

Bachelor's Thesis

submitted in partial fulfillment of the requirements
for the course “Applied Computer Science”

Accelerating Irregular Parallel Algorithms on SpiNNaker2 Hardware

Jake (Pseudonym)

Institute of Computer Science

Bachelor's and Master's Theses
at Georg-August-Universität Göttingen
ISSN 1612-6793

2026-August-11

Georg-August-Universität Göttingen
Institute of Computer Science

Goldschmidtstraße 7
37077 Göttingen
Germany



+49 (551) 39-172000



+49 (551) 39-14403



office@informatik.uni-goettingen.de



www.informatik.uni-goettingen.de

First Supervisor: Herr Prof. Dr. Kunkel (julian.kunkel@gwdg.de)
Second Supervisor: Herr Decker (jonathan.decker@uni-goettingen.de)

I hereby declare that I have written this thesis independently without any help from others and without the use of documents or aids other than those stated. I have mentioned all used sources and cited them correctly according to established academic citation rules.

In this work I have used LLMs as follows:

- Writing assistance. Converting bullet points and information from papers using RAG into paragraphs. Rephrasing sentences. Summarizing results.
- Brainstorming of research topics.
- Explanation of topics related to this thesis.
- Summarizing papers.
- Extracting information from documentation and papers using RAG.
- Brainstorming of methods for data visualization.
- Software Development and code generation, especially for repetitive and well known software patterns and plotting of results.
- Description of Figures.
- Code debugging.
- Code documentation.
- Spell & grammar checking. Text Reviewer.

I hereby declare that I have stated all uses completely.

Missing or incorrect information will be considered as an attempt to cheat.

Göttingen, 2026-August-11

Abstract

Modern accelerators such as GPUs achieve high performance primarily on regular, data-parallel workloads, but perform poorly on irregular parallel algorithms characterized by data-dependent control flow, unpredictable memory access, and dynamic communication patterns. SpiNNaker2 is a many-core chip developed for energy-efficient neuromorphic computing that provides 152 independently programmable ARM Cortex-M4F processing elements connected by a packet-switched Network-on-Chip. Its Multiple Instruction, Multiple Data (MIMD) organization is architecturally well suited to irregular workloads, yet its software ecosystem is oriented almost exclusively toward spiking neural networks, and its applicability to general-purpose, non-neural algorithms has not been evaluated empirically. This thesis investigates whether SpiNNaker2 can be used as a general-purpose accelerator for irregular parallel computation, using the Ant Colony System (ACS) applied to the Traveling Salesman Problem (TSP) as a representative case study.

To enable this investigation, we develop `s2generic`, a host- and chip-side software layer that exposes SpiNNaker2’s interrupt, remote-memory, and synchronization facilities to arbitrary C applications, addressing gaps in the existing tooling that assumes a neural-network workload. Building on this layer, we design a memory-efficient representation of ACS state that fits entirely within the limited local memory of a processing element: city coordinates replace a quadratic distance matrix, tours are stored as one-byte city identifiers, and the symmetric pheromone matrix is encoded as a triangular array of 16-bit fixed-point values. This representation supports Euclidean TSP instances of up to 256 cities without requiring access to slower, NoC-attached external memory during tour construction. We then implement ACS as a multi-colony island model, first across the processing elements of a single chip using direct broadcast of improved tours, and subsequently across a 48-chip board using a supervisor-based hierarchy that limits host-chip communication to one core per chip, compensating for the absence of working direct inter-chip routing in our deployment. The implementation is benchmarked on TSPLIB instances ranging from 51 to 226 cities, using a shared ACS parameter configuration, and compared against a single-threaded x86 reference implementation.

The results show that the developed memory representation successfully keeps all performance-critical ACS state in local SRAM for every evaluated instance size. Increasing the number of independently searching colonies, from a single worker to a full chip and finally to the entire board, consistently reduces the variance of final tour lengths and improves typical solution quality, with the full-board configuration producing the strongest and most consistent results and discovering competitive tours early during execution. These findings confirm that SpiNNaker2’s distributed-memory MIMD architecture can effectively support large-scale parallel stochastic search when communication is restricted to small, infrequent messages rather than shared global state. At the same time, iteration-count-based measurements and the available board-level power data are insufficient to establish wall-clock speedup or energy efficiency relative to the x86 baseline. We conclude that SpiNNaker2 is a promising platform for irregular parallel algorithms whose state can be partitioned across autonomous processing elements, provided that data representation and communication are carefully co-designed with the hardware’s memory and networking constraints, and we identify controlled runtime and energy measurements, working inter-chip routing, and broader algorithmic evaluation as key directions for future work.

Contents

1	Introduction	1
1.1	Motivation	1
1.2	Goals	2
1.3	Contributions	3
2	Background	3
2.1	Flynn’s Taxonomy	4
2.2	SpiNNaker2 Hardware	6
2.2.1	Overview	6
2.2.2	ARM Cortex-M4F Processing Element (PE)	7
2.2.3	Network-on-Chip (NoC)	8
2.2.4	Regfile	9
2.2.5	SpiNNaker2 Router & 48-Node Board	9
2.2.6	SNN Application Development	10
2.3	Irregular Parallel Algorithms	10
2.4	Ant Colony Optimization & Traveling Salesman Problem	11
2.4.1	The Traveling Salesman Problem	11
2.4.2	The Ant System (AS) Algorithm for TSP	12
2.4.3	Ant Colony System (ACS)	12
2.4.4	Multi-Colony Island Model	13
2.4.5	Irregularity in ACO, ACS, and Multi-Colony Models	14
3	Related Works	16
3.1	GPU-Based Parallelization of ACO	16
3.2	FPGA-Based Acceleration of ACO	17
3.3	SpiNNaker2 Beyond Neuromorphic Workloads	17
3.4	Gap Addressed by This Thesis	17
4	Methodology	18
4.1	Design of the s2generic SDK and Library	18
4.1.1	Accessing Chip-Side Variables from Host	20
4.1.2	Regfile Interrupts	22
4.1.3	Feedthrough Interrupts	23
4.1.4	Accessing Variables in SRAM of other PEs	25
4.1.5	Semaphores	25
4.1.6	Enabled Cores	27
4.1.7	Chip Side Logging and Assertions	28
4.1.8	Inter-Chip Communication	28
4.2	Implementing ACO on SpiNNaker2	28
4.2.1	Representation of TSP Instances	29
4.2.2	Memory-Efficient ACS State	31
4.2.3	Single-PE Implementation	35
4.2.4	Single-Chip Multi-Colony Island Model Implementation	36
4.2.5	Scaling Multi-Colony Island Model Implementation on 48-Node Board	37

