, 2, \dots, k$ do
        if $g'(\beta_i) \notin \Delta^{(i)}$ then
            return false
        end if

        Find $u_i \in U^{(i)}$ such that $u_i(\beta_i) = g'(\beta_i)$

        $g' \leftarrow g' u_i^{-1}$
    end for

    if $g' = 1$ then
        return true
    else
        return false
    end if
end procedure
```

Furthermore we can perform membership testing via Algorithm 2.2. The idea behind this algorithm is simple: since by Theorem 2.2 any $g \in G$ can be decomposed as $g = u_k u_{k-1} \dots u_1$

we can attempt to "reduce" g to 1 by composing g (step by step) with $u_1^{-1}, u_2^{-1}, \ldots, u_k^{-1}$ so that we end up with $g' = u_k \cdots u_1 u_1^{-1} \cdots u_k^{-1} = 1$. Determining the necessary u_i^{-1} is easy because all $u_x^{(i)} \in U^{(i)}$ stabilize base elements $\beta_1, \beta_2, \ldots, \beta_{i-1}$ as should be obvious from the definition of $U^{(i)}$. Thus, when have reduced g to $g' = u_k u_{k-1} \cdots u_i$ we know that $u_i(\beta_i) = g'(\beta_i)$ and thus we have $u_i = u_{g'(\beta_i)}^{(i)}$. Since any $u_k u_{k-1} \cdots u_1$ produces an element of G, we can be sure that we can reduce g' to 1 iff $g \in G$. For $g \notin G$ Algorithm 2.2 either terminates before we reach this point because encounter a g' for which $g'(\beta_i) \notin \Delta^{(i)}$ so that we can not find an appropriate u_i or the final g' is not the identiy permutation.

Example 2.18: Test Permutation Group Membership

Once again, let $G = A_4$ with the same BSGS as in Example 2.17. We will use Algorithm 2.2 to verify that $(1\ 2)(3\ 4) \in G$:

i	β_i	$g'(\beta_i)$	u_i	g'
0	-	-	-	$(1\ 2)(3\ 4)$
1	1	2	$(1\ 2\ 4)$	$(1\ 2)(3\ 4) \cdot (1\ 4\ 2) = (2\ 4\ 3)$
2	2	4	$(2\ 4\ 3)$	$(2\ 4\ 3) \cdot (2\ 3\ 4) = 1$

For a negative example consider the permutation $(1\ 2\ 3\ 4) \notin G$:

i	β_i	$g'(\beta_i)$	u_i	g'
0	-	-	-	$(1\ 2\ 3\ 4)$
1	1	2	$(1\ 2\ 4)$	$(1\ 2\ 3\ 4) \cdot (1\ 4\ 2) = (2\ 3)$
2	2	3	$(2\ 3\ 4)$	$(2\ 3) \cdot (2\ 4\ 3) \neq 1$

2.3 Partial Permutations

2.3.1 Motivation

We have seen in Section 2.1 how we can represent symmetries of graphs via graph automorphisms described by permutations. But remember that in Section 1.3 we hinted at the fact that we are interested not only in symmetries but also in *partial* symmetries of graphs. In this section we will show how to represent these partial symmetries via *partial graph automorphisms* described by *partial permutations*. We begin with a preliminary definition:

Definition 2.18: Vertex-Induced Subgraph

Given an undirected graph $G = (V, E)$, a vertex-induced subgraph of G is another undirected graph $G_{\text{sub}} = (V_{\text{sub}}, E_{\text{sub}})$ such that $V_{\text{sub}} \subseteq V$ and $E_{\text{sub}} = \{\{i, j\} \mid \{i, j\} \in E, i, j \in V_{\text{sub}}\}$.

Example 2.19: Vertex-Induced Subgraph

Consider again the undirected graph G from Example 2.1. The graph $G_{\mathrm{sub}} = (V_{\mathrm{sub}}, E_{\mathrm{sub}})$, visualized in Figure 2.4, with $V_{\mathrm{sub}} = \{1, 2, 3\}$ and $E_{\mathrm{sub}} = \{\{1, 2\}, \{2, 3\}\}$ is a vertex-induced subgraph of G.

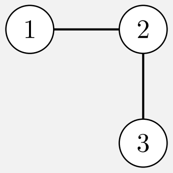


Figure 2.4: G_{sub}

With this definition out of the way we can now define:

Definition 2.19: Partial Graph Automorphism

Given an undirected graph G, a partial automorphism of G is an isomorphism from one vertex-induced subgraph of G to another.

Example 2.20: Partial Graph Automorphism

Consider the undirected graph G presented in Figure 2.5. The function pautom : $\{1, 4, 7\} \to \{2, 5, 8\}$ with:

$$\mathrm{pautom}(1) = 2,\ \mathrm{pautom}(4) = 5,\ \mathrm{pautom}(7) = 8,$$

is a partial automorphism of G because it is an isomorphism from the vertex-induced subgraph $G_{\mathrm{sub},1} = (\{1, 4, 7\}, \dots)$ to the vertex-induced subgraph $G_{\mathrm{sub},2} = (\{2, 5, 8\}, \dots)$.

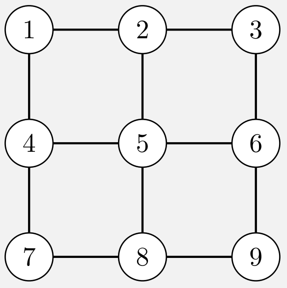


Figure 2.5: G

Note that every automorphism of an undirected graph G is also a partial automorphism of G but the converse does not necessarily hold so that conceptually, partial graph automorphisms generalize graph automorphisms. Thus it seems natural that we can describe them by generalized permutations. These are exactly the partial permutations this section is concerned with.

2.3.2 Definition

Definition 2.20: Partial Permutation

A partial permutation is a bijection not necessarily from a set Ω to itself but from some *domain* $\mathrm{Dom}_\Omega \subseteq \Omega$ to an *image* $\mathrm{Im}_\Omega \subseteq \Omega$.

We can view ordinary permutations as special partial permutations for which $\mathrm{Dom}_\Omega = \mathrm{Im}_\Omega = \Omega$.

We will denote partial permutations with the letters s and t and write $\mathrm{dom}(s)$ and $\mathrm{im}(s)$ to denote the domain and image of some partial permutation s. Similar to ordinary permutations we can notate partial permutations in two-line notation or *chain/cycle notation*. The former should be self explanatory, the latter needs to account for the fact that not every element of Dom_Ω needs to map back to Dom_Ω. For a partial permutation s, we call a sequence of elements of $\mathrm{dom}(s)$ resulting from repeated application of s which is terminated by some element of $\mathrm{im}(s)$ a *chain* and use square brackets to differentiate it from a cycle.

Example 2.21: Partial Permutation

Let $\Omega = \Omega_5$ and $s : \{1, 2, 3, 5\} \to \{1, 2, 4, 5\}$ such that:

$$s(1) = 2,\ s(2) = 1,\ s(3) = 4,\ s(5) = 5$$

Then we can write s as:

$$s = \begin{pmatrix} 1 & 2 & 3 & 4 & 5 \\ 2 & 1 & 4 & - & 5 \end{pmatrix}$$

or:

$$s = (1\ 2)[3\ 4](5)$$

Note that we did not omit the cycle of length one in chain/cycle notation to signify that $5 \in \mathrm{dom}(s)$.

Because we will need it frequently, we also define the application of a partial permutation on a set:

Definition 2.21: Partial Permutation Set Application

Given a partial permutation s and a set X, we define $s(X) = \{s(x) \mid x \in X \cap \mathrm{dom}(s)\}$.

As with ordinary permutations, we can define the composition of two partial permutations and the inverse of a partial permutation:

Definition 2.22: Partial Permutation Composition

The composition of two partial permutations s and t is another partial permutation $st : s^{-1}((\mathrm{im}(s) \cap \mathrm{dom}(t))) \to t((\mathrm{im}(s) \cap \mathrm{dom}(t)))$ such that $\forall x \in \mathrm{dom}(st) : st(x) = s(t(x))$.

Definition 2.23: Partial Permutation Inverse

The inverse of a partial permutation s is another partial permutation $s^{-1} : \mathrm{im}(s) \to \mathrm{dom}(s)$ such that $s = ss^{-1}s$ and $s^{-1} = s^{-1}ss^{-1}$. It also holds that $ss^{-1} = 1_{\mathrm{dom}(s)}$ and $s^{-1}s = 1_{\mathrm{im}(s)}$ where $1_X : X \to X$ is the *identity partial permutation* on X, i.e. $\forall x \in X : 1_X(x) = x$.

Note the difference between this and Definition 2.6.

As with ordinary permutations, composition is associative but not commutative. Note that for any partial permutation s it holds that $1_{\mathrm{dom}(s)}s = s1_{\mathrm{im}(s)} = s$. If $\mathrm{im}(s) \cap \mathrm{dom}(t) = \emptyset$ the resulting partial permutation is the *empty partial permutation* $0 : \emptyset \to \emptyset$ such that for any partial permutation s it holds that $s0 = 0s = 0$.

2.3.3 Representing Partial Permutations on a Computer

To represent a partial permutation s in a computer program, we use an array `arr` of $\max(\mathrm{dom}(s))$ unsigned integer entries such that (assuming one-based indexing) $\forall x \in \mathrm{dom}(s) : s(x) = y \Leftrightarrow \texttt{arr}[x] = y \land \forall x \notin \mathrm{dom}(s),\ 1 \leq x \leq \max(\mathrm{dom}(s)) : \texttt{arr}[x] = 0$.

This representation allows us to evaluate $s(x)$ in $O(1)$ time. Because we store no information about Ω, we simply let this operation return 0 not only for $\forall x \notin \mathrm{dom}(s),\ 1 \leq x \leq \max(\mathrm{dom}(s))$ but also for any other $x \in \mathbb{N}_+$. Obtaining the inverse of s as well as composing s with another partial permutation t is trivially possible in $O(\max(\mathrm{dom}(s))$ time.

We can optionally utilize two separate (sorted) sets to store $\mathrm{dom}(s)$ and $\mathrm{im}(s)$ so that these do not have to be constructed from `arr` when they are required but we have to take care to preserve the correctness of these sets when composing or inverting partial permutations.

2.4 Partial Permutation Inverse Monoids

Similar to how we can represent all symmetries of a graph via permutation groups, we can represent all partial symmetries of a graph via a set of partial permutations closed under composition and inversion. However, these sets do not, in general, form groups but rather *inverse semigroups/monoids* which share some of the properties of groups.

2.4.1 Definition

Given such a set S together with the composition of partial permutations $\circ$, the reason that $(S, \circ)$ does not necessarily form a group is simple: S does not have to contain a unique identity element. Given some $s \in S$ we have that $e_l s = se_r = s$ for any $e_l \supseteq \mathrm{dom}(s),\ e_r \supseteq \mathrm{im}(s)$. However, $(S, \circ)$ is guaranteed to form a *semigroup*, an algebraic structure generalizing the group concept:

Definition 2.24: Semigroup

A (finite) semigroup is as a tuple $(S, \circ)$ where S is a set and $\circ : S \times S \to S$ is a binary operator. A semigroup must satisfy only closedness under and associativity of $\circ$ as given in Definition 2.7.

We can add a further restriction to this definition by noting that:

Lemma 2.4: Uniqueness of Inverse

Let S be a set of partial permutations closed under composition and inversion, then it holds that $\forall s \in S : \exists! s^{-1} \in S : s = ss^{-1}s \wedge s^{-1} = s^{-1}ss^{-1}$.

Proof. Because S is closed under inversion, existence of s^{-1} is trivially given and we only need to prove that s^{-1} is unique. Given some $s \in S$, assume that there are $s_1^{-1}, s_2^{-1} \in S$ such that $s = ss_1^{-1}s$, $s_1^{-1} = s_1^{-1}ss_1^{-1}$ and $s = ss_2^{-1}s$, $s_2^{-1} = s_2^{-1}ss_2^{-1}$. Then we have $s_1^{-1} = s_1^{-1}1_{\mathrm{dom}(s)} = s_1^{-1}ss_2^{-1} = 1_{\mathrm{im}(s)}s_2^{-1} = s_2^{-1}$. □

Motivated by this we introduce *inverse semigroups*:

Definition 2.25: Inverse Semigroup

A (finite) inverse semigroup is a tuple $(S, \circ)$ where S is a set and $\circ : S \times S \to S$ is a binary operator. It must hold that:

- $(S, \circ)$ is a semigroup
- $\forall s \in S : \exists s^{-1} \in S : s = ss^{-1}s,\ s^{-1} = s^{-1}ss^{-1}$ (existence of inverse)

Accordingly, $(S, \circ)$ forms what we will refer to as a *partial permutation inverse semigroup*[5]. We say that $(S, \circ)$ acts on $\Omega = \bigcup_{s \in S} \mathrm{dom}(s)$ and if S additionally contains the element 1_Ω we say that $(S, \circ)$ is an *partial permutation inverse monoid*. The difference between inverse semigroups and inverse monoids is minor but for technical reasons we will from here on only work with the latter.

It is easy to see that every inverse monoid is a semigroup and that every group is an inverse monoid. Figure 2.6 depicts this generalization relationship graphically.

2.4.2 Representing Partial Permutation Inverse Monoids on a Computer

We now present a method of efficiently representing partial permutation inverse monoids in a computer program. This representation can be constructed from a generating set of partial permutations (defined analogously to definition 2.9) and used to perform membership testing and, to some extent, inverse monoid enumeration. Its theoretical underpinnings are more involved than for the relatively straightforward BSGS. For proof of the theoretical soundness of what is presented here refer to [5] which this section is based on.

The central data structure we will make use of is the so called *orbit graph*:

[5]There exist other semigroups than those made up of partial permutations but we will only focus on these particular ones in this thesis.

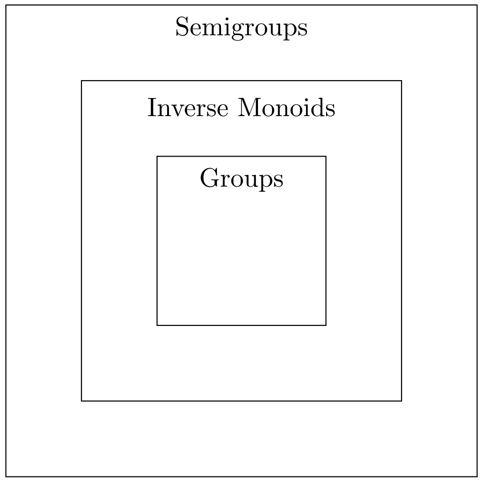


Figure 2.6: Generalization relationship between groups, inverse monoids and semigroups.

Definition 2.26: Orbit Graph

An orbit graph OG of a partial permutation inverse monoid $M = \langle S \rangle$ acting on a set Ω is a directed graph (V, E, L). Let $S' = S \cup \{s^{-1} \mid s \in S\}$, then:

- $V = \{\alpha_1 = \Omega, \alpha_2 \subseteq \Omega, \ldots, \alpha_n \subseteq \Omega\}$ i.e. the nodes of OG are labelled with subsets of Ω one of which, namely α_1 which we call the *root* of OG, is equal to Ω.
- $E : \{(\alpha_i, \alpha_j) \mid \exists s \in S' : s(\alpha_i) = \alpha_j\}$[a]
- $L : E \to M$ where $(\alpha_i, \alpha_j) \mapsto s \in S' \implies s(\alpha_i) = \alpha_j$.

[a]Note that E is a multiset.

Obtaining an orbit graph given a generating set S is straightforward: We simply start out with the root α_1 and "explore" the rest of OG in breadth-first fashion, finding new nodes by taking the image of existing ones under the elements of S' as detailed by Algorithm 2.3.

Example 2.22: Orbit Graph

Consider the partial permutation inverse monoid M acting on $\Omega = \{1, 2, 3\}$ generated by $S = \langle \{s_1 = (1\ 2), s_2 = [1\ 2\ 3], s_3 = [3\ 2\ 1]\} \rangle$ (so that $S' = S$). Figure 2.7 shows the orbit graph for M obtained via Algorithm 2.3. For clarity, all edges to $\alpha_7 = \emptyset$ except one are omitted.

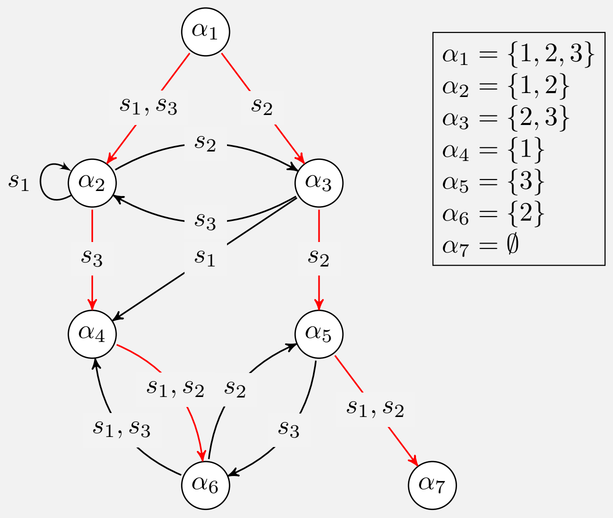


Figure 2.7: Orbit graph and spanning tree for M.

To perform membership testing we also require a *spanning tree* for and the *strongly connected components* (s.c.c.s) of OG.

The spanning tree is a tree whose root is the root of OG and whose nodes are all the nodes of OG. We can construct this tree simply by choosing its edges to be those edges of the orbit graph over which we first "discover" new nodes during the construction of OG.

Example 2.23: Orbit Graph Spanning Tree

A spanning tree (in general there may be several equally valid spanning trees to choose from) for the orbit graph presented in Example 2.22 is indicated by the red edges in Figure 2.7.

Informally, the s.c.c.s of OG are the "coarsest" possible partition of V into sets of vertices for which every vertex is reachable from every other vertex in the set. Finding the s.c.c.s of a directed graph is possible in $O(|V| + |E|)$ time e.g. via *Tarjan's Algorithm* [20].

Example 2.24: Orbit Graph Strongly Connected Components

The s.c.c.s for the orbit graph presented in Example 2.22 are $\{\{\alpha_1\}, \{\alpha_2, \alpha_3\}, \{\alpha_4, \alpha_5, \alpha_6\}, \{\alpha_7\}\}$.

Given an orbit graph OG for a partial permutation inverse monoid M as well as a spanning tree and the s.c.c.s of OG, we can conclusively determine whether $s \in M$ for any partial permutation s as follows:

Firstly, it is trivial to see that $\text{im}(s) \notin V \implies s \notin M$: Because any $s \in M = \langle S \rangle$ can be obtained by composing elements of S', there must a "path" $t_1 \to t_2 \to \cdots \to t_n$, $t_i \in S'$, $\forall 1 \le i \le n$

Algorithm 2.3 Construct orbit graph.

```
procedure CONSTRUCT_ORBIT_GRAPH($M = \langle S \rangle$)
▷ $M$ acts on a set $\Omega$.

    $S' \leftarrow S \cup \{s^{-1} \mid s \in S\}$

    $V \leftarrow \{\alpha_1 = \Omega\}$
    $E \leftarrow \{\}$

    $Q \leftarrow \{\alpha_1\}$
    $D \leftarrow \{\}$

    while $Q \neq \emptyset$ do
        Choose an arbitrary $\alpha \in Q$

        for $s \in S'$ do
            $\alpha' = s(\alpha)$

            $E \leftarrow E \cup \{(\alpha, \alpha')\}$
            $L((\alpha, \alpha')) = s$

            if $\alpha' \notin D$ then
                $Q \leftarrow Q \cup \{\alpha'\}$
            end if
        end for

        $Q \leftarrow Q \setminus \{\alpha\}$
        $D \leftarrow D \cup \{\alpha\}$
    end while

    return OG $= (V, E, L)$
end procedure
```

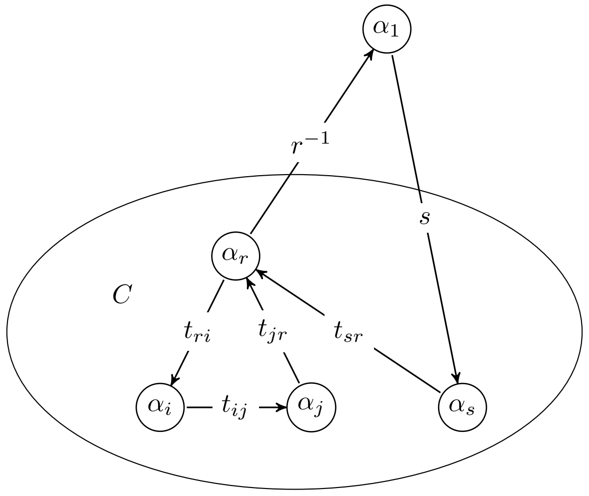


Figure 2.8: Intuition behind $s \in M \iff r^{-1}st_{sr} \in M_{\alpha_r}$.

through OG such that $s = t_1t_2 \cdots t_n$. This implies that $\text{im}(s) = \text{im}(t_2t_2 \cdots t_n) \in V$. Because $s \in M \implies s^{-1} \in M$, we also have $\text{dom}(s) = \text{im}(s^{-1}) \notin V \implies s \notin M$.

Unfortunately, it is possible that $\text{dom}(s), \text{im}(s) \in V$ and we still have $s \notin M$. To check whether this is the case we first have to introduce additional algebraic structures related to the s.c.c.s of OG. Refer to Figure 2.8 throughout the following explanation.

Firstly, for every s.c.c. C of OG we choose an arbitrary $\alpha_r \in C$ to be the *representative* of C. Now we let M_{α_r} be the inverse submonoid (defined analogously to Definition 2.8) of M which contains only partial permutations $t = r^{-1} \cdots$ with $\text{dom}(t) = \text{im}(t) = \alpha_r$ where r is the partial permutation obtained by composing all $t \in S'$ along our spanning tree of OG from α_1 to α_r.

We can obtain a generating set for M_{α_r} as follows: Let $t_{ij} \in M$ be such that $t(\alpha_i) = \alpha_j$. For any two $\alpha_i, \alpha_j \in C$ it is easy to see that $r^{-1}t_{ri}t_{ij}t_{jr} \in M_{\alpha_r}$. In fact, *all* elements of M_{α_r} must correspond to r^{-1} composed with circular paths through C starting and ending at α_r. Intuitively, we should thus have $M_{\alpha_r} = \langle \{ r^{-1}t_{ri}t_{ij}t_{j}r \mid \alpha_i, \alpha_j \in C \} \rangle$.

We can use M_{α_r} to test whether $s \in M$ as follows: Let $\alpha_s = \text{im}(s)$ and α_r be the representative of the s.c.c. in which α_s lies. Then $t = r^{-1}st_{sr}$ is a partial permutation with $\text{dom}(t) = \text{im}(t) = \alpha_r$.

For *all* such t we have $t \in M \iff t \in M_{\alpha_r}$. At first this does not seem to help us greatly because M_{α_r} is also a partial permutation inverse monoid and testing membership in such is what we're trying to accomplish in the first place. We can solve this chicken and egg problem by realising that since $\forall t \in M_{\alpha_r} : \text{dom}(t) = \text{im}(t)$, M_{α_r} is *isomorphic to a permutation group* so we can check whether $t \in M_{\alpha_r}$ via Algorithm 2.2. We also have $s \in M \iff t \in M$ because $t \in M$ iff s actually corresponds to a path through OG. Thus $s \in M \iff t \in M_{\alpha_r}$.

Algorithm 2.4 summarizes the above. Note that we have to find generators and construct BSGSs for all M_{α_r} only once after OG has been constructed. Proving that this algorithm actually returns true iff $s \in M$ is non-trivial, refer to [5].

Algorithm 2.4 Test partial permutation inverse monoid membership.

```
1: procedure IS_MEMBER($s, \mathrm{OG} = (V, E, L)$)
   ▷ OG is an orbit graph for some partial permutation inverse monoid $M = \langle S \rangle$.
2:
3:     if $\mathrm{dom}(s) \notin V \vee \mathrm{im}(s) \notin V$ then
4:         return false
5:     end if
6:
7:     $C \leftarrow$ the s.c.c. such that $\mathrm{im}(s) \in C$
8:     $\alpha_r \leftarrow$ the representative of $C$
9:     $r \leftarrow$ the composition of all $t \in S'$ along the spanning tree of OG from $\alpha_1$ to $\alpha_r$
10:    $M_{\alpha_r} \leftarrow$ the inverse submonoid of $M$ with $\forall t = r^{-1} \cdots \in M_{\alpha_r} : \mathrm{dom}(t) = \mathrm{im}(t) = \alpha_r$
11:
12:    if $r^{-1} s t_{sr} \in M_{\alpha_r}$ then
13:        return true
14:    else
15:        return false
16:    end if
17: end procedure
```

Example 2.25: Test Partial Permutation Inverse Monoid Membership

Let M and OG be as in Example 2.22 and let $\alpha_1, \alpha_2, \alpha_4, \alpha_7$ be the representatives of their respective s.c.c.s. We will use Algorithm 2.4 to verify that $s = (2\ 3) \in M$. We have $\mathrm{dom}(s) = \mathrm{im}(s) = \{2, 3\} = \alpha_3$ such that $\alpha_r = \alpha_2$, $r = (1\ 2)$, and $M_{\alpha_r} = M_{\alpha_2} = \langle \{(1\ 2)\} \rangle$. And it is $r^{-1} s t_{32}^{-1} = (1\ 2)(2\ 3)[1\ 2\ 3] = 0 \in M_{\alpha_2}$. For a negative example consider $s = (1)[3\ 2] \notin M$ which is obviously true because $\mathrm{dom}(s) = \{1, 3\} \notin V$.

We can also use OG to determine $|M|$: It is easy to see that for any $s \in M$, $\mathrm{dom}(s)$ and $\mathrm{im}(s)$ must lie in the same s.c.c. of OG. It also holds that:

Lemma 2.5:

Let M be a partial permutation inverse monoid, OG an orbit graph for M and C an s.c.c. of OG with representative α_r, then there are exactly $|C|^2 \cdot |M_{\alpha_r}|$ partial permutations $s \in M$ with $\mathrm{dom}(s), \mathrm{im}(s) \in C$. Notice that this implies that $|M_{\alpha_r}|$ is independent of our choice of α_r.

Proving this is non-trivial, refer to [5]. It immediately follows that:

Corollary 2.1: Partial Permutation Inverse Monoid Order

Let M be a partial permutation inverse monoid, OG an orbit graph for M and $\mathcal{C}$ the set of all s.c.c.s of OG. Furthermore, let $\alpha_{r,i}$ be the representative of $C_i \in \mathcal{C}$. Then it holds that:

$$|M| = \sum_{C_i \in \mathcal{C}} |C_i|^2 \cdot |M_{\alpha_{r,i}}|$$

Example 2.26: Partial Permutation Inverse Monoid Order

For M from Example 2.22 we have:

$$\begin{aligned} M_{\alpha_1} &= \langle \{1\} \rangle \\ M_{\alpha_2} &= \langle \{(1\ 2)\} \rangle \\ M_{\alpha_4} &= \langle \{1\} \rangle \\ M_{\alpha_7} &= \langle \{1\} \rangle \end{aligned}$$

And thus $|M| = (1^2 \cdot 1) + (2^2 \cdot 2) + (3^2 \cdot 1) + (1^2 \cdot 1) = 19$.

In principle, we could use Lemma 2.5 to somewhat systematically (albeit not as efficiently as we would ideally like) enumerate M by repeatedly applying the equivalent of Algorithm 2.1 for all admissible domains until we have found the right number of partial permutations for each of them.

Chapter 3

BSGS Construction

We will now discuss how to efficiently obtain a BSGS for any permutation group G given a generating set for G. This topic is vast and many variants of the algorithms presented here, utilizing different optimizations and trade-offs, as well as additional BSGS construction algorithms like the *Solvable BSGS Algorithm* have been published in the computational group theory literature. Refer to e.g. [10], Chapter 4 and 6 for a deeper treatment.

As a preliminary matter, in Section 3.1 we will see how to construct the basic orbits $\Delta^{(i)}$ and transversals $U^{(i)}$ from β_i and $S^{(i)}$. In Section 3.2 we will then present the first BSGS construction algorithm, the *Deterministic Schreier-Sims Algorithm*. Finally, in Section 3.3 we present an alternative Monte Carlo BSGS construction algorithm, the *Random Schreier-Sims Algorithm*.

3.1 Representing Basic Orbits and Transversals

Since basic orbits are simply unordered sets of elements of Ω, we can represent them using hash tables and then perform orbit membership tests in $O(1)$ time (on average). We can obtain $\Delta^{(i)}$ from β_i and $S^{(i)}$ via Algorithm 3.5 which is a simple fixed-point algorithm.

Algorithm 3.5 Determine basic orbit.