5	Benchmarking	39
5.1	TSP Instances	39
5.2	SpiNNaker2 Experimental Setup	40
5.3	Selection of ACS Parameters	41
5.4	Common SpiNNaker2 Benchmarks	41
5.5	Single-Threaded x86 Reference	42
6	Evaluation	43
6.1	Selected Parameter Configuration	43
6.2	Scaling with TSP Dimension	45
6.3	Consistency of Solution Quality	47
6.4	Consistency of Termination Behavior	49
6.5	Development of Solution Quality	51
6.6	Summary of Experimental Results	53
7	Discussion	53
7.1	Suitability of SpiNNaker2 for Irregular Search	53
7.2	Effectiveness of the Memory Representation	54
7.3	Parallelization and Communication Strategy	54
7.4	Interpretation of Solution-Quality Improvements	55
7.5	Performance and Energy Claims	56
7.6	Correctness and Reproducibility	56
7.7	Threats to Validity	57
8	Conclusions	58
8.1	Future Work	59
A	Bibliography	61
B	Source Code	64

1 Introduction

This chapter motivates the use of SpiNNaker2 for irregular parallel optimization algorithms (Section 1.1), states the goals pursued by the thesis (Section 1.2), and summarizes its concrete contributions (Section 1.3).

1.1 Motivation

Modern computing systems derive much of their performance from parallelism, but the effectiveness of a given parallel architecture depends strongly on the structure of the workload it executes. Throughput-oriented accelerators such as GPUs perform particularly well when many processing units can apply the same operation to regularly arranged data. [1] Algorithms with data-dependent control flow, unpredictable memory accesses, dynamic load imbalance, and irregular communication are considerably more difficult to execute efficiently on such hardware. Branch divergence and poorly coalesced memory accesses can leave substantial parts of these accelerators underutilized. [1]

Many important optimization and graph algorithms exhibit precisely these characteristics. Ant Colony Optimization (ACO), for example, constructs solutions through independently acting agents whose decisions depend on random choices and on information accumulated during the search. Different agents therefore follow different paths, access different parts of the problem state, and may need to communicate at unpredictable times. These properties complicate acceleration on lockstep architectures, but they expose a form of parallelism that is well suited to systems composed of independently executing processing elements rather than to systems that require every unit to perform the same operation at the same time.

SpiNNaker2 provides such an architecture. Although it was developed primarily for event-based neuromorphic workloads, each SpiNNaker2 chip contains 152 independently programmable ARM Cortex-M4F processing elements connected by a packet-switched Network-on-Chip (NoC) [2]. Its Multiple Instruction, Multiple Data (MIMD) organization allows processing elements to follow different control-flow paths without incurring a branch-divergence penalty, while its event-driven communication mechanisms support asynchronous interaction between them. SpiNNaker2 is therefore a promising platform not only for neural simulation, but potentially also for general-purpose irregular parallel algorithms whose structure conflicts with the regularity assumed by SIMD accelerators.

Using SpiNNaker2 outside its intended application domain nevertheless presents significant challenges. Each processing element has only a small amount of local memory, accesses to non-local memory must traverse the NoC, and programmers must explicitly manage task placement, communication, synchronization, and interaction with the host system. The available software ecosystem is oriented almost exclusively toward spiking neural networks and offers limited support and documentation for applications that are not themselves neural models. Evaluating SpiNNaker2 as an accelerator for irregular optimization therefore requires two things at once: an algorithmic implementation appropriate for the hardware’s memory and communication constraints, and a practical software foundation on which that implementation can be developed in the first place.

This thesis investigates these challenges through the Ant Colony System (ACS), a widely used variant of ACO, applied to the Traveling Salesman Problem (TSP). ACS is a suitable case study because its probabilistic tour construction, data-dependent memory accesses, and population

of independently operating search agents expose both the strengths and the limitations of a distributed-memory MIMD architecture. At the same time, its pheromone matrix and its frequent distance queries place substantial pressure on the limited local memory available to each processing element. The resulting implementation therefore provides concrete insight into whether SpiNNaker2 can serve as an accelerator for irregular algorithms beyond its established neuromorphic use cases.

Throughout this thesis, “accelerating” an irregular algorithm on SpiNNaker2 is understood primarily in terms of exploiting its many independent processing elements to widen and speed up the parallel search itself, that is, to explore more of the solution space concurrently and to reach good solutions in fewer algorithmic steps, rather than as a guarantee of a lower wall-clock runtime or lower energy consumption than an existing CPU or GPU implementation. Whether such a wall-clock or energy advantage also materializes is treated as an open, empirical question rather than as an assumed outcome, and it is addressed separately in the evaluation.

1.2 Goals

The primary goal of this thesis is to determine how an irregular parallel optimization algorithm can be implemented and accelerated on SpiNNaker2 hardware. SpiNNaker2 is treated throughout as a collection of general-purpose MIMD processing elements rather than as a dedicated neural-network accelerator. The thesis accordingly examines the programming, memory, communication, and scaling considerations that arise from this use of the architecture.

The thesis pursues the following goals:

1. **Establish a usable environment for general-purpose applications.** The work should provide a host- and chip-side software layer for loading applications, selecting processing elements, exchanging data with individual cores, handling interrupts, and using the hardware’s synchronization facilities. This environment should build upon the existing SpiNNaker2 software ecosystem while exposing the functionality required by applications that are not spiking neural networks.
2. **Design an ACS implementation compatible with SpiNNaker2 hardware.** The TSP instance, pheromone trails, agent state, intermediate values, and application stack should fit within the local memory of a single processing element. The implementation should therefore avoid storing a complete distance matrix, use a compact tour representation, and reduce the storage required by the pheromone matrix, keeping frequently accessed state in local SRAM rather than in external SDRAM.
3. **Exploit parallelism across processing elements and chips.** The implementation should support multiple independently executing ACS colonies and extend them through a multi-colony island model. Separate colonies should search concurrently and periodically exchange their best-known tours. This design should map naturally onto SpiNNaker2’s distributed-memory MIMD organization and its packet-based communication mechanisms.
4. **Evaluate performance, scalability, and solution quality.** The implementation should be benchmarked on standard TSP instances and compared against a single-threaded x86 baseline using equivalent algorithmic parameters. The evaluation should consider execution time, tour quality, and scaling across processing elements, and, where the experimental environment permits, it should also examine power and energy consumption. Any claim of

wall-clock or energy acceleration relative to the x86 baseline is treated as a hypothesis to be tested by this evaluation, not as a premise of the implementation work.

The goal of this work is not to demonstrate that SpiNNaker2 is universally preferable to CPUs or GPUs for this class of problem, nor is a favorable wall-clock or energy outcome presupposed. Instead, the thesis aims to identify the trade-offs involved in using the architecture for an irregular, non-neural workload, and to determine which hardware and software characteristics benefit or limit this particular use case, and to report honestly on which forms of “acceleration” - algorithmic, in terms of parallel search breadth and solution quality, or physical, in terms of runtime and energy - are actually supported by the evidence gathered.

1.3 Contributions

This thesis makes the following contributions:

1. It develops **s2generic**, a software environment for running general-purpose C applications on SpiNNaker2. The environment provides host-side application deployment and core selection, typed access to chip-side variables, and abstractions for hardware facilities including Regfile interrupts, Feedthrough interrupts, remote SRAM access, and hardware semaphores.
2. It presents a **memory-efficient representation of ACS state** for SpiNNaker2 processing elements. Euclidean TSP instances are stored as city coordinates rather than as quadratic distance matrices, tours use one-byte city identifiers, and symmetric pheromone trails are represented as a triangular array of 16-bit fixed-point values. Together, these design choices allow the algorithm’s performance-critical state to remain in local SRAM for instances of up to 256 cities.
3. It implements **ACS as a general-purpose irregular workload** on SpiNNaker2 and extends it to a multi-colony island model. Independent colonies execute concurrently on different processing elements and exchange compact best-known tours, matching the asynchronous, distributed character of the underlying hardware.
4. It provides an **experimental evaluation against an x86 implementation** using established TSP benchmark instances. The evaluation examines the practical effects of the memory representation, parallel execution, intercore communication, and system scale on runtime and solution quality, and explicitly distinguishes the resulting gains in parallel search effectiveness from the separate, and more limited, evidence available for wall-clock and energy acceleration.

Together, these contributions provide both a concrete case study of irregular optimization on SpiNNaker2 and reusable infrastructure for investigating other general-purpose algorithms on the platform.

2 Background

This chapter provides the background required for the remainder of the thesis. [Section 2.1](#) introduces Flynn’s taxonomy, which is used throughout the thesis to distinguish SIMD from MIMD execution. [Section 2.2](#) describes the SpiNNaker2 hardware on which the implementation runs, including its processing elements, Network-on-Chip, Regfile, multi-chip board, and application-development workflow. [Section 2.3](#) defines irregular parallel algorithms and explains

why they are difficult to accelerate on SIMD architectures. [Section 2.4](#) introduces Ant Colony Optimization, the Traveling Salesman Problem, the Ant System, the Ant Colony System, and the multi-colony island model, and closes by relating their irregularity back to [Section 2.3](#).

2.1 Flynn’s Taxonomy

Flynn’s taxonomy classifies computer architectures according to the number of instruction and data streams that they can process concurrently. Flynn defines an **instruction stream** as the sequence of instructions executed by a machine and a **data stream** as the sequence of operands, intermediate results, and input data requested by those instructions [\[3\]](#). The term **stream** therefore describes execution as observed by the machine rather than the static instructions or data stored in a program. Combining one or multiple instruction streams with one or multiple data streams produces the four classes shown in [Table 1](#).

Class	Streams	General organization
SISD	Single instruction, single data	One instruction stream operates on one data stream.
SIMD	Single instruction, multiple data	One instruction stream simultaneously controls operations on multiple data streams.
MISD	Multiple instruction, single data	Multiple instruction streams operate on a single data stream or successive derivations of it.
MIMD	Multiple instruction, multiple data	Independent instruction streams operate on independent data streams.

Table 1: Flynn’s taxonomy of computer architectures.

A **Single Instruction, Single Data** (SISD) system executes one instruction stream on one data stream. It corresponds to the conventional sequential computer used by Flynn as the reference organization. An SISD processor can nevertheless overlap different stages of instruction processing through pipelining and can contain several specialized execution units. Flynn describes this overlap as **confluence**: several instructions may be in different execution phases concurrently even though the architecture exposes only one instruction and one data stream [\[3\]](#). Consequently, the taxonomy describes the multiplicity of streams rather than every form of internal hardware parallelism.

A **Single Instruction, Multiple Data** (SIMD) system applies the same instruction to multiple operands concurrently. It is particularly effective when a computation performs an identical operation over a large, regularly structured dataset. Typical applications include vector arithmetic, dense matrix operations, image processing, and other numerical kernels whose loop iterations are largely independent. Only one instruction stream has to be decoded and distributed, allowing many comparatively simple execution units to perform useful work simultaneously.

The requirement for a common instruction stream also imposes important restrictions. Flynn notes that SIMD systems require regular operand streams and generally restrict communication between execution units to predetermined, uniformly controlled patterns [\[3\]](#). If different data elements require different operations, the execution units cannot proceed independently. Modern SIMD and GPU-style SIMT processors commonly handle such branches by executing the possible paths separately and disabling lanes to which the current path does not apply. [\[1\]](#) This **branch divergence** reduces utilization because some execution lanes remain inactive.

Data-dependent memory accesses can similarly prevent contiguous or coalesced transfers, while irregular communication may require expensive rearrangement of data. SIMD architectures therefore offer high throughput for regular data-parallel workloads but may use their available execution resources inefficiently when control flow and memory access depend strongly on the processed data.

In a **Multiple Instruction, Single Data** (MISD) organization, multiple instruction streams operate on one data stream. Flynn discusses organizations in which a value passes through several execution units, each applying a different operation, as well as arrangements in which execution hardware is shared by several instruction sequences [3]. Pure MISD systems are uncommon as general-purpose computers. The category is nevertheless useful for describing specialized pipelines in which successive processing stages transform the same sequence of input data.

A **Multiple Instruction, Multiple Data** (MIMD) system contains multiple processing elements that execute independent instruction streams on separate data. Different processing elements may therefore follow different branches, operate at different rates, and perform different tasks. MIMD is suitable for task-parallel applications, distributed services, graph processing, discrete-event simulation, and other workloads whose activities cannot efficiently be expressed as one uniform operation over many operands. In particular, data-dependent control flow does not inherently produce the lane-idling penalty found in SIMD execution: one processing element can take a branch without requiring the others to execute the same path. [3]

This flexibility introduces different costs. Work must be partitioned among the processing elements, and the resulting tasks may require explicit communication and synchronization. Unpredictable task sizes can cause load imbalance, while shared resources can introduce contention. In distributed-memory MIMD systems, data placement is especially important because accessing remote state is typically slower and more expensive than accessing local memory. [4] Flynn also identifies interconnection between processing units and the identification of applications capable of using the available parallelism as central difficulties of MIMD organizations [3]. MIMD consequently avoids the requirement for uniform control flow, but does not automatically guarantee efficient parallel execution.