```
procedure BASIC_ORBIT($\beta_i, S^{(i)}$)
    $\Delta^{(i)} \leftarrow \{\beta_i\}$

    while $\Delta^{(i)}$ is changing do
        for $x \in \Delta^{(i)}$, $g \in S^{(i)}$ do
            $\Delta^{(i)} \leftarrow \Delta^{(i)} \cup \{g(x)\}$
        end for
    end while

    return $\Delta^{(i)}$
end procedure
```

While it is possible to similarly store all $u_x^{(i)} \in U^{(i)}$ explicitly, this can be impractical for groups of large degree because $|U^{(i)}| = |\Delta^{(i)}|$ can be as large as $|\Omega|$. A possible workaround is storing $U^{(i)}$ as a tree structure from which we can reconstruct the $u_x^{(i)}$ on demand. These tree structures are commonly implemented as *(Shallow) Schreier Vectors*, refer to e.g. [10] Chapter 4.

Since for the purpose of this thesis we can assume that $|\Omega| = |\Omega_n|$ is of manageable size[1], we will not consider Schreier Vectors in detail and instead simply present Algorithm 3.6, a modified version of Algorithm 3.5 that determines $U^{(i)}$ in addition to $\Delta^{(i)}$. The necessary modifications should be self-explanatory.

Algorithm 3.6 Determine basic orbit and transversal.

```
procedure BASIC_ORBIT_AND_TRANSVERSAL($\beta_i, S^{(i)}$)
    $\Delta^{(i)} \leftarrow \{\beta_i\}$
    $u_{\beta_i} \leftarrow 1$

    while $\Delta^{(i)}$ is changing do
        for $x \in \Delta^{(i)}$, $g \in S^{(i)}$ do
            if $g(x) \notin \Delta^{(i)}$ then
                $\Delta^{(i)} \leftarrow \Delta^{(i)} \cup \{g(x)\}$
                $u_{g(x)}^{(i)} \leftarrow u_x^{(i)} g$
            end if
        end for
    end while

    $U^{(i)} \leftarrow \{u_x^{(i)} \mid x \in \Delta^{(i)}\}$

    return $\Delta^{(i)}$, $U^{(i)}$
end procedure
```

3.2 The Deterministic Schreier-Sims Algorithm

Let's now turn our attention to a general deterministic algorithm which returns a BSGS for any permutation group given a generating set for that group. This algorithm is commonly refered to as the *Schreier-Sims Algorithm.* Here we will refer to it as the *Deterministic Schreier-Sims Algorithm* to differentiate it from the *Random Schreier-Sims Algorithm* introduced in Section 3.3.

The Deterministic Schreier-Sims Algorithm makes use of Algorithm 3.7, a slightly modified version of Algorithm 2.2 which returns both g' and the loop index i. Importantly, since in every iteration we "strip" u_i from g' by multiplying it with u_i^{-1}, at the start of a given iteration i we know that the returned g' is of the form $g' = u_k u_{k-1} \cdots u_i$ and thus stabilizes $\beta_1, \ldots, \beta_{i-1}$.

Algorithm 3.8 is a high-level description of the Deterministic Schreier-Sims Algorithm. It is based on a simple observation: Referring back to Definition 2.16 it is easy to see that for a valid

[1]Because MPSoCs generally do not contain thousands of processing elements, let alone millions.

Algorithm 3.7 Strip

```
procedure STRIP($g, (B, S, \Delta, U)$)
    $g' \leftarrow g$

    for $i = 1, 2, \ldots, k$ do
        if $g'(\beta_i) \notin \Delta^{(i)}$ then
            return $g$, $i$
        end if

        Find $u_i \in U^{(i)}$ such that $u_i(\beta_i) = g'(\beta_i)$

        $g' \leftarrow g' u_i^{-1}$
    end for

    return $g$, $k + 1$
end procedure
```

BSGS with base of length k for some permutation group G it holds that $G^{(k+1)} = 1$ and $G^{(i)}_{\beta_i} = G^{(i+1)}$ because both $G^{(i)}$ and $G^{(i+1)}$ stabilize $\beta_1, \beta_2, \ldots, \beta_{i-1}$ and $G^{(i+1)}$ additionally stabilizes β_i.

The Deterministic Schreier-Sims algorithm exploits this property as follows: We start by extending an initially empty base B until no $g \in G$ stabilizes every $\beta \in B$, such that we have $G^{(k+1)} = 1$ (line 3) and then initialize the basic stabilizer generating sets $S^{(i)}$ as well as the basic orbits $\Delta^{(i)}$ and transversals $U^{(i)}$ (lines 5–8).

We then assure that $G^{(i)}_{\beta_i} = G^{(i+1)}$ for $i = k, k-1, \ldots, 1$. It is easy to see that $G^{(i+1)} \leq G^{(i)}_{\beta_i}$ always holds and so the problem reduces to assuring that $G^{(i)}_{\beta_i} \leq G^{(i+1)}$. Assuming for a moment that we know a generating set for $G^{(i)}_{\beta_i}$, we can accomplish this by iterating over this generating set and adjoining any generator g that is not already an element of $G^{(i+1)}$ to $S^{(i+1)}$. In fact, given $g', j = \text{STRIP}(g)$ it suffices to adjoin g' because we can represent g as $g = g' u_{j-1} \cdots u_1$ where $u_{j-1}, \ldots, u_1$ are already elements of $U_{j-1}, \ldots, U_1$ (lines 14–18). Finding the generators of $G^{(i)}_{\beta_i}$ is possible by making use of *Schreier's Subgroup Lemma*, refer to [10] Chapter 4.

This process is complicated by the fact that in a certain iteration i we need to adjoin g' not only to $S^{(i+1)}$ but to $S^{(i+1)}, \ldots, S^{(j)}$ (lines 25–28) and then reset i to j because $G^{(j)}_{\beta_j} \leq G^{(j+1)}$ might not hold anymore (lines 29–30). If $j = k + 1$ then g' stabilizes all $\beta \in B$ and thus we also need to adjoin an additional base point not stabilized by g' to B such that $G^{(k+1)} = 1$ remains true (lines 20–23).

For a permutation group G acting on Ω_n, the time complexity of the Deterministic Schreier-Sims is polynomial in n. For an overview of the time complexity of several variants of this algorithm refer to [2] Chapter 14.

3.3 The Random Schreier-Sims Algorithm

While the Deterministic Schreier-Sims Algorithm is guaranteed to terminate and produce a correct BSGS for every possible generating set it can be slow in practice. With the Random Schreier-Sims

Algorithm 3.8 The Deterministic Schreier-Sims Algorithm

```
procedure SCHREIER_SIMS_DETERMINISTIC($B, S$)
▷ $B$ initially is an empty base.
▷ $S$ initially is any generating set for $G$.

    Append elements $x \in \Omega$ to $B$ until no $g \in G$ stabilizes $B$

    for $1 \leq i \leq k$ do
        $S^{(i)} = S \cap G_{\beta_1,\beta_2,\ldots,\beta_{i-1}}$
        $\Delta^{(i)}, U^{(i)}$ = BASIC_ORBIT_AND_TRANSVERSAL($\beta_i, S^{(i)}$)
    end for

    $i \leftarrow 1$
    while $i \geq 1$ do
next :
        for every generator $g$ of $G^{(i)}_{\beta_i}$ do
            $g', j \leftarrow$ STRIP($g$)

            if $g' = 1$ then
                continue
            end if

            if $j = k + 1$ then
                Append an $x \in \Omega$ with $g'(x) \neq x$ to $B$
                $S^{(k)} \leftarrow \{\}$
            end if

            for $l = i + 1, \ldots, j$ do
                $S^{(l)} \leftarrow S^{(l)} \cup \{g'\}$
                $\Delta^{(l)}, U^{(l)}$ = BASIC_ORBIT_AND_TRANSVERSAL($\beta_l, S^{(l)}$)
            end for

            $i \leftarrow j$
            goto next
        end for

        $i \leftarrow i - 1$
    end while

    return $(B, S, \Delta, U)$
end procedure
```

Algorithm there exists a very fast Monte Carlo algorithm with which we can potentially find a BSGS quickly, even for groups of large degree, the drawback being that this BSGS is not guaranteed to be valid.

Despite its name, the Random Schreier-Sims algorithm is not based on Schreier's Subgroup Lemma at all but rather on the following observations:

- It is possible to efficiently and uniformly sample elements from a permutation group without the need to construct a BSGS first, refer to [10] Chapter 3.
- Given some $g \in G$, it can be shown that Algorithm 3.7 returns a permutation $g' = 1$ with probability $< 1/2$ if (B, S) is not a valid BSGS for G.

Thus, given a BSGS (B, S), if we randomly generate some $g \in G$ $w \in \mathbb{N}_+$ times and $\text{STRIP}(g, (B, S))$ returns the identity for all of them, we know that (B, S) is a BSGS for G with probability $p \geq 1 - 2^{-w}$.

The Random Schreier-Sims Algorithm starts by initializing a BSGS as in Algorithm 3.8 (lines 3–8). We then keep generating random $g \in G$ and determine $g', j = \text{STRIP}(g, (B, S))$. If $g' \neq 1$ we extend our BSGS as in Algorithm 3.8 (lines 20–28, with i always equal to 1). If $g' = 1$ w consecutive times, we stop. In certain cases we also know $|G|$ ahead of time, e.g. when G is a wreath product, such that we can keep extending the BSGS until $|U^{(k)}| \cdot |U^{(k-1)}| \cdots |U^{(1)}| = |G|$.

For reasonably large w we can be very sure that the resulting BSGS is a valid BSGS for G. If we want to be *absolutely* sure that this is the case we can:

- Run the Deterministic Schreier-Sims Algorithm on the resulting BSGS.
- Use the Todd-Coxeter-Schreier-Sims or the Sims "Verify" Algorithm to correct the resulting BSGS. While this can be much faster for groups of large degree, these algorithms are beyond the scope of this thesis, refer to [10] Chapter 6.

For a complete pseudocode description of the Random Schreier-Sims Algorithm refer to e.g. [10] Chapter 4.

Chapter 4

Architecture Graphs

We have previously seen in Sections 2.2 and 2.4 how permutation groups and partial permutation inverse monoids can describe symmetries and partial symmetries of undirected graphs respectively. This implies that we can capture the (partial) symmetries inherent in abstract MPSoC architectures by first modelling them as undirected graphs and then determining the automorphism groups and the partial automorphism inverse monoids of these graphs, which we shall term *architecture graphs*.

In this chapter we will first formally define architecture graphs in Section4.1. In Section4.2 we will then examine how we can determine a generating set for the automorphism group of an architecture graph and in Section4.3 we demonstrate how we can decompose the automorphism groups of certain *separable* and *hierarchical* architecture graphs into direct and wreath products respectively. Finally, in Section4.4 we describe a simple but inefficient algorithm for finding a generating set for the partial automorphism inverse monoid of an architecture graph.

4.1 Definition

Transforming abstract MPSoC architectures into undirected graphs is easy: We introduce a vertex for every processing element and an edge for every communication channel. It should now be apparent why we have only considered undirected graphs so far: MPSoC communication channels are usually bi-directional[1]. In addition, we need to take into account the fact that processing elements and communication channels might be heterogeneous. Thus we introduce *vertex* and *edge colors* which assign identical processing elements and communication channels the same unique color (where we simply represent colors by positive integers). More formally we define an architecture graph as follows:

[1]But note that this need not be true in general, we can imagine e.g. one-directional FIFO channels between processing elements or shared memory that can be read by multiple processing elements but only be written to by a subset of them. Nevertheless we will continue assuming that all communication channels are bi-directional to keep notation simple. The following discussion is trivially extensible to directed architecture graphs.

Definition 4.1: Architecture Graph

An architecture graph A is a four-tuple $(P, C, \text{col}_P, \text{col}_C)$ where:

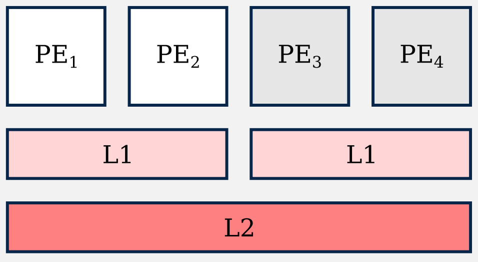
- P is a set of vertex indices of the form $\{1, 2, \ldots, n \mid n \in \mathbb{N}_+\}$.
- $C = \{\{i, j\} \mid i \leq j \wedge$ an edge exists between i and j, $i, j \in P\}$[a].
- $\text{col}_P : P \to \mathbb{N}_+$ maps every $p \in P$ to some color.
- $\text{col}_C : C \to \mathbb{N}_+$ maps every $c \in C$ to some color.

[a]Note that C is a multiset.

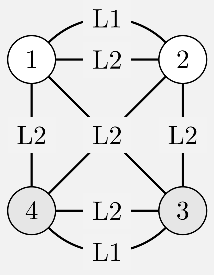
Example 4.1: Architecture Graph

Consider the abstract MPSoC architecture presented in Figure 4.1a. Take note of the fact that unlike before we are now dealing with an MPSoC architecture with two types of processing elements and communication channels respectively. The different processor types are indicated by different shades of gray and the communication channels arise from L1 caches shared between processing elements of the same type and an L2 cache shared by all processing elements. If we omit any loops for the sake of simplicity, the corresponding architecture graph can described as follows, see Figure 4.1b for a graphical representation:

- $P = \{1, 2, 3, 4\}$
- $C = \{\{1, 2\}, \{1, 2\}, \{1, 3\}, \{1, 4\}, \{2, 3\}, \{2, 4\}, \{3, 4\}, \{3, 4\}\}$
- $\text{col}_P(1) = 1,\ \text{col}_P(2) = 1, \text{col}_P(3) = 2,\ \text{col}_P(4) = 2$
- $\text{col}_C(\{1, 2\}) = 1,\ \text{col}_C(\{3, 4\}) = 1,\ \text{col}_C(\{1, 2\}) = 2,\ \text{col}_C(\{1, 3\}) = 2,$
  $\text{col}_C(\{1, 4\}) = 2,\ \text{col}_C(\{2, 3\}) = 2,\ \text{col}_C(\{2, 4\}) = 2,\ \text{col}_C(\{3, 4\}) = 2$



(a)



(b)

Figure 4.1: From abstract MPSoC architecture to automorphism graph.

4.2 Determining Automorphisms

Finding a generating set for the automorphism group of a graph is a well-studied problem in graph theory for which there exist efficient algorithms. The program *nauty/Traces* [12] is able to determine such generating sets for vertex colored graphs. Since automorphism graphs can be totally colored we furthermore need to be able to transform any totally colored graph into a vertex colored graph with the same automorphism group, which is possible via the following theorem[2]:

Theorem 4.1: Reducing Totally Colored Architecture Graphs

Given a totally colored architecture graph $A = (P, C, \mathrm{col}_P, \mathrm{col}_C)$ and letting

$$\begin{aligned}|\mathrm{col}_P| &= \text{the number of distinct vertex colors of A}\\ |\mathrm{col}_C| &= \text{the number of distinct edge colors of A}\end{aligned}$$

we construct a vertex colored architecture graph $A' = (P', C', \mathrm{col}'_P, \mathrm{col}'_C)$ as follows:

- $P' = \{1, 2, \dots, |P| \cdot (\lfloor \log_2(|\mathrm{col}_C|) \rfloor + 1)\}$
- $C' = \{\{i, j\} \mid j = i + |P|,\ i, j \in P'\}$
  $\cup \{\{i, j\} \mid l = \mathrm{lvl}(i) = \mathrm{lvl}(j) \wedge \{\mathrm{p}(i), \mathrm{p}(j)\} \in C \wedge \mathrm{bd}_l(\mathrm{col}_C(\{i, j\})) = 1,\ i, j \in P'\}$
- $\forall p \in P' : \mathrm{col}'_P(p) = \mathrm{col}_P(p) + |\mathrm{col}_P| \cdot \mathrm{lvl}(p)$
- $\forall c \in C' : \mathrm{col}'_C(c) = 1$

where:

$$\begin{aligned}\mathrm{p} &: P' \to P,\ p \mapsto ((p-1) \bmod |P|) + 1\\ \mathrm{lvl} &: P' \to P,\ p \mapsto \left\lfloor \frac{p-1}{|P|} \right\rfloor\\ \mathrm{bd}_i &: \mathbb{N}_+ \to \{0, 1\},\ n \mapsto \text{the ith digit of the binary representation of } n\end{aligned}$$

we have A and A' then have isomorphic automorphism group, a proof of which is beyond the scope of this thesis.

Example 4.2: Reducing Totally Colored Architecture Graphs

Consider again the totally colored architecture graph presented in Example 4.1. A vertex colored architecture graph with the same automorphism group can be described as follows, see Figure 4.2 for a graphical representation:

[2]This construction is based on a description given in the nauty manual accessible at `http://pallini.di.uniroma1.it/Guide.html`.

- $P' = \{1, 2, 3, 4, 5, 6, 7, 8\}$
- $C' = \{\{1, 2\}, \{3, 4\}, \{1, 5\}, \{2, 6\}, \{3, 7\}, \{4, 8\}, \{5, 6\}, \{5, 7\}, \{5, 8\}, \{6, 7\}, \{6, 8\}, \{7, 8\}\}$
- $\mathrm{col}'_P(1) = 1,\ \mathrm{col}'_P(2) = 1,\ \mathrm{col}'_P(3) = 2,\ \mathrm{col}'_P(4) = 2,$
  $\mathrm{col}'_P(5) = 3,\ \mathrm{col}'_P(6) = 3,\ \mathrm{col}'_P(7) = 4,\ \mathrm{col}'_P(8) = 4$
- $\forall c \in C : \mathrm{col}'_C(c) = 1$

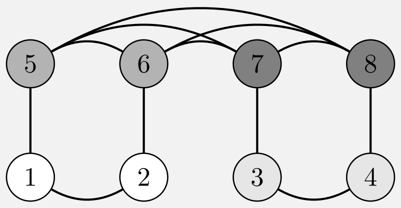


Figure 4.2: Vertex colored automorphism graph.

Thus we can construct the automorphism group of any architecture graph by first transforming the graph into an isomorphic vertex colored graph, finding a generating set for its automorphism group using nauty and then a corresponding BSGS using the algorithms presented in Chapter 3.

4.3 Automorphism Decomposition

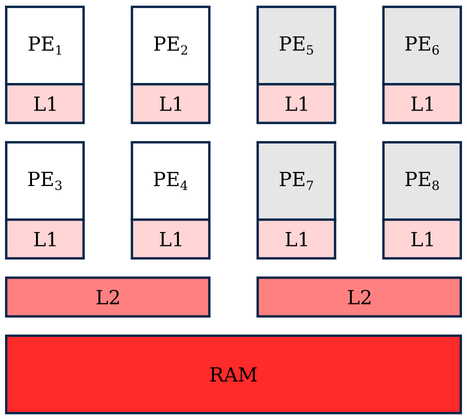


Figure 4.3: Exynos architecture with two different types of processing elements. Each processing element has its own L1 cache and all processing elements of the same type share a separate L2 cache. All processing elements can also communicate via shared RAM.

The architecture graphs we are dealing with in practice are based on real world abstract MPSoC architectures. Many of these are either *separable* or *hierarchical* in nature, properties that we, as it

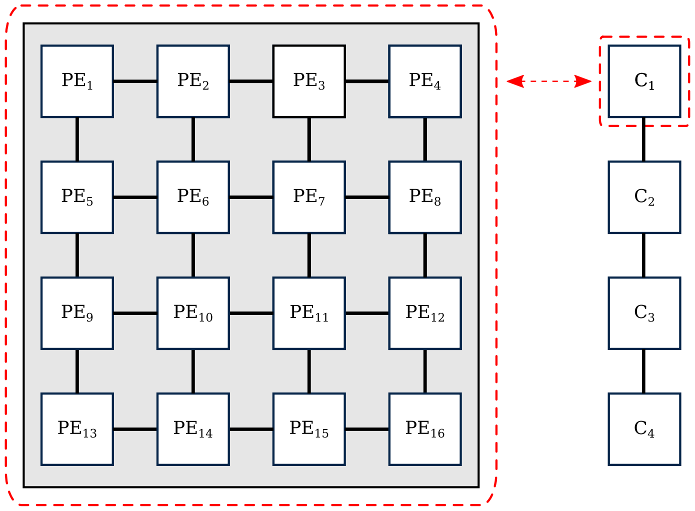


Figure 4.4: HAEC architecture with four separate SoCs, each similar to Parallellas Epiphany coprocessor. The processing elements on one SoC can communicate with each other via optical links and with *all* processing elements on an adjacent SoC via a wireless link.

turns out, can use to decompose the autmorphism groups of the corresponding architecture graphs into direct and wreath products which we introduced in Section2.2.4. This is not just an interesting observation but can also help us solve the task mapping problem more efficiently for these types of architecture graphs as discussed in Section5.4.

Separable architecture graphs are all those made up of "islands" of processing elements for which no automorphism maps from one island to another. This occurs e.g. for abstract MPSoC architectures with several distinct processor types. Consider e.g. the *Exynos* architecture shown in Figure 4.3. Its automorphism group G_{Exynos} is the direct product of the automorphism groups of the subgraphs created by restricting that architecture graph to one of the two processor types, i.e.:

$$\begin{aligned} G_{\text{Exynos}} &= \langle\{(1\ 2),(2\ 3),(2\ 4),(3\ 4)\}\rangle \times \langle\{(5\ 6),(6\ 7),(6\ 8),(7\ 8)\}\rangle \\ &= \langle\{1\ 2),(2\ 3),(2\ 4),(3\ 4),(5\ 6),(6\ 7),(6\ 8),(7\ 8)\}\rangle \end{aligned}$$

Hierarchical architecture graphs on the other hand arise from abstract architecture graphs that are made up of interconnected identical "clusters" of processing elements. Consider e.g. the *HAEC* architecture shown in Figure 4.4. We can model the corresponding architecture graph as a hypergraph according to Definition 2.14. Its automorphism group G_{HAEC} is then the wreath product of one cluster's automorphism group and the autormophism group of the "cluster graph" itself, i.e.:

$$\begin{aligned} G_{\text{HAEC,proto}} = \langle\{&(1,4)(2,3)(5,8)(6,7)(9,12)(10,11)(13,16)(14,15), \\ &(2,5)(3,9)(4,13)(7,10)(8,14)(12,15)\}\rangle \\ G_{\text{HAEC,super}} = \langle\{&(1\ 4)(2\ 3)\}\rangle \end{aligned}$$

and:

$$G_{\text{HAEC}} = G_{\text{HAEC,proto}} \wr G_{\text{HAEC,super}}$$

We can either manually specify whether a given architecture graph is separable or hierarchical or automatically detect this property by using the algorithms presented in [4] which are able to deterministically discover possible direct and wreath product decompositions for any given permutation group. These algorithms are implemented in `mpsym` but their inner working is beyond the scope of this thesis.

4.4 Determining Partial Automorphisms

Unfortunately, there seems to be no existing research on efficient algorithms for determining a generating set for a graphs partial automorphism inverse monoid.

Determining whether a given partial permutation is a partial automorphism of an architecture graph is trivial. However, the total number of possible partial permutations of $\{1, \ldots, |P|\}$ is $\sum_{i=0}^{|P|} \frac{|P|!}{(|P|-i)!} = \left(\sum_{i=0}^{|P|} \frac{1}{i!}\right)(|P|+1)! \approx e \cdot (|P|+1)!$ and thus generally very large. To avoid searching through all of them to find a suitable generating set we can make use of the fact that if a partial permutation is not contained in a partial permutation inverse monoid then neither are any of its *extensions* (or their extensions and so on), where we define extensions as follows:

Definition 4.2: Extensions of a Partial Permutation

Given a partial permutation s of a set Ω, an extension of s is another partial permutation t with $\operatorname{dom}(t) = \operatorname{dom}(s) \cup \{x\}$ for an $x \in \Omega \setminus \operatorname{dom}(s)$ and $\operatorname{im}(t) = \operatorname{im}(s) \cup \{x\}$ for an $x \in \Omega \setminus \operatorname{im}(s)$. We refer to all extensions of s (of which there are $|\operatorname{dom}(s)|^2$) by $\operatorname{ext}(s)$ and to the set resulting from a recursive application of ext as $\operatorname{ext}^*(s)$.

Example 4.3: Extensions of a Partial Permutation

Let $\Omega = \Omega_4$ and $s = (1,2)$, i.e. $\operatorname{dom}(s) = \operatorname{im}(s) = \{1,2\}$, then $\operatorname{ext}(s) = \{(1,2)(3), (1,2)(4), (1,2)[3,4], (1,2)[4,3]\}$.

So formally, it holds that if s is not a partial automorphism of an architecture graph then neither are any of the partial permutations in $\operatorname{ext}^*(s)$.

Algorithm 4.9 is based on this. If we invoke this recursive algorithm with $s = 0$ and M initially only containing 0 and $1_{\{1,\ldots,|P|\}}$, then the algorithm will traverse a search tree of all possible partial permutations s of $\{1, \ldots, |P|\}$ in depth first fashion and adjoin any $s \notin M$ that is a partial automorphism of the architecture graph A to M[3]. Because the nodes of any subtree of this search tree with root s are exactly $\operatorname{ext}^*(s)$, we can completely prune such a subtree if s is not a partial automorphism of A.

[3]This of course assumes that we can efficiently modify our representation of M in a suitable way such as to adjoin s to M, refer to [5] to see that this is indeed possible.

Algorithm 4.9 Find partial automorphism inverse monoid generating set