Flynn’s taxonomy is relevant to this thesis because it provides the architectural distinction needed to relate SpiNNaker2 to the irregularity of ACS. GPU execution is throughput-oriented and typically groups processing lanes that execute a common instruction, giving it important SIMD-like characteristics. [1] The probabilistic city selection in ACS causes ants to follow different control paths and access different pheromone and city entries. Such behavior conflicts with the regularity from which SIMD architectures obtain their efficiency.

SpiNNaker2, by contrast, is a distributed-memory MIMD system. Each PE executes its own program state and instruction stream, allowing colonies or ants to progress independently. Its packet-switched NoC provides communication between these independently operating PEs rather than requiring all of them to perform the same communication operation simultaneously. This organization is a conceptual match for irregular algorithms, although its practical effectiveness still depends on local-memory capacity, data placement, communication overhead, synchronization, and load balance.

The taxonomy must not be interpreted as a complete performance model. As Flynn observes, the four classes do not describe how instruction or data streams interact, and systems belonging to the same class can therefore have substantially different organizations and performance

characteristics [3]. For this thesis, the classification establishes why SpiNNaker2’s independent instruction streams may benefit ACS. The subsequent analysis of its memory hierarchy, NoC, and processing elements is required to determine whether that architectural advantage translates into practical performance.

2.2 SpiNNaker2 Hardware

This section describes the SpiNNaker2 chip and the multi-chip system used throughout this thesis. Subsection 2.2.1 gives a high-level overview of the chip’s organization; Subsection 2.2.2 describes the ARM Cortex-M4F processing element that executes application code; Subsection 2.2.3 describes the on-chip network used for communication; Subsection 2.2.4 explains the per-module register file used for configuration, interrupts, and synchronization; and Subsection 2.2.5 and Subsection 2.2.6 describe, respectively, the multi-chip board used for the experiments and the existing software used to develop and deploy applications on it.

2.2.1 Overview

The SpiNNaker2 chip is a massively parallel, low-power many-core processor designed primarily for energy-efficient event-based neuromorphic computing [2]. It is the successor to the original SpiNNaker architecture, featuring significant upgrades in compute capability and interconnect bandwidth.

The chip consists of 152 independent ARM Cortex-M4F Processing Elements (PEs), described in more detail in Subsection 2.2.2. These PEs are interconnected using a specialized Network-on-Chip (NoC) designed to handle massive numbers of small packets with very low latency, mimicking the spike traffic of biological neurons. [2]

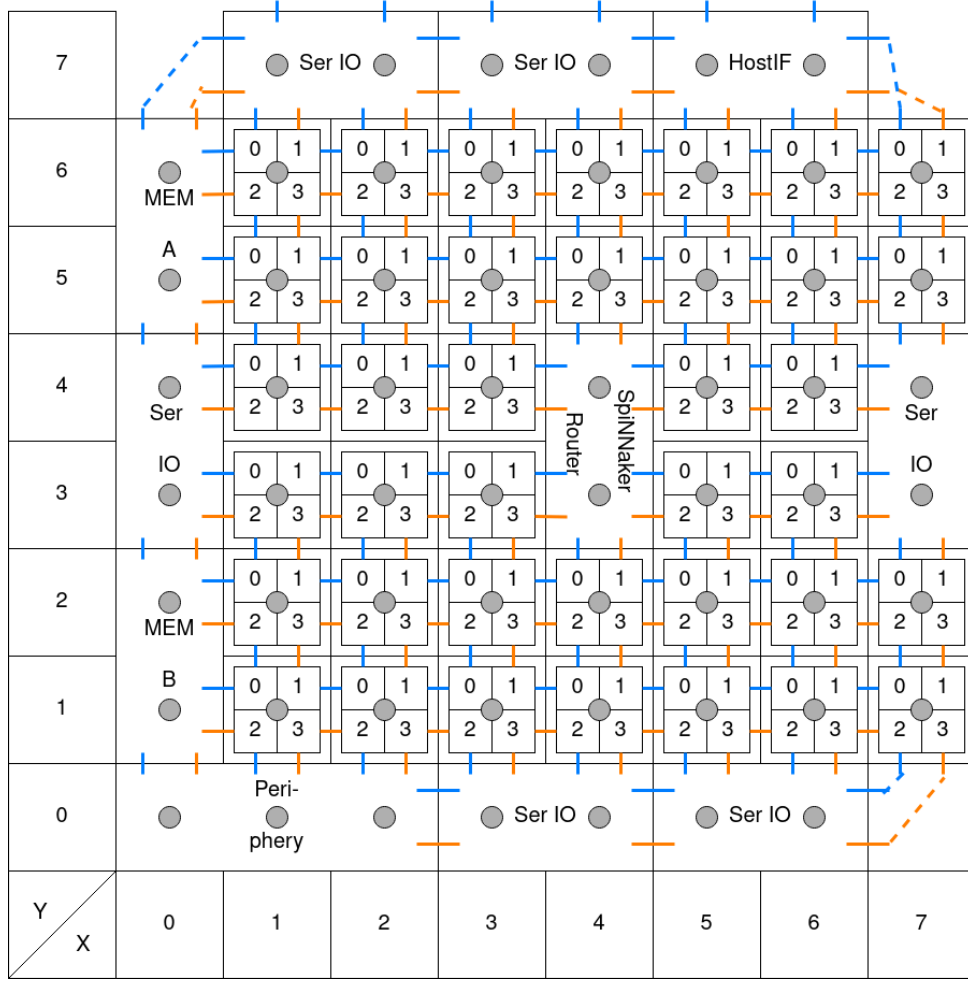


Figure 1: Topology of the SpiNNaker2 chip. [5] CNoC (blue) and DNoC (orange) bus lines connect the modules.

As seen in Figure 1, the modules are arranged in a 2D grid. Every module contains at least one NoC router for communication and is addressable via its X and Y location on the grid.

The chip is organized into 38 identical Quad Processing Elements (QPEs). [5] Each QPE contains four PEs sharing a common SDRAM controller and NoC router.

Communication with the outside world is handled by the SpiNNaker Router, which features six SerIO interfaces. [5], [6] These allow multiple SpiNNaker2 chips to be interconnected to form massive multi-chip systems.

Memory is hierarchical. Each PE has 128KB of local SRAM for code and data. Additionally, every chip has 2GB of dedicated LPDDR4 SDRAM, accessible to all PEs via the NoC. [7] This external memory is essential for storing large datasets that do not fit in local SRAM.