```
1: procedure PARTIAL_AUTOM_GENERATORS($A$, $s$, $M$)
2:     if $s$ is a partial automorphism of $A$ then
3:         Adjoin $s$ to $M$
4:     else
5:         return
6:     end if
7:
8:     for Every extension $s'$ of $s$ do
9:         PARTIAL_AUTOM_GENERATORS($A$, $s'$, $M$)
10:    end for
11: end procedure
```

Unfortunately, preliminary experiments have shown that in practice, Algorithm 4.9 is too slow for all but the smallest architecture graphs. Developing a more efficient algorithm could be the subject of further research.

Chapter 5

The TMOR Problem

This chapter concerns itself with the problem at the heart of this thesis: the TMOR problem introduced in Section 1.3. In Section 5.1 we will first formalize the TMOR problem in light of the theoretical foundations laid out in Chapter 2 and then describe some algorithms that address it in Section 5.2. Finally, Section 5.4 will introduce some important optimizations for certain types of separable and hierarchical architecture graphs. We will not consider partial symmetries in this chapter, for one because most of what is discussed here[1]applies to them equally and for another because, as we have seen previously, the algorithms we use for determining and representing them are not quite efficient enough to be useful in practice.

5.1 Definition

In Chapter 4 we have seen how to model abstract MPSoC architectures as undirected graphs and how to describe their symmetries and partial symmetries by finding the automorphism groups and partial automorphism inverse monoids of these undirected graphs respectively. In a similar manner we will now formalize what we mean when we say "task mapping" and relate this concept to that of architecture graphs.

Definition 5.1: Task Mapping

Given an architecture graph $A = (P, C, \mathrm{col}_P, \mathrm{col}_C)$, a k-task mapping is a sequence $T_A^k = (t_1, t_2, \dots, t_k), t_i \in P, \forall 1 \leq i \leq k$. Instead of t_i we also write $T_A^k[i]$.

[1]Except optimization by decomposition.

Example 5.1: Task Mapping

Consider again the task mappings presented in Figure 1.2, and assume that for the given abstract MPSoC architecture we construct the architecture graph presented in Figure 5.1. Then the task mappings (a)-(d) are described by:

- (a) $\rightarrow (1, 5, 9, 13)$
- (b) $\rightarrow (1, 9, 5, 13)$
- (c) $\rightarrow (1, 2, 3, 4)$
- (d) $\rightarrow (2, 6, 10, 14)$

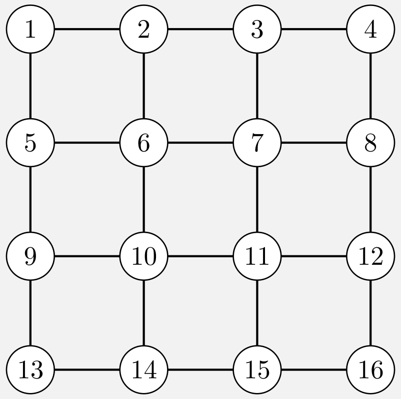


Figure 5.1: Architecture graph corresponding to abstract MPSoC architecture presented in Figure 1.2.

In general, we will not always be interested in *every* possible task mapping for a given architecture graph. For example, we might not be interested in task mappings which map two or more tasks to the same processing element. To formalize this notion, we introduce the concept of *task mapping spaces*:

Definition 5.2: Task Mapping Space

Given an architecture graph $A = (P, C, \mathrm{col}_P, \mathrm{col}_C)$, we denote the space of all admissible k-task mappings on A by Θ_A^k and note that $\Theta_A^k \subseteq P^k$.

Example 5.2: Task Mapping Space

The k-task mapping space for the architecture graph from Example 5.1 which only admits task mappings for which no two tasks share the same processing element is $\Theta_A^k \subset$

$\{1, 2, \ldots, 16\}^k$ such that $\forall T_A^k \in \Theta_A^k : i \neq j \implies T_A^k[i] \neq T_A^k[j]$.

Remember that we are interested in finding task mappings that are equivalent by symmetry. We can find such task mappings via the automorphisms of A represented as permutations as follows:

Definition 5.3: Task Mapping Permutation

Given an architecture graph $A = (P, C, \mathrm{col}_P, \mathrm{col}_C)$, a k-task mapping $T_A^k = (t_1, t_2, \ldots, t_k) \in \Theta_A^k$ and a permutation g over P, we define $g : \Theta_A^k \to \Theta_A^k$ with:

$$g(T_A^k) = \begin{cases} (g(t_1), g(t_2), \ldots, g(t_k)), & \text{if } (g(t_1), g(t_2), \ldots, g(t_k)) \in \Theta_A^k \\ \text{undefined}, & \text{otherwise} \end{cases}$$

From now on we assume that if G is the automorphism group of A, then $\forall T_A^k \in \Theta_A^k : \forall g \in G : g(T_A^k) \in \Theta_A^k$ (alternatively: we assume that Θ_A^k is a G-set) such that the following discussions about task mapping orbits make sense.

Example 5.3: Task Mapping Permutation

As we shall see in Example 5.5, the permutation $g = (1\ 4)(2\ 3)(5\ 8)(6\ 7)(9\ 12)(10\ 11)(13\ 16)(14\ 15)$ is an automorphism of the architecture graph from Example 5.1 and it holds e.g. that $g((1, 5, 9, 13)) = (4, 8, 12, 16)$.

If we seek to determine *all* task mappings equivalent by symmetry to a given task mapping we can again make use of the concept of group orbits introduced in Definition 2.10. By introducing a strict ordering on any task mapping space we can furthermore find a canonical representative for any such orbit, analogous to Definition 2.11. This is everything we need to formalize the TMOR problem: In theory, given an architecture graphs automorphism group, we can partition any task mapping space into orbits and then reduce these orbits to their canonical representatives.

Definition 5.4: Task Mapping Ordering

On any k-task mapping space Θ_A^k we define a total strict ordering $< \subseteq \Theta_A^k \times \Theta_A^k$ as follows: Let $T_A^k = (t_1, t_2, \ldots, t_k), \widetilde{T}_A^k = (\widetilde{t}_1, \widetilde{t}_1, \ldots, \widetilde{t}_k) \in \Theta_A^k$ then:

$$(T_A^k, \widetilde{T}_A^k) \in < \Leftrightarrow \exists 1 \leq i \leq k : (\forall 1 \leq j < i : t_j = \widetilde{t}_j) \wedge (t_i < \widetilde{t}_i)$$

This type of ordering is called *lexicographical*. We also write $T_A^k < \widetilde{T}_A^k$.

Example 5.4: Task Mapping Ordering

For the task mappings from Example 5.1 it holds that:

$$(1, 2, 3, 4) < (1, 5, 9, 13) < (1, 9, 5, 13) < (2, 6, 10, 14)$$

Definition 5.5: Task Mapping Orbit

Given a permutation group G and a task mapping space Θ_A^k, we define the orbit of some task mapping $T_A^k \in \Theta_A^k$ as $T_A^{k,G} = [T_A^k]_{\sim G} = \{g(T_A^k) \mid g \in G\}$. We also define $\mathrm{repr}(T_A^{k,G})$ analogous to Definition 2.11, with $<$ as in Definition 5.4. When G is the automorphism group of A we simply write $\mathrm{repr}(T_A^k)$.

Example 5.5: Task Mapping Orbits

Consider again the architecture graph A from Example 5.1 and the task mapping space Θ_A^k from Example 5.2. We can now formally determine which task mappings are equivalent by symmetry to e.g. $T_{A,1}^4 = (1, 5, 9, 13)$. This is analogous to determining $T_{A,1}^{4,G}$ where G is the automorphism group of A. It is well known that the automorphism group G of any $n \times n$ regular mesh architecture graph A is ismorphic to the *dihedral group* D_8 with $|D_8| = 8$. It holds that:

$$\begin{aligned} G = \langle \{&(1\ 4)(2\ 3)(5\ 8)(6\ 7)(9\ 12)(10\ 11)(13\ 16)(14\ 15), \\ &(2\ 5)(3\ 9)(4\ 13)(7\ 10)(8\ 14)(12\ 15)\} \rangle \end{aligned}$$

which we can interpret geometrically as reflections about the meshes upper left to lower right diagonal and its vertical center respectively. As can be readily verified, we have:

$$\begin{aligned} T_{A,1}^{k,G} = \{&(1, 2, 3, 4), (1, 5, 9, 13), (4, 3, 2, 1), (4, 8, 12, 16), \\ &(13, 14, 15, 16), (13, 9, 5, 1), (16, 12, 8, 4), (16, 15, 14, 13)\} \end{aligned}$$

And $\mathrm{repr}(T_{A,1}^k) = (1, 2, 3, 4)$. We can determine how many task mapping orbits there are in total by making use of Lemma 2.2.3: For all $g \in G$, let $\Theta_A^{k,g} = \{T_A^k \in \Theta_A^k \mid g(T_A^k) = T_A^k\}$ and note that:

$$|\Theta_A^{k,g}| = \begin{cases} \frac{|P^g|!}{(|P^g|-k)!}, & \text{if } |P^g| \geq k \\ 0, & \text{otherwise} \end{cases}$$

as should be obvious from the definition of Θ_A^k. We have:

$$\begin{aligned}
&|\Theta_A^{k,()}| = {}^{16!}/_{12!} = 43680 \\
&|\Theta_A^{k,(1\ 13)(2\ 14)(3\ 15)(4\ 16)(5\ 9)(6\ 10)(7\ 11)(8\ 12)}| = 0 \\
&|\Theta_A^{k,(1\ 13\ 16\ 4)(2\ 9\ 15\ 8)(3\ 5\ 14\ 12)(6\ 10\ 11\ 7)}| = 0 \\
&|\Theta_A^{k,(1\ 16)(2\ 12)(3\ 8)(5\ 15)(6\ 11)(9\ 14)}| = {}^{4!}/_{0!} = 24 \\
&|\Theta_A^{k,(1\ 16)(2\ 15)(3\ 14)(4\ 13)(5\ 12)(6\ 11)(7\ 10)(8\ 9)}| = 0 \\
&|\Theta_A^{k,(1\ 4)(2\ 3)(5\ 8)(6\ 7)(9\ 12)(10\ 11)(13\ 16)(14\ 15)}| = 0 \\
&|\Theta_A^{k,(1\ 4\ 16\ 13)(2\ 8\ 15\ 9)(3\ 12\ 14\ 5)(6\ 7\ 11\ 10)}| = 0 \\
&|\Theta_A^{k,(2\ 5)(3\ 9)(4\ 13)(7\ 10)(8\ 14)(12\ 15)}| = {}^{4!}/_{0!} = 24
\end{aligned}$$

By averaging these set sizes we obtain $\frac{1}{8}(43680+24+24) = 5466$. Since $|\Theta_A^k| = \frac{16!}{14!} = 43680$, if we could successfully determine representatives for every orbit of G, we would effectively reduce the size of the task mapping search space by a factor of $\frac{43680}{5466} \approx 8$.

5.2 Algorithmic Approaches

Algorithm 5.10 Obtain orbit identifier.

```
1: procedure ORBIT_IDENTIFIER($T_A^k$)
   ▷ reprs retains its state across invocations.
2:
3:     if $A$ has changed since last invocation then
4:         reprs ← ()
5:     end if
6:
7:     if repr($T_A^k$) $\notin$ reprs then
8:         Append repr($T_A^k$) to reprs
9:     end if
10:
11:    return index of repr($T_A^k$) in reprs
12: end procedure
```

It now remains to discuss how we can partition a task mapping space in practice. Because $|\Theta_A^k|$ might be very large, in general it is impractical to explicitly determine all orbits for a given automorphism group. An iterative approach is more feasible: We consider a series of task mappings in turn and for each of them determine the canonical representative of the orbit it lies in. In this way, for every new task mapping we can discern whether we have already encountered a task mapping in the same orbit earlier (and can thus e.g. skip some expensive simulation step for this new task mapping) by matching canonical representatives. Algorithm 5.10 is a formalization of this. It takes as its argument a task mapping and returns an integer uniquely identifying the orbit in which this task mapping lies. The central importance of orbit representatives to this approach is why introduced the term TMOR.

Algorithm 5.10 is particularly interesting because it lends itself to be used in conjunction with e.g. a genetic algorithm which generates new "interesting" task mappings based on the simulated characteristics of old ones. We can then simply discard those task mappings that are equivalent by symmetry to, i.e. lie in the same orbit as, previously generated ones.

Algorithm 5.11 Determine canonical representatives via orbit enumeration.

```
procedure TMOR_ORBIT($T_A^k$, $G = \langle S \rangle$)
    ▷ $G$ is the automorphism group of $A$.

    orbit $\leftarrow \{\}$
    while orbit is changing do
        for $\widetilde{T}_A^k \in$ orbit, $g$ in $S$ do
            orbit $\leftarrow$ orbit $\cup \{g(\widetilde{T}_A^k)\}$
        end for
    end while

    return min(orbit)
end procedure
```

Algorithm 5.12 Determine canonical representatives via group enumeration.

```
procedure TMOR_ITERATE($T_A^k$, $G$)
    ▷ $G$ is the automorphism group of $A$.

    repr($T_A^k$) $\leftarrow T_A^k$
    for $g \in G$ do
        if $g(T_A^k) <$ repr($T_A^k$) then
            repr($T_A^k$) $\leftarrow g(T_A^k)$
        end if
    end for

    return repr($T_A^k$)
end procedure
```

In order to implement Algorithm 5.10, we need to be able to efficiently determine repr(T_A^k). If we desire guaranteed correctness there are two basic approaches which both completely enumerate $T_A^{k,G}$: We either enumerate the orbit explicitly via the fixed point Algorithm 5.11 or by iterating over the elements of G as in Algorithm 5.12. We will also refer to these algorithms as *bruteforce orbit enumeration* and *bruteforce iteration* respectively.

If we (as is usually the case) want to find canonical representatives for a large number of task

Algorithm 5.13 Determine canonical representatives via local search.

```
1: procedure TMOR_LOCAL_SEARCH($T_A^k$, $G = \langle S \rangle$)
   ▷ $G$ is the automorphism group of $A$.
2:
3:     repr($T_A^k$) ← $T_K^k$
4:     while repr($T_A^k$) is changing do
5:         repr($T_A^k$) ← min($\{g(\text{repr}(T_A^k)) \mid g \in S\}$)
6:     end while
7:
8:     return repr($T_A^k$)
9: end procedure
```

mappings, we can potentially improve performance in both cases by hashing already discovered canonical representatives and aborting the task mapping orbit discovery when a known canonical representative is encountered. We could also choose to additionally hash a number of random task mappings per orbit in an attempt to abort the enumeration earlier.

It is hard to say in general if bruteforce orbit enumeration or iteration is more efficient for a given automorphism group. Overall, bruteforce iteration seems to be most suitable if $|G|$ is small and bruteforce orbit enumeration if Θ_A^k is partitioned into many small orbits by G. Unfortunately, this latter characteristic is not easily determinable without actually computing the complete orbit partition. We shall analyse this matter more thorougly in Section 5.3.

If both methods of orbit enumeration prove to be impractically slow, we can employ Algorithm 5.13 which computes an *approximate* canonical representative via local search. In effect, this algorithm performs breadth first search within an orbit. The returned canonical representative is thus not guaranteed to be the correct one which might be acceptable depending on the application and likelihood that this will occur. Note that it might be sensible to append additional elements to the generating set S, e.g. $\{g^{-1} \mid g \in S\}$ or random elements of G, in order to "widen" the search tree and/or to employ e.g. simulated annealing or similar heuristic techniques instead of breadth first search, refer to e.g.[16]. These potential improvements are however beyond the scope of this thesis.

5.3 Complexity and Accuracy Considerations

The time complexity of bruteforce iteration is clearly $O(|G|)$. Furthermore, the execution time of bruteforce iteration should not vary substantially with k, assuming that iterating through G is more computationally expensive than finding $g(T_A^k)$.

At worst, bruteforce orbit iteration only adds a single element to $T_A^{k,G}$ in every iteration and its runtime complexity is thus $O(|S|(1 + 2 + \cdots + (|T_A^{k,G}| - 1))) = O(|S||T_A^{k,G}|^2)$. If Θ_A^k is partitioned into relatively few but large orbits, finding representatives via orbit enumeration is potentially prohibitively expensive.

As we have demonstrated in Example 2.12, determining the number of task mapping orbits for a given automorphism group G is possible algorithmically in $O(|G|)$ time by making use of Lemma 2.2.3. Since these orbits partition Θ_A^k, we also know that their sizes must sum to $|\Theta_A^k|$ and we can thus easily determine the *average* orbit size. However, the total execution time of the bruteforce orbit enumeration algorithm largely depends on the size of the *largest* orbit or orbits[2]. There are two reasons for this:

- For a given task mapping T_A^k the runtime complexity of the bruteforce orbit enumeration algorithm grows superlinearly with $|T_A^{k,G}|$ as described above.
- For two orbits, one significantly larger than the other, the probability of some T_A^k randomly drawn from a uniform distribution over Θ_A^k lying in the larger orbit is greater than the probability of it lying in the smaller orbit.

Unfortunately, it is very difficult to determine a tight upper bound for the size of any given orbit for even moderately large values of k. We can however trivially state that any orbit can at most be of size $|G|$, irrespective of $|\Theta_A^k|$. This is because every element in an orbit is "reachable" from every other element in the same orbit via some $g \in G$, i.e. $\forall T_A^k \in \Theta_A^k : \forall T_{A,1}^k, T_{A,2}^k \in T_A^{k,G} : \exists g \in G : g(T_{A,1}^k) = T_{A,2}^k$[3]. This also implies that $O(\langle S \rangle\, |T_A^{k,G}|^2) = O(\langle S \rangle\, |G|^2)$.

It remains to ask whether we expect the average orbit size to increase with k. A simple observation shows that this is almost certainly the case: Assuming again that no two tasks can be mapped to the same processing element it trivially holds that $|\Theta_A^{k+1}| = (|P| - k) \cdot |\Theta_A^k|$. Thinking back to Example 2.12, for the number of orbits of Θ_A^k we have:

$$|\Theta_A^k / {\sim_G}| = \frac{1}{|G|} \sum_{g \in G} |\Theta_A^{k,g}|$$

with:

$$|\Theta_A^{k,g}| = \begin{cases} \frac{|P^g|!}{(|P^g|-k)!}, & \text{if } |P^g| \geq k \\ 0, & \text{otherwise} \end{cases}$$

Since $\forall g \in G : |P^g| \leq |P|$ we thus have $|\Theta_A^{k+1,g}| \leq (|P| - k) \cdot |\Theta_A^{k,g}|$. And thus for the average orbit size it holds that:

$$\frac{|\Theta_A^{k+1}|}{|\Theta_A^{k+1} / {\sim_G}|} \geq \frac{|\Theta_A^k|}{|\Theta_A^k / {\sim_G}|}$$

i.e. the average orbit size cannot decrease with k. Thus we can reasonably expect bruteforce orbit enumeration to perform worse for larger values of k.

The runtime complexity of the local search algorithm obviously mainly depends on the size of the chosen generating set for G. On the other hand, a larger generating set may sometimes make finding correct representatives more likely, i.e. it can potentially be sensible to trade off execution time for increased accuracy by adding redundant elements to G's generating set during local search as described previously.

[2]This does not hold if G partitions Θ_A^k into a large number of very small orbits and relatively few large ones whose combined size is still small compared to $|\Theta_A^k|$. We shall not analyse when or if this occurs in detail.

[3]It can additionally be shown that the size of every orbit must *divide* $|G|$. In theory we could use this fact to find one or several possibilities for the orbit size distribution by looking for an integer partition of $|\Theta_A^k|$ which has $|\Theta_A^k / {\sim_G}|$ summands $\in \{s \in \mathbb{N}_+ \mid s \mid |G|\}$. However, because the number of orbits is typically very large even for moderately large values of k, known integer partition algorithms cannot cope with this task, refer to e.g. [21].

In general, we expect local search to execute faster than bruteforce iteration or orbit enumeration, especially for large values of $|G|$ such that our main concern with this algorithm is accuracy. A trivial observation we can make here is that, as with the execution time of bruteforce orbit enumeration, this metric should mainly depend on the orbit size distribution. This is because during local search we never "leave" a task mapping's orbit, such that smaller orbits make it more likely that the representative returned is the correct one. Note that there may be pathological cases where local search never or almost never produces correct representatives for a given automorphism group and generating set but investigating whether and when this happens is beyond the scope of this thesis.

In conclusion, the performance of bruteforce iteration and orbit enumeration as well as local search depends directly or indirectly (via an upper bound on the orbit size) on $|G|$. Only bruteforce enumeration is likely to be significantly impacted by k.

5.4 Optimization via Decomposition

As we discussed in Section4.3, we can decompose the automorphism groups of certain separable and hierarchical architecture graphs into direct and wreath products. We can make use of this to potentially decrease the execution time of the algorithms presented in the previous section. This idea was first presented in [4] in the context of model checking, see also [3].

Essentially, decomposition allows us to solve the task mapping problem separately for smaller permutation groups than the full automorphism group. Algorithms 5.14 and 5.15 demonstrate how this works for the case of direct and wreath product decomposition respectively. For a definition of $\sigma(G_{\text{proto}}) = \{\sigma_i(G_{\text{proto}}) \mid 1 \leq i \leq \deg(G_{\text{super}})\}$ and $\sigma(G_{\text{super}})$ and proof of correctness of these algorithms refer to [4]. Note that it is possible to employ these algorithms recursively, i.e. we could decompose an automorphism group into a direct or wreath product and further decompose the components of this product into direct or wreath products themselves.

Algorithm 5.14 Determine canonical representatives for separable architecture graphs.