2.2.2 ARM Cortex-M4F Processing Element (PE)

Computation is done on the SpiNNaker2 hardware through ARM Cortex-M4F cores, a 32-bit RISC processor designed for embedded applications. [5] Key features include:

- **Nested Vectored Interrupt Controller (NVIC):** Supports low-latency interrupt processing, which is critical for the event-driven model of SpiNNaker2. [2]
- **Floating Point Unit (FPU):** Provides hardware support for single-precision (32-bit) floating-point arithmetic, compliant with the IEEE 754 standard. [8]

- **Memory Protection Unit (MPU)**: Enforces memory access permissions, improving system reliability and security.

Beyond the core, each PE includes hardware accelerators accessible via the local AHB bus:

- **Neural Math Unit (NMU)**: Accelerates fixed-point exponential and logarithmic functions, common in neural models, and provides a true random number generator. [9]
- **Machine Learning Accelerator (MLA)**: A systolic array for matrix-matrix multiplication and 2D convolution, optimized for inference tasks. [10]
- **Communications Control Unit (CCU)**: Manages the injection and reception of NoC packets. [6]

Unlike conventional multiprocessing systems with a shared OS, SpiNNaker2 PEs typically run a “bare-metal” or minimal runtime environment. There is no dynamic scheduling of tasks by an OS; instead, each core runs a pre-compiled program that reacts to interrupts (events) from the NoC or timers. [11] This minimizes overhead and allows cores to sleep aggressively to save power.

We use the terms “PE” and “core” interchangeably. We use “PE(x, y, id)” when referring to specific individual PEs, where id (0-3) identifies the PE on QPE(x, y). “QPE(x, y)” refers to a QPE with its X and Y NoC coordinates, as seen in Figure 1.

2.2.3 Network-on-Chip (NoC)

The SpiNNaker2 NoC is a custom 2D mesh interconnect that connects the routers of every module shown in Figure 1. It is split into two logically separate networks that share the same physical links: the Configuration NoC (CNoC), which carries small packets used to read and write Regfile registers (Subsection 2.2.4), and the Data NoC (DNoC), which carries the larger packets used to transfer application data between PEs and between a PE and SDRAM. [12] As shown in Figure 2, every PE on a QPE connects to that QPE’s DNoC router, while the QPE’s Regfile connects to its CNoC router; a NoC gateway allows packets to cross between the two networks when required. This separation keeps latency-critical configuration and interrupt traffic on the CNoC largely independent of the bulk data traffic on the DNoC.

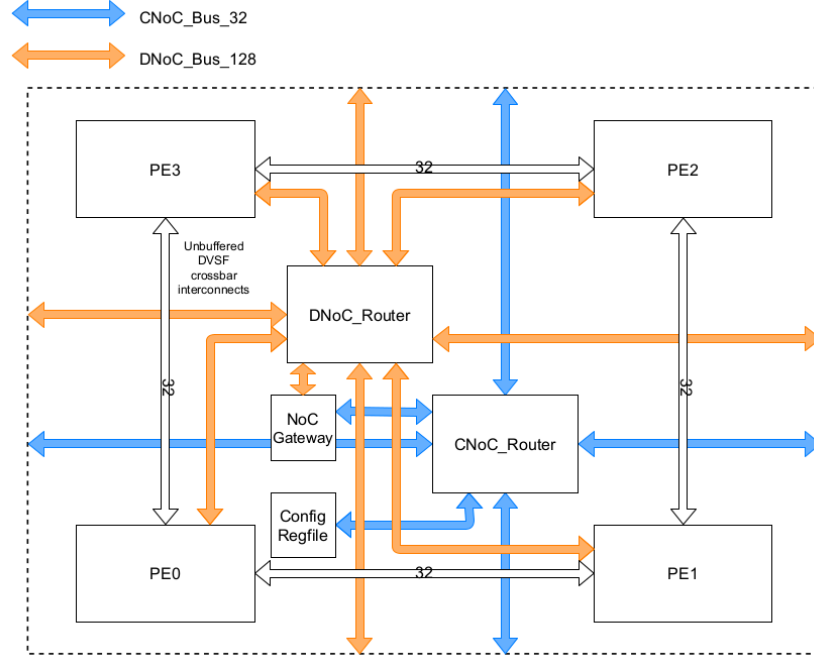


Figure 2: NoC routing inside a QPE. [5] Each PE is connected to the DNoC Router of the module. The Regfile is connected to the CNoC Router. Packets can cross over through a NoC gateway.

2.2.4 Regfile

Many of the coordination mechanisms used later in this thesis - starting and stopping cores, triggering interrupts, and synchronizing access to shared state - are not implemented as ordinary memory operations or software libraries, but as side effects of reading and writing specific hardware registers. Understanding the Regfile is therefore a prerequisite for understanding the interrupt (Subsection 4.1.2, Subsection 4.1.3) and semaphore (Subsection 4.1.5) abstractions built on top of it in Section 4.1.

Every module on a SpiNNaker2 chip has a register file (Regfile), which is an array of registers. [13] Various aspects of a module can be configured through the Regfile, such as the NoC routers, power management of the PEs on QPEs, SRAM clocking on modules with SRAM, and much more. [14] Some registers are also used for other features such as Regfile interrupts and Semaphores.

Regfiles can be accessed and changed through NoC packets. [12]

2.2.5 SpiNNaker2 Router & 48-Node Board

While a single SpiNNaker2 chip is powerful, it is often used as part of a larger system. A standard deployment involves at least one board containing 48 SpiNNaker2 chips, totaling 7,296 cores. [15] These boards communicate via high-speed links, scaling the system to hundreds of thousands of cores for large-scale simulations. [2]

Each chip's SpiNNaker Router, introduced in Subsection 2.2.1, exposes six SerIO links that connect neighboring chips on a board and, potentially, boards to each other. On the 48-node board used in this thesis, these links form a fixed board-level topology connecting every chip to its neighbors. [15] In principle, this topology would allow packets to be routed directly between chips without host involvement, which would be attractive for the multi-colony island model

developed in [Subsection 4.2.5](#). However, as discussed in [Subsection 4.1.8](#), we were unable to get this direct inter-chip routing to work in our deployment, and the board-scale implementation therefore relies on the host to relay information between chips.

2.2.6 SNN Application Development

The typical method for interacting with the SpiNNaker2 hardware as a user is using the py-spinnaker2 Python library [\[16\]](#), [\[17\]](#). It is a library designed for running Spiking Neural Networks (SNNs): a user defines a network of neuron and synapse populations in Python, and py-spinnaker2 maps this network onto PEs, generates the corresponding chip-side binaries, deploys them, and retrieves recorded spikes and other data after the simulation has finished. This workflow is well suited to SNN experiments but exposes little of the underlying hardware to applications that are not themselves spiking neural networks, motivating the more general-purpose tooling developed as *s2generic* in [Section 4.1](#).