```
1: procedure TMOR_DIRECT_PROD($T^k_A$, $G_1, \ldots, G_n$)
   ▷ $G_1 \times \cdots \times G_n$ is the automorphism group of $A$.
   ▷ TMOR is any canonical representative algorithm.
2:
3:     repr($T^k_A$) ← TMOR($T^k_A, G_1$)
4:
5:     for $i = 2 \ldots n$ do
6:         repr($T^k_A$) ← TMOR(repr($T^k_A$), $G_i$)
7:     end for
8:
9:     return repr($T^k_A$)
10: end procedure
```

We can especially expect hierarchical decomposition to result in significant speedups because

Algorithm 5.15 Determine canonical representatives for hierarchical architecture graphs.

```
 1: procedure TMOR_WREATH_PROD($T_A^k$, $G_{\text{proto}}, \ldots, G_{\text{super}}$)
    ▷ $G_{\text{proto}} \wr G_{\text{super}}$ is the automorphism group of $A$.
    ▷ $|\sigma_i(G_{\text{proto}})| = |G_{\text{proto}}|, \forall 1 \leq i \leq \deg(G_{\text{super}})$ and $|\sigma(G_{\text{super}})| = |G_{\text{super}}|$.
    ▷ TMOR is any canonical representative algorithm.
 2:
 3:     $\text{repr}(T_A^k) \leftarrow T_A^k$
 4:
 5:     for $i = 1 \ldots \deg(G_{\text{super}})$ do
 6:         $\text{repr}(T_A^k) \leftarrow \text{TMOR}(\text{repr}(T_A^k), \sigma_i(G_{\text{proto}}))$
 7:     end for
 8:
 9:     return $\text{TMOR}(\text{repr}(T_A^k), \sigma(G_{\text{super}}))$
10: end procedure
```

the combined order of two permutation groups G_{proto} and G_{super}[4] making up a wreath product $G = G_{\text{proto}} \wr G_{\text{super}}$ is usually much smaller than the order of the wreath product itself.

[4]And thus of all $\sigma_i(G_{\text{proto}})$ and $\sigma(G_{\text{super}})$.

Chapter 6

Experimental Results

This chapter describes experimental results collected during the evaluation of the most important algorithms presented in previous chapters. We implement these algorithms both in the computer algebra system/scripting language GAP [8] and in `mpsym`. All experiments were performed on a single machine with two Intel® Core™ i5-6200U CPU cores and 4 GiB DDR3-SDRAM.

Section 6.1 first describes the abstract MPSoC architectures for which we perform our experiments. In Section6.2 we then examine how fast we can construct a BSGS for an architecture's automorphism group. Finally, in Section6.3 we compare the performance of the different TMOR algorithms outlined in Section5.2 for large sets of randomly generated k-task mappings, for different values of k.

Again, we do not consider partial automorphisms for practical reasons as indicated in the introduction of Chapter 5.

6.1 Architectures

We perform our experiements for four examplary abstract MPSoC architectures based on real-world MPSoCs. We briefly list and visualize these architectures here but do not explicitly provide architecture graphs for brevity's sake. In the following sections we denote the architecture graph corresponding to an architecture X by A_X and the automorphism group of A_X by G_X.

- The *Exynos*[1] MPSoC developed by Samsung, with *Octa Big-Little* multi core configuration. We have already encountered this architecture in Section 4.3, see Figure 4.3.
- A 16 by 16 processing element regular mesh based on the *Epiphany* [14] coprocessor of the *Parallella*[2] board. See Figure 6.1.
- The *HAEC* [6] architecture, consisting of several optical-link-based MPSoCs connected to each other by wireless links. We have already encountered this architecture in Section 4.3, see Figure 4.4.

[1]https://www.samsung.com/semiconductor/minisite/exynos/
[2]https://www.parallella.org/

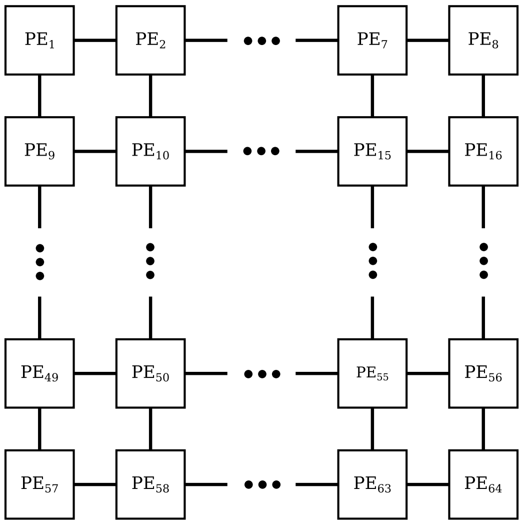


Figure 6.1: Parallela's Epiphany coprocessor architecture with 64 processing elements connected in a regular mesh fashion.

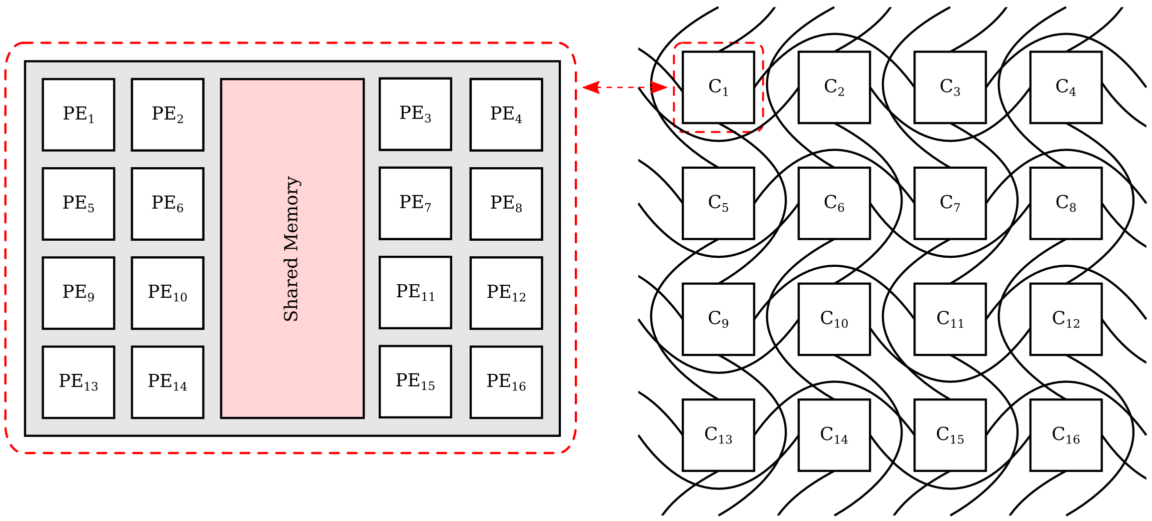


Figure 6.2: Kalray MMPA-256 architecture made up of 16 identical clusters of 16 processing elements fully connected via shared memory. The clusters themselves are connected in a 2D torus topology.

- The *Kalray MMPA-256*[3] architecture made up of 16 identical compute clusters. See Figure 6.2.

For reference we also list the orders of these four architectures' automorphism groups here:

- $|G_{\text{Exynos}}| = 576$.
- $|G_{\text{Parallella}}| = 8$.
- $|G_{\text{HAEC}}| = 8192$ and $G_{\text{HAEC}} = G_{\text{HAEC,proto}} \wr G_{\text{HAEC,super}}$ with $|G_{\text{HAEC,proto}}| = 8$ and $|G_{\text{HAEC,super}}| = 2$
- $|G_{\text{Kalray}}| \approx 1.079 \cdot 10^{214}$ and $G_{\text{Kalray}} = G_{\text{Kalray,proto}} \wr G_{\text{Kalray,super}}$ with $|G_{\text{Kalray,proto}}| = 16!$ and $|G_{\text{Kalray,super}}| = 8$

6.2 Automorphism Group Construction

6.2.1 Methodology

We first analyse how quickly we can construct a BSGS for a given architectures automorphism group. This is a prerequisite to solving the TMOR problem and we will from now on use the term *setup time* to refer to the overhead incurred by this. Constructing a BSGS for an architecture graph A_X requires us to perform the following steps:

- 1. Construct a graph data structure representing A_X.
- 2. From this data structure, find a generating set for G_X.
- 3. From this generating set, construct a BSGS for G_X.

For Step 1 and 2, our GAP implementation makes use of the *grape* [19] package and `mpsym` makes use of the *Boost Graph Library* [18] and *nauty* [12][4], a program which is able to efficiently determine generating sets for the automorphism groups of vertex colored graphs[5]

Step 3 is where the Schreier-Sims Algorithm comes into play. GAP does not directly expose its internal implementations of the different BSGS construction algorithms. Instead, it automatically chooses which algorithm variant or variants to apply for any given generating set based on complex heuristics. `mpsym` currently only implements the "basic" Deterministic Schreier-Sims Algorithm and the Random Schreier-Sims Algorithm. Since Step 3 is usually significantly more computationally expensive than Step 1 and 2, we only present total execution time data. We compare the following algorithm variants:

- BSGS construction using GAP.

[3]https://www.kalrayinc.com/

[4]grape also uses nauty internally.

[5]Remember that we showed in Section 4.2 how to convert any totally colored undirected graph to an isomorphic vertex colored one.

- BSGS construction using `mpsym`. We employ both the Deterministic Schreier-Sims Algorithm on its own and the Random followed by the Deterministic Schreier-Sims Algorithm (to guarantee correctness).

  We also employ the Random Schreier-Sims Algorithm on its own. Note, however, that we do this with the sole purpose of demonstrating how the execution times of a single run of the Deterministic and Random Schreier-Sims Algorithms compare. We do not attempt to draw a direct comparison between the two algorithms because the Random Schreier-Sims Algorithm it is not guaranteed to produce a correct BSGS. Of course we could simply choose w to be suitably large as to make correctness of the constructed BSGS very likely. But because `mpsym` currently does not implement any of the algorithms outlined in Section 3.3 for detecting/guaranteeing correctness of a BSGS, we have no simple way of checking whether a BSGS constructed this way is correct. Thus, we cannot empirically verify the relationship between w and the likelihood of a correctly constructed BSGS[6] for any given generating set. For this reason, we omit this information altogether.

6.2.2 Results

Figure 6.3 visualizes the BSGS construction execution times for three of our four architectures using both our GAP and `mpsym` implementations. We can make the following observations:

- For these three architectures `mpsym` outperforms GAP significantly.
- For the HAEC architecture, the difference in execution time between the Random and Deterministic Schreier-Sims Algorithm, as implemented in `mpsym`, is most pronounced. Nevertheless, combining both algorithms does not result in a speedup over the deterministic algorithm alone.

We have purposefully omitted the Kalray architecture from Figure 6.3 because deterministically constructing a BSGS for G_{Kalray} is computationally very expensive. While GAP is able to so after several seconds, `mpsym` is unable to even after several minutes of execution time. However, we can potentially do much better in two ways:

- (1) *Avoiding BSGS construction*: Using Algorithm 5.15 we can solve the TMOR problem without the need for a BSGS for $G_{\text{Kalray,proto}} \wr G_{\text{Kalray,super}}$. Instead, we construct BSGSs for $\sigma(G_{\text{Kalray,super}})$ and $\sigma(G_{\text{Kalray,proto}})$[7].
- (2) *Symmetric group detection*: There exists an efficient and reliable Monte-Carlo algorithm for testing whether a given generating set generates a symmetric group, refer to e.g. [10] Chapter 3, such that it is possible to explicitly construct a BSGS for S_{16} without the need to run the Schreier-Sims algorithm, even when we are initially unaware that $G_{\text{Kalray,proto}}$ is a symmetric group.

[6]Unless we verify correctness via complete group enumeration which is obviously impractical for large automorphism groups.

[7]This obviously also applies to other hierarchical architectures, e.g. HAEC. But since BSGS construction for G_{HAEC} is already very fast we only present a comparison for the Kalray architecture.

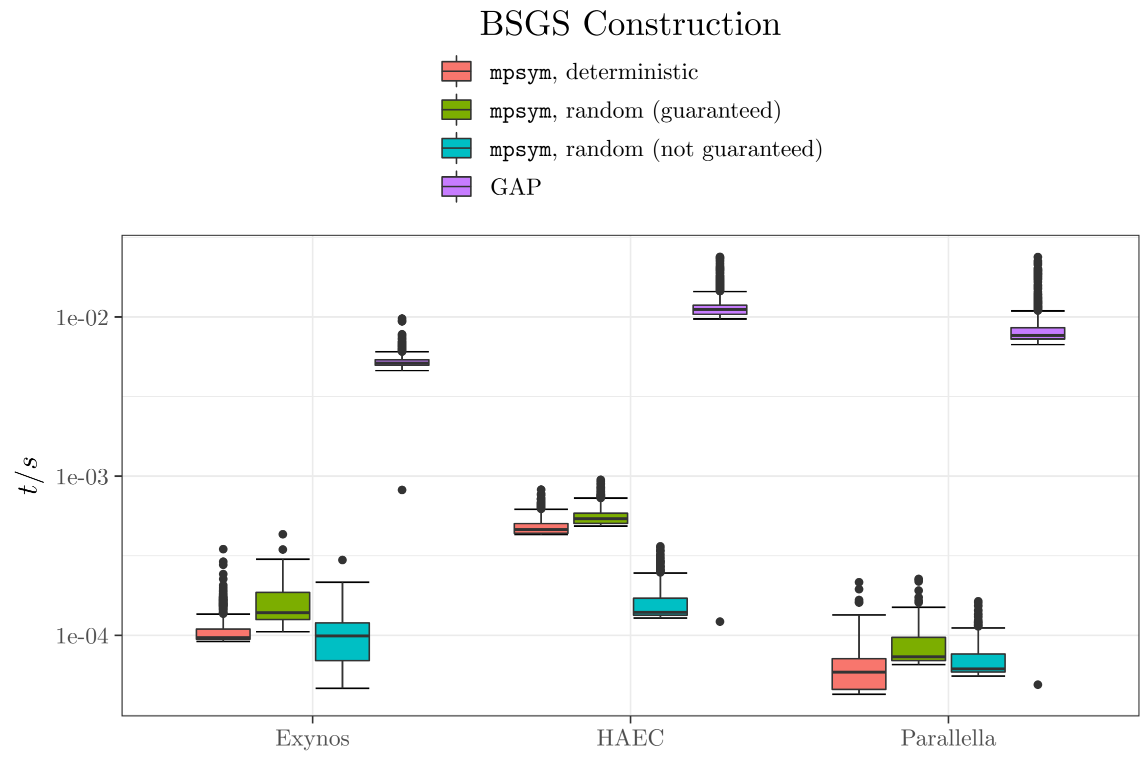


Figure 6.3: BSGS construction execution times for the Exynos, HAEC and Parallella architectures. The value $w = 10$ was chosen for `mpsym`s implementation of the Random Schreier-Sims Algorithm. Boxplots were generated from 1000 runs each.

Table 6.1 compares the total setup time when using GAP versus when using `mpsym` in combination with the methods described above. Clearly, using method (1) is much less computationally expensive than constructing a BSGS for G_{Kalray}. Of course taking this approach implies that we must subsequently use Algorithm 5.15 to solve the TMOR problem. In Section 6.3 we will analyse whether this results in further speedup. Using method (2) results in an additional speedup of about 25%.

Method	Execution Time
GAP	$15.31 \pm 0.05s$
`mpsym`, (1)	$3.31 \cdot 10^{-1} \pm 0.01 \cdot 10^{-1}$
`mpsym`, (1) + (2)	$2.43 \cdot 10^{-1} \pm 0.07 \cdot 10^{-1}$

Table 6.1: BSGS construction execution times for the Kalray board. `mpsym` makes use of the Deterministic Schreier-Sims Algorithm. Means and standard deviations obtained from 10 runs each and rounded to nearest two/three decimal places.

6.3 Solving the TMOR problem

6.3.1 Methodology

We now analyse how quickly we are able to solve the TMOR problem for each of our four architectures. To this end we generate 10,000 suitable random k-task mappings per architecture by uniformly sampling from Θ_A^k from Example 5.2 (for $k \in \{4, 8, 12, 16\}$) and then determine how long it takes to find orbit representatives for all of them using each of the following algorithms:

- Algorithm 5.12, i.e. bruteforce iteration, implemented in both GAP and `mpsym`.
- Algorithm 5.11 , i.e. bruteforce orbit enumeration, implemented in both GAP and `mpsym`.