2.3 Irregular Parallel Algorithms

As introduced through Flynn’s taxonomy in [Section 2.1](#), the effectiveness of a parallel architecture depends strongly on whether the instruction and data streams it processes are regular. Parallel algorithms are commonly classified into two broad categories based on their computational and communication patterns: regular and irregular. Regular parallel algorithms exhibit predictable, uniform control flow and data access patterns, making them well-suited for data-parallel architectures such as Graphics Processing Units (GPUs). In contrast, irregular parallel algorithms are characterized by non-uniform, data-dependent control flow, unpredictable memory access patterns, and dynamic load imbalances that arise during execution.

Formally, an irregular parallel algorithm can be defined as one where the computation graph—the set of tasks and their dependencies—is not known statically at compile time but emerges dynamically as the algorithm progresses. This dynamic nature manifests in several ways:

1. **Data-dependent control flow:** The execution path taken by each processing element depends on the data it processes, leading to divergent branches across parallel units.
2. **Unstructured communication:** Communication patterns between processing elements are irregular and may involve many-to-many interactions that cannot be predetermined.
3. **Dynamic load imbalance:** The computational workload per processing element varies unpredictably, causing some units to remain idle while others are overloaded.
4. **Irregular memory access:** Memory accesses follow non-contiguous, data-dependent patterns that defeat caching and prefetching mechanisms.

These characteristics make irregular algorithms challenging to accelerate on SIMD (Single Instruction, Multiple Data) architectures like GPUs, which achieve high throughput precisely through lockstep execution of identical operations on contiguous data. As noted in the literature, “for applications with abundant parallelism, GPUs deliver higher peak computational throughput than latency-oriented CPUs,” but this advantage is contingent on the parallelism being regular and well-structured [\[18\]](#). The irregular pattern of algorithms such as Ant Colony Optimization (ACO) in both data access and control flow constrains the performance of GPU-based approaches due to hardware limitations [\[19\]](#). Specifically, when the algorithm diverges in selection processes such as roulette wheel selection, the vectorization module of the compiler must execute both code paths of conditional statements using masks or predicates, dramatically reducing the utilization of SIMD units [\[19\]](#).

Multiple Instruction, Multiple Data (MIMD) architectures, by contrast, are inherently better suited to irregular parallelism. In a MIMD system, each processing element can execute its own instruction stream on its own data, allowing for divergent control flow and asynchronous execution. The SpiNNaker2 chip exemplifies this architectural philosophy: it is a massively parallel MIMD system designed specifically for irregular, event-driven computation. Each core on SpiNNaker2 operates independently, communicating with others via a packet-switched network, making it an ideal platform for algorithms where computation and communication patterns are unpredictable.

The relationship between irregular algorithms and MIMD architectures is fundamental: irregular algorithms require the flexibility of independent instruction streams that MIMD provides, while MIMD architectures benefit from algorithms that can exploit their ability to handle diverse, asynchronous workloads. Ant Colony Optimization (ACO) serves as an exemplary case of an irregular parallel algorithm, as we shall see in the following section.

2.4 Ant Colony Optimization & Traveling Salesman Problem

Ant Colony Optimization (ACO) is a population-based metaheuristic inspired by the foraging behavior of real ants [20]. It belongs to the class of swarm intelligence algorithms and has been successfully applied to a wide range of combinatorial optimization problems, many of which are graph-theoretic in nature [20]. The algorithm was first applied to the Traveling Salesman Problem (TSP) by Dorigo et al. [20], and TSP remains the canonical benchmark for evaluating ACO implementations.

2.4.1 The Traveling Salesman Problem

A precise definition of the TSP is required because the memory-efficient representation developed in Subsection 4.2.1 and the benchmark instances used in Section 5.1 both depend on specific properties of the problem, in particular that it can be restricted to symmetric, two-dimensional Euclidean distances. The following definitions also fix the notation for tours and tour length used throughout the remainder of this thesis.

The Traveling Salesman Problem is a classic NP-hard combinatorial optimization problem [18]. Formally, it can be defined as follows: Given a set of n cities and a distance matrix $D = [d_{ij}]$ where d_{ij} represents the distance between city i and city j , the goal is to find the shortest possible tour that visits each city exactly once and returns to the starting city. For a symmetric TSP, the distance matrix satisfies $d_{ij} = d_{ji}$ for all i, j , and the problem can be represented as a complete weighted graph $G = (V, E)$ with n nodes, where the edge weights correspond to the distances between cities [18].

Mathematically, a solution to the TSP is a permutation $\pi = (\pi_1, \pi_2, \dots, \pi_n)$ of the cities, and the total tour length is given by: [18]

$$L(\pi) = \sum_{i=1}^{n-1} d_{\pi_i, \pi_{i+1}} + d_{\pi_n, \pi_1} \quad (1)$$

The objective is to find the permutation π^* that minimizes $L(\pi)$. The search space contains $\frac{(n-1)!}{2}$ possible tours for a symmetric TSP, making exhaustive search infeasible for even moderate values of n .

2.4.2 The Ant System (AS) Algorithm for TSP

The Ant System (AS) is the earliest variant of ACO, first proposed by Dorigo [20]. In AS, simulated ants construct solutions to the TSP in the form of tours. The artificial ants are simple agents that construct tours in a parallel, probabilistic fashion, guided by simulated pheromone trails and heuristic information [20]. Pheromone trails are a fundamental component of the algorithm, as they facilitate indirect communication between agents via their environment [20].

The algorithm operates as follows. Initially, a population of m ants is placed randomly on the cities. Each ant then iteratively constructs a tour by moving from its current city to an unvisited city according to a probabilistic state transition rule. The probability that ant k at city i chooses to move to city j is given by:

$$p_{ij}^k = \frac{[T_{ij}]^\alpha \cdot [\eta_{ij}]^\beta}{\sum_{l \in N_i^k} [T_{il}]^\alpha \cdot [\eta_{il}]^\beta} \quad (2)$$

where:

- T_{ij} is the pheromone trail intensity on edge (i, j)
- $\eta_{ij} = \frac{1}{d_{ij}}$ is the heuristic information (visibility), representing the desirability of moving directly from city i to city j
- α and β are parameters controlling the relative importance of pheromone trails versus heuristic information
- N_i^k is the set of cities not yet visited by ant k (the feasible neighborhood)

This probabilistic selection mechanism, known as the random proportional rule, introduces randomness into the search process, allowing ants to explore different solutions while being guided by accumulated knowledge [18].