- Algorithm 5.13, i.e. local search, implemented in both GAP and `mpsym`. Since the taks mapping representatives determined via local search are not guaranteed to be correct, we also note the success rate of this algorithm, i.e. the ratio of correctly determined orbit representatives to the number of task mappings. For the sake of brevity we do not experiment with any of the possible augmentations to this algorithm discussed in Section5.2.
- Algorithm 5.15 , i.e. hierarchical decomposition, implemented in `mpsym`. We perform decomposition explicitly for the HAEC and Kalray architectures. We use this algorithm in combination with both bruteforce iteration and local search and note any execution time and/or accuracy improvements.

  Due to time constraints, we did not implement hierarchical decomposition in GAP and consequently only present data obtained by use of our `mpsym` implementations for the HAEC and Kalray architectures.

6.3.2 Results

We now present numerical results and analyse if they are consistent with the considerations we made in Section5.3. For all figures in this section, presented execution times are means of ten independent runs and local search results are annotated with the achieved accuracy.

Exynos

Starting with the results for the Exynos architecture presented in Figure 6.4 we can already make several important observations:

- Bruteforce orbit enumeration is faster than bruteforce iteration for the chosen values of k but its performance, as expected, depends more strongly on k.
- Our `mpsym` implementations of bruteforce orbit enumeration and iteration are more than an order of magnitude faster than the corresponding GAP implementation.
- Local search, as implemented in both GAP and `mpsym`, is significantly faster than bruteforce orbit enumeration and iteration and returns the correct representative for all 10,000 task mappings.

Parallella

The results for the Parallella architecture presented in Figure 6.5 exhibit similar trends. Here however, bruteforce iteration is faster than orbit enumeration for the chosen values of k which is most likely due to the very small order of Parallellas automorphism group. Nevertheless, local search is still faster than bruteforce iteration and achieves perfect accuracy as well.

HAEC

For the HAEC architecture, for which results are presented in Figure 6.6, the performance of bruteforce orbit enumeration again strongly depends on k. Figure 6.7 hints at why this is the case. It visualizes how the size of more and more orbits approaches or equals $|G_{\text{HAEC}}|$ as k grows[8].

Furthermore we can observe that hierarchically decomposing HAECs automorphism group results in a speedup of several orders of magnitude when used in combination with bruteforce orbit enumeration or iteration and a jump from mediocre to perfect accuracy when used in combination with local search.

Kalray

The benefits of hierarchical decomposition are made even more clear by the results for the Kalray architecture presented in Figure 6.8. Here, without architecture graph both bruteforce orbit enumeration and iteration, implemented in both GAP and `mpsym`, are not able to find a representative for **a single** task mapping even after executing for several minutes and local search is not able to accurately determine even a single representative. Finding representatives for thousands of task mappings without hierarchical decomposition is thus completely infeasible.

[8]Note that all orbit sizes divide $|G_{\text{HAEC}}|$.

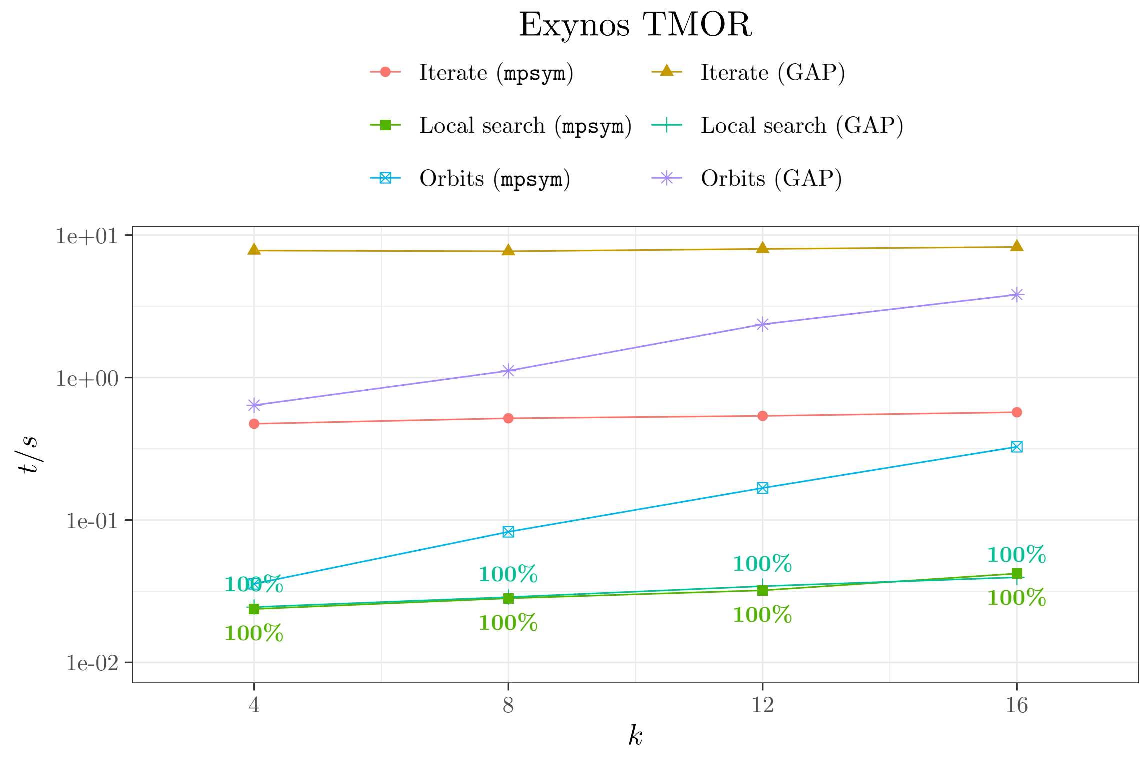


Figure 6.4: TMOR results for the Exynos architecture.

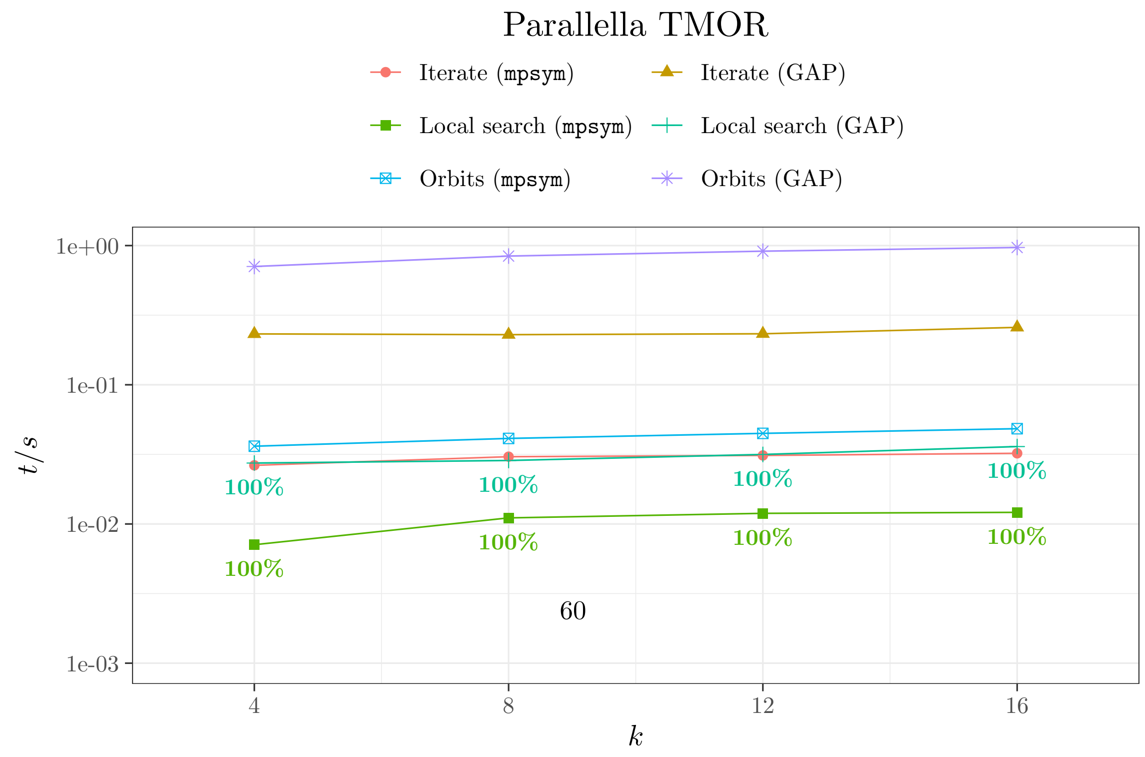


Figure 6.5: TMOR results for the Parallella architecture.

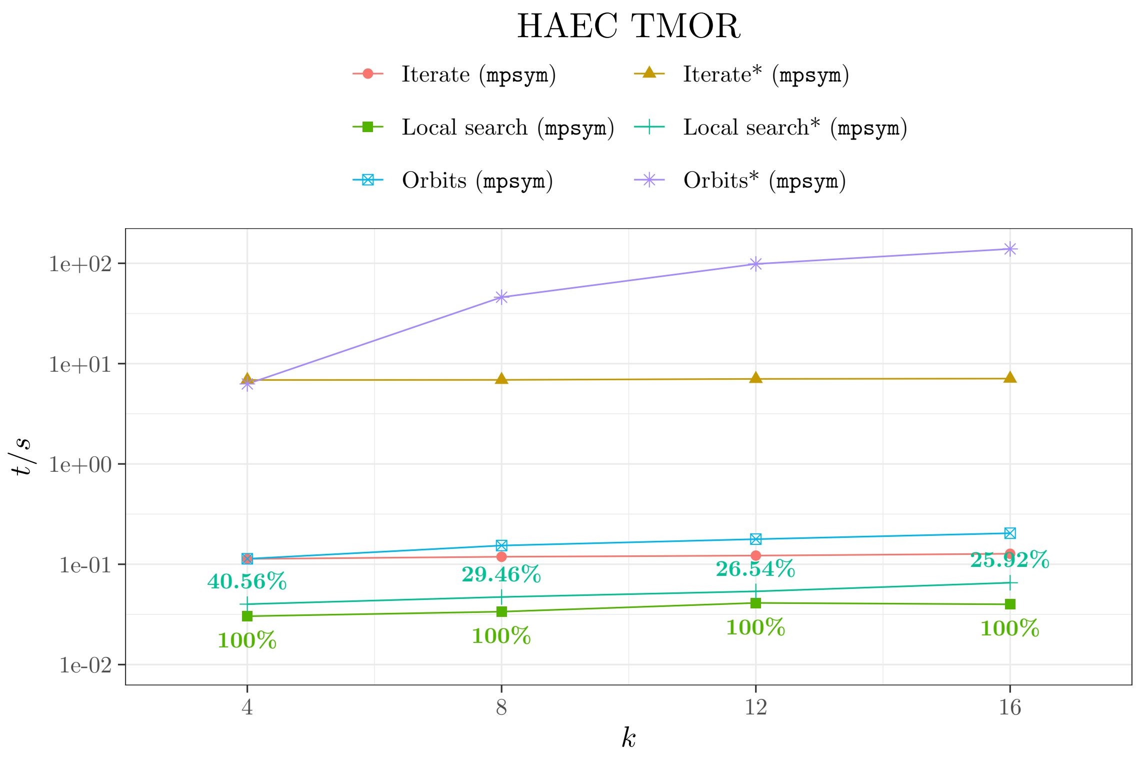


Figure 6.6: TMOR results for the HAEC architecture. Data series marked with "*" do **not** make use of hierarchical decomposition.

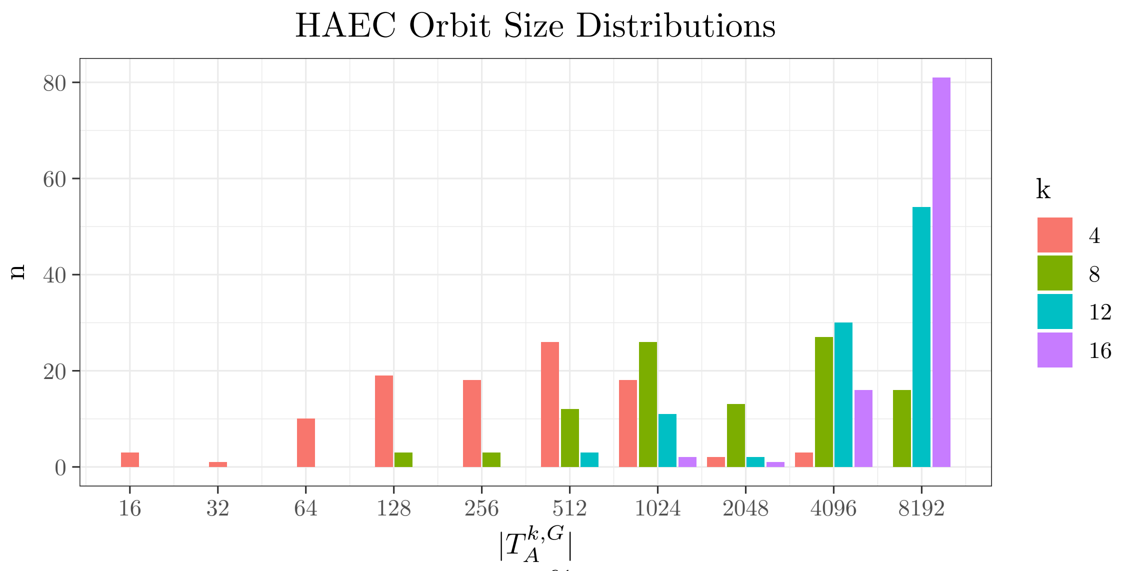


Figure 6.7: Orbit size distribution for 100 random task mappings for the HAEC architecture, for different values of k. n refers to the number of occurences of orbits of the corresponding size.

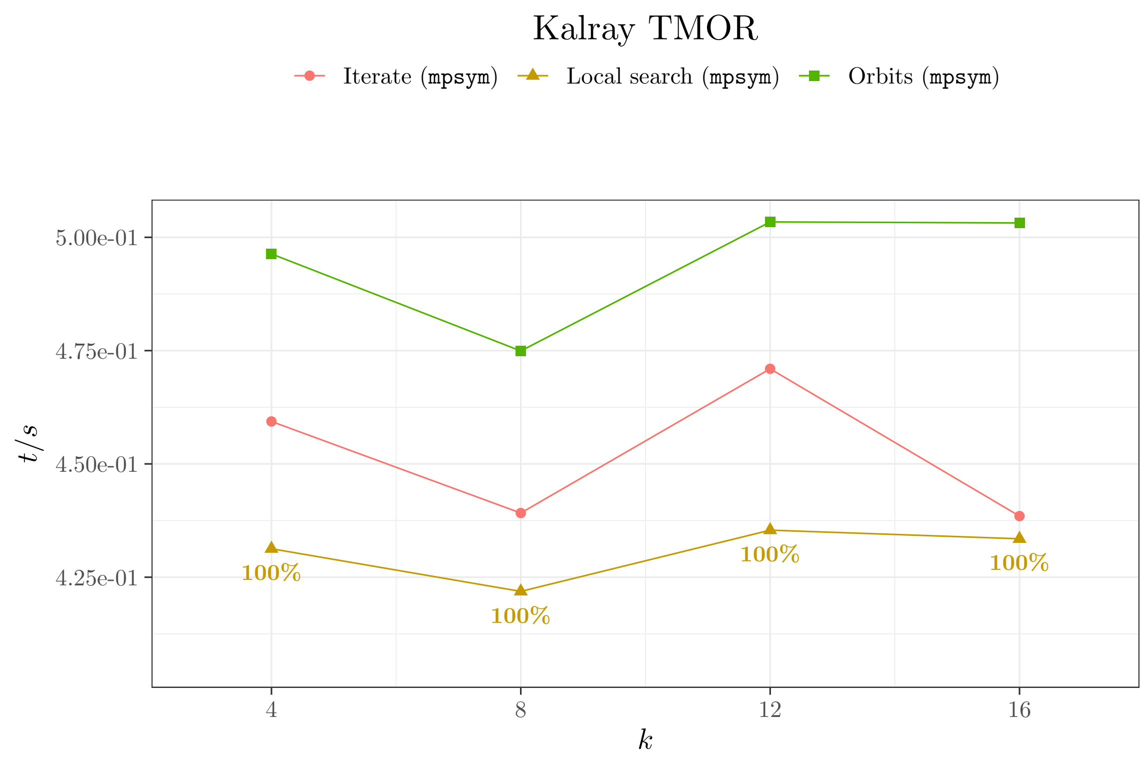


Figure 6.8: Results for the Kalray architecture. All data series make use of hierarchical decomposition and transitive group optimization.

With architecture graph decomposition alone, the situation still looks dire: Each of the clusters making up the Kalray architecture has the automorphism group $G_{\text{Kalray,proto}} = S_{16}$ and similarly, all $\sigma_i(G_{\text{Kalray,proto}}) \in \sigma(G_{\text{Kalray,proto}})$ are isomorphic to S_{16}. Because $|S_{16}| = 16!$, albeit much smaller than the complete automorphism group, is still very large, bruteforce orbit enumeration and iteration slow and local search remains unacceptably inaccurate. This is however easily remedied because symmetric permutation groups are transitive which allows us to trivially solve the TMOR problem for all $\sigma_i(G_{\text{Kalray,proto}}) \in \sigma(G_{\text{Kalray,proto}})$ [9]. Since in this case $|G_{\text{Kalray,super}}|$ is small, the overall TMOR problem becomes feasible.

Figure 6.8 visualizes results obtained using both architecture graph decomposition and the aforementioned transitive group optimization. Finding correct representatives for all 10,000 task mappings takes less than a second in total in all cases and local search achieves perfect accuracy as well.

[9]Since there is only one orbit, the representative is the same for every possible task mapping.

Chapter 7

Conclusion

In this thesis we have demonstrated how we can simplify MPSoC task mapping by exploiting symmetries inherent in common MPSoC architectures. In doing so we have explored some fundamental computational group theory concepts, data structures and algorithms. We have seen how to describe abstract MPSoC architecture as graphs and how to extract a representation of their (partial) symmetries from this description. We have also introduced and formalized the TMOR problem and devised several algorithms to adress it. Experiments on real world MPSoC architectures have shown that these algorithms can work well in practice. However, they are as of now not suitable for use with partial symmetries due to shortcomings in the way we determine and represent partial automorphism inverse monoids.

As we have seen in Chapter 6, the overhead incurred by symmetry reduction is potentially very small, making it a sensible preprocessing step for most other task mapping algorithms. We have also seen that `mpsym` is able to outperform GAP when it comes to solving the TMOR problem. As such, development of `mpsym` has paid off.

However, due to the high complexity of many of the algorithms underlying computational group theory, maintaining and extending `mpsym` is not possible without significant mathematical domain knowledge, especially when correctness and efficiency of these algorithms is of high importance. One alternative approach might thus be compiling GAP code to C in the spirit of Cython [1] and so directly exposing its interface to a fast compiled language such as C++.

As far as the current state of `mpsym` goes, it is mostly complete in regard to symmetries but lacking in regard to partial symmetries.

For the former, it might be desireable to further improve BSGS construction via careful analysis of the success rate of the Random Schreier-Sims Algorithm as well as implementations of the Todd-Coxeter Schreier-Sims and Sims' "Verify" Algorithm mentioned in Section 3.3. In solving the TMOR problem, it could furthermore be fruitful to investigate advanced local search methods as outlined in Section 5.2 and to speed up orbit enumeration beyond the simple fixed point algorithm currently employed by `mpym`.

For the latter, a first important step would be faster discovery of a generating set for the partial automorphism inverse monoid of an architecture graph, improving over the relatively naive search tree pruning algorithm presented in Section 4.4 which does not scale to architectures with even a moderately large number of processing elements. Additionally, for deterministically solving the TMOR problem we would also require a fast way of iterating through an inverse monoid or an altogether different approach whose complexity remains manageable even for larger sets of tasks.

Bibliography

[1] Behnel, S., Bradshaw, R., Citro, C., Dalcin, L., Seljebotn, D. S., and Smith, K. (2011). Cython: The best of both worlds. *Computing in Science & Engineering*, 13(2):31–39.

[2] Butler, G. (1991). *Fundamental Algorithms for Permutation Groups*. Springer Berlin Heidelberg.

[3] Donaldson, A. F. (2007). *Automatic techniques for detecting and exploiting symmetry in model checking*.

[4] Donaldson, A. F. and Miller, A. (2009). On the constructive orbit problem. *Ann Math Atrif Intell*, 57:1–35.

[5] East, J., Egri-Nagy, A., Mitchell, J., and Péresse, Y. (2019). Computing finite semigroups. *Journal of Symbolic Computation*, 92:110–155.

[6] Fettweis, G., Dörpinghaus, M., Castrillon, J., Kumar, A., Baier, C., Bock, K., Ellinger, F., Fery, A., Fitzek, F. H. P., Härtig, H., Jamshidi, K., Kissinger, T., Lehner, W., Mertig, M., Nagel, W. E., Nguyen, G. T., Plettemeier, D., Schröter, M., and Strufe, T. (2019). Architecture and advanced electronics pathways toward highly adaptive energy-efficient computing. *Proceedings of the IEEE*, 107(1):204–231.

[7] Goens, A., Siccha, S., and Castrillon, J. (2017). Symmetry in software synthesis. *ACM Trans. Archit. Code Optim.*, 14(2).

[8] Group, T. G. (2020). *GAP – Groups, Algorithms, and Programming, Version 4.11.0.*

[9] Hemminger, R. L. (1968). The group of an x-join of graphs. *Journal of Combinatorial Theory*, 5(4):408 – 418.

[10] Holt, D. F. (2005). *Handbook of Computational Group Theory*. CRC Press.

[11] Lawson, M. V. (1998). *Inverse Semigroups: The Theory of Partial Symmetries*. World Scientific.

[12] McKay, B. D. and Piperno, A. (2014). Practical graph isomorphism, ii. *Journal of Symbolic Computation*, 60:94 – 112.

[13] Nicolai, T. (2020). mpsym. `https://github.com/Time0o/mpsym`.

[14] Olofsson, A. (2016). Epiphany-v: A 1024 processor 64-bit RISC system-on-chip.

[15] Rotman, J. J. (1995). *An Introduction to the Theory of Groups*. Springer New York, 4 edition.

[16] Russell, S. and Norvig, P. (2020). *Artificial Intelligence: A Modern Approach*. Prentice Hall Press, 4 edition.

[17] Seress, Á. (2009). *Permutation Group Algorithms*. Cambridge University Press.

[18] Siek, J., Lee, L.-Q., and Lumsdaine, A. (2000). Boost graph library. `http://www.boost.org/libs/graph/`.

[19] Soicher, L. H. (2019). The grape package for gap, version 4.8.3. `https://gap-packages.github.io/grape`.

[20] Tarjan, R. E. (1972). Depth-first search and linear graph algorithms. *SIAM J. Comput.*, 1:146–160.

[21] Zoghbi, A. and Stojmenovic, I. (1994). Fast algorithms for generating integer partitions. *International Journal of Computer Mathematics*, 70(2):319–332.

Appendix A

mpsym Code Samples

Architecture graph descriptions are passed to `mpsym` in the form of *Lua* scripts. For example, Listing A.1 shows the Lua code used to describe the HAEC architecture.

Listing A.1 haec.lua

```
cal mpsym = require 'mpsym'

local sg_clusters = mpsym.identical_clusters(4, 'SoC')
local sg_channels = mpsym.linear_channels(sg_clusters, 'wireless')

local prot_processors = mpsym.identical_processors(16, 'P')
local prot_channels = mpsym.grid_channels(prot_processors, 'optical')

return mpsym.ArchUniformSuperGraph:create{
  super_graph = mpsym.ArchGraph:create{
    clusters = sg_clusters,
    channels = sg_channels
  },
  proto = mpsym.ArchGraph:create{
    processors = prot_processors,
    channels = prot_channels
  }
}
```

Finding task mapping representatives in C++ is then possible as demonstrated in Listing A.2.

Listing A.2 haec_map_tasks.cpp

```
auto arch_graph(mpsym::ArchGraphSystem::from_lua("haec.lua"));

std::vector<mpsym::TaskMapping> task_mappings{
  {1, 45, 35, 17, 58},
  {2, 46, 22, 19, 20},
  {3, 11, 8,  14, 55},
  {4, 38, 43, 16, 47}
  // ...
};

mpsym::TaskOrbits task_orbits;
for (auto const &task_mapping : task_mappings)
  arch_graph->repr(task_mapping, &task_orbits);

for (auto const &repr : task_orbits)
  std::cout << "Found task orbit representative: " << repr << std::endl;
```