After all ants have completed their tours, the pheromone trails are updated. This update consists of two phases: evaporation and deposition. First, all pheromone values are reduced by a constant factor to simulate evaporation:

$$T_{ij} \rightarrow (1 - \rho) \cdot T_{ij} \quad (3)$$

where $\rho \in (0, 1]$ is the evaporation rate. Then, each ant deposits pheromone on the edges of its constructed tour:

$$T_{ij} \rightarrow T_{ij} + \sum_{k=1}^m \Delta T_{ij}^k \quad (4)$$

where ΔT_{ij}^k is the amount of pheromone deposited by ant k on edge (i, j) , typically defined as:

$$\Delta T_{ij}^k = \begin{cases} \frac{Q}{L_k} & \text{if ant } k \text{ traversed edge } (i, j) \text{ in its tour} \\ 0 & \text{otherwise} \end{cases} \quad (5)$$

Here, Q is a constant and L_k is the length of the tour constructed by ant k . This formulation ensures that shorter tours receive more pheromone reinforcement, creating a positive feedback loop that guides future ants toward better solutions [20].

2.4.3 Ant Colony System (ACS)

While the Ant System, described in [Subsection 2.4.2](#), laid the foundation for ACO, its performance on larger TSP instances was limited. To address these limitations, Dorigo and

Gambardella introduced the **Ant Colony System (ACS)** [21]. ACS introduces three key modifications to the original AS that significantly improve solution quality and convergence speed:

1. **Pseudo-random proportional rule:** Instead of the purely probabilistic selection used in AS, ACS employs a more aggressive exploitation mechanism. The next city j for ant k at city i is chosen according to:

$$j = \begin{cases} \arg \max_{l \in N_i^k} ([T_{il}]^\alpha \cdot [\eta_{il}]^\beta) & \text{if } q \leq q_0 \\ J & \text{otherwise} \end{cases} \quad (6)$$

where q is a random variable uniformly distributed in $[0, 1]$, $q_0 \in [0, 1]$ is a parameter that controls the balance between exploitation and exploration, and J is a city selected according to the same probabilistic rule as in Equation 2. When $q \leq q_0$, the ant deterministically chooses the best possible next city (exploitation); otherwise, it explores probabilistically. This mechanism allows ACS to focus search on promising regions of the solution space while still maintaining diversity.

2. **Global pheromone update:** In ACS, only the globally best ant (i.e., the ant that found the shortest tour since the start of the algorithm) is allowed to deposit pheromone after each iteration. The global update rule is:

$$T_{ij} \Rightarrow (1 - \rho) \cdot T_{ij} + \rho \cdot \Delta T_{ij}^{\text{best}} \quad (7)$$

where $\Delta T_{ij}^{\text{best}} = \frac{1}{L_{\text{best}}}$ if edge (i, j) belongs to the global-best tour, and 0 otherwise. This differs from AS, where all ants deposit pheromone. By reinforcing only the best solution, ACS accelerates convergence toward high-quality solutions.

3. **Local pheromone update:** During tour construction, each ant applies a local pheromone update immediately after moving from city i to city j . The local update rule is:

$$T_{ij} \Rightarrow (1 - \xi) \cdot T_{ij} + \xi \cdot T_0 \quad (8)$$

where $\xi \in (0, 1)$ is the local pheromone decay parameter, and T_0 is the initial pheromone value. This local update reduces the pheromone on visited edges, making them less attractive to subsequent ants in the same iteration. This mechanism encourages exploration by preventing multiple ants from following the same path, thereby diversifying the search.

The combination of these three mechanisms makes ACS significantly more effective than AS on TSP benchmarks. As noted by Dorigo and Gambardella, “the key to the application of ACS to a new problem is to identify an appropriate representation for the problem (to be represented as a graph searched by many artificial ants) and an appropriate heuristic that defines the distance between any two nodes of the graph” [21]. Furthermore, ACS is “amenable to efficient parallelization, which could greatly improve the performance for finding good solutions, especially for high-dimensional problems” [21].

2.4.4 Multi-Colony Island Model

While the Ant Colony System (ACS) improves upon the basic Ant System through enhanced exploitation and exploration mechanisms, further performance gains can be achieved by distributing the search across multiple independent colonies that periodically exchange information. This approach is known as the **multi-colony island model** [22].

In the multi-colony island model, the complete population of ants is divided into several sub-colonies (islands), each of which searches independently for solutions [22]. These colonies may interchange information—such as pheromone trails or best solutions—during the execution process at regular intervals [22]. The specific variant where colonies share the best solutions between them and substitute their own solutions if the received one is better is called the **multi-colony island model** [22].

Formally, consider a set of p islands, each containing a subpopulation of ants. Each island i maintains its own pheromone matrix T^i and runs an independent ACO algorithm (e.g., AS or ACS) for a fixed number of iterations or until a migration event is triggered. At regular intervals (every k iterations), islands exchange information according to a predefined neighborhood topology. The migration process can be described as follows:

1. **Selection:** Each island selects its best-so-far solution s_i^* (the shortest tour found by any ant on that island).
2. **Communication:** The selected solution is sent to neighboring islands according to the topology (e.g., ring, torus, or fully connected).
3. **Replacement:** Upon receiving a solution s_j^* from a neighbor, island i compares it with its own best solution.

If $L(s_j^*) < L(s_i^*)$, the island replaces its best solution with the received one and may also update its pheromone matrix accordingly.

The migration interval k is a critical parameter. As noted in the literature, “the bigger the k value is, the faster the algorithm reaches the solution” [22], because less frequent communication reduces synchronization overhead. However, too infrequent migration may cause islands to converge to different local optima without benefiting from the discoveries of other islands.

The multi-colony island model is inspired by the island model from Genetic Algorithms [22] and belongs to the broader category of **decentralized multi-colony approaches**. This is in contrast to **centralized master/slave architectures**, where “there is one node, called master process, which collects the solutions or the pheromone information from all the other nodes” [22]. The decentralized nature of the island model makes it particularly well-suited for MIMD architectures like SpiNNaker2, where each island can be assigned to an independent core or group of cores, communicating asynchronously via the packet-switched network.

The island model has been shown to yield excellent results in multi-objective optimization contexts, where different islands can specialize in exploring different regions of the Pareto front [22]. For single-objective problems like the TSP, the model provides a natural way to parallelize the search while maintaining solution quality through periodic information exchange.

2.4.5 Irregularity in ACO, ACS, and Multi-Colony Models

Both AS and ACS exhibit several characteristics that classify them as irregular parallel algorithms. The multi-colony island model introduces additional irregularity due to its decentralized communication patterns:

1. **Data-dependent control flow:** The probabilistic state transition rule causes each ant to follow a different path through the city graph. The sequence of cities visited—and consequently the computational path—depends on the random choices made by each ant, leading to divergent execution across parallel agents. In ACS, the pseudo-random proportional rule introduces an additional conditional branch (the comparison $q \leq q_0$), further increasing

control flow divergence. As noted in the literature, “due to the fact that the ACO algorithm diverges in the RW selection process, the vectorization module of the compiler has to execute both code path of an if-statement (or loop-statement) by masks (predicates). Thus, the utilization of SIMD units could be reduced dramatically because more instructions are generated” [19].

2. **Irregular memory access:** During tour construction, each ant must access the pheromone and distance matrices to compute transition probabilities. The access pattern depends on the ant’s current city and the set of unvisited cities, which varies dynamically and unpredictably across ants. The tour list, which maintains the chronological order of visited cities, must be read and updated frequently, contributing to irregular memory access patterns [18, 23]. Furthermore, “since the ant travel is guided by a stochastic process, the memory access pattern is unpredictable” [19]. In the multi-colony island model, each island maintains its own pheromone matrix, which introduces additional memory access irregularity as islands may access different regions of the solution space.
3. **Dynamic load imbalance:** The computational cost of constructing a tour is not uniform across ants. The number of feasible moves (cities not yet visited) decreases as the tour progresses, and the time required to compute transition probabilities depends on the size of the feasible neighborhood, which varies across ants and over time. In ACS, the local pheromone update adds another computational step after each move, further contributing to load imbalance. In the island model, load imbalance can also arise between islands: some islands may converge faster than others, leading to idle time during synchronization points if synchronous communication is used.
4. **Unstructured communication:** In the pheromone update phase, all ants must update the global pheromone matrix. This creates a many-to-many communication pattern where multiple ants may attempt to update the same edge simultaneously, requiring synchronization or atomic operations. In ACS, the global update is restricted to the best ant, but the local update introduces additional write operations during tour construction. The multi-colony island model introduces an additional layer of communication: islands must exchange the best solutions at regular intervals. This communication pattern is irregular because the timing and content of migrations depend on the progress of each island, which is itself data-dependent. As noted in the literature, “when the transferred information is just a solution (usually the best), some synchronization points are required and so, the computational time is increased as the communication frequency increases” [22].

These irregular characteristics make ACO, ACS, and multi-colony island models challenging to accelerate on SIMD architectures like GPUs, where divergent control flow and irregular memory access can severely degrade performance. As noted in the literature, early GPU implementations of ACO focused on accelerating only the tour construction stage using task-based parallelism, with pheromone update performed on the CPU [18]. More recent work has attempted to parallelize both stages using data-parallel approaches, but these require careful design to accommodate the irregular nature of the algorithm [18]. Despite these efforts, “due to the irregular (random) pattern of ACO algorithms in data access and control flow, the performance of GPU-based approaches is constrained by hardware limitations” [19].

The MIMD architecture of SpiNNaker2, with its independent cores and asynchronous communication, is naturally suited to handle the irregular parallelism inherent in ACO, ACS, and multi-colony island models. Each ant can be assigned to an independent core, allowing for divergent execution paths without the overhead of SIMD divergence penalties. The packet-switched net-

work enables flexible communication patterns for pheromone updates and interisland migration, while the distributed memory model avoids the contention issues that arise in shared-memory systems. This makes SpiNNaker2 a promising platform for accelerating these irregular parallel algorithms.

3 Related Works

This chapter surveys prior approaches to accelerating Ant Colony Optimization (ACO) on parallel hardware and situates SpiNNaker2 within this landscape. [Section 3.1](#) discusses GPU-based parallelizations of ACO, which represent the most extensively studied hardware target and were already used in [Section 2.3](#) to motivate the mismatch between irregular algorithms and SIMD execution. [Section 3.2](#) then summarizes FPGA-based approaches. [Section 3.3](#) reviews how SpiNNaker2 itself has been positioned in the literature beyond its neuroscience origins, and [Section 3.4](#) concludes by identifying the specific gap that this thesis addresses.

3.1 GPU-Based Parallelization of ACO

Graphics Processing Units are the most thoroughly studied accelerator target for ACO, precisely because their throughput-oriented, data-parallel design creates the tension already introduced in [Section 2.3](#): high peak performance is available only if the workload can be expressed as regular, largely uniform operations across many data elements [\[18\]](#). Early GPU implementations exploited this by parallelizing only the tour-construction phase, mapping ants onto threads or thread blocks, while the sequential pheromone update was left on the host CPU [\[18\]](#). This “task-based” split avoids the most divergent part of the algorithm - the many-to-many pheromone deposition - but does not remove the divergence inherent in per-ant city selection, and it introduces a data transfer between host and device on every iteration.

Later work attempted to parallelize both the construction and update phases directly on the GPU using data-parallel formulations, mapping the pheromone matrix onto the GPU’s memory hierarchy and updating it with atomic operations or per-edge reduction schemes [\[18\]](#). These designs improve GPU utilization but still have to accommodate the irregular access pattern produced by the probabilistic city-selection rule. Zhou et al. describe this limitation directly: because ACO “diverges in the RW [roulette-wheel] selection process, the vectorization module of the compiler has to execute both code path[s] of an if-statement [...] by masks (predicates)”, which reduces SIMD-unit utilization and constrains achievable speedups [\[19\]](#). The same source concludes that, “due to the irregular (random) pattern of ACO algorithms in data access and control flow, the performance of GPU-based approaches is constrained by hardware limitations” [\[19\]](#). Mitigations reported in this line of work include grouping ants so that threads within a warp are more likely to follow the same branch, restructuring the candidate-list computation to improve memory coalescing, and compressing or reordering the distance and pheromone matrices to reduce irregular global-memory traffic. [\[24\]](#) Even with these techniques, published GPU implementations generally report that speedups over sequential CPU baselines fall well short of the raw arithmetic throughput advantage of the device, which the literature attributes to exactly the irregularity characteristics enumerated in [Section 2.3](#).

3.2 FPGA-Based Acceleration of ACO

Field-Programmable Gate Arrays offer an alternative that sidesteps SIMD divergence entirely by allowing a custom, application-specific data path to be synthesized for the algorithm. [25], [26], [27] Reported FPGA implementations of ACO typically pipeline the probability computation and city-selection steps in dedicated hardware, and some designs replicate this pipeline to process several ants concurrently, achieving substantially better energy efficiency per solution than comparable CPU or GPU implementations. This efficiency comes at a considerable development cost: expressing the algorithm in a Hardware Description Language requires re-implementing arithmetic, memory arbitration, and control logic that a software implementation obtains from a compiler and standard library, and the resulting design is tied to the specific TSP instance size, numeric precision, and parameter ranges for which it was synthesized. FPGA accelerators for ACO are consequently well suited to fixed, latency- or power-critical deployments, but their lack of general-purpose programmability makes them a poor match for the exploratory, general-purpose use case pursued in this thesis, where the same platform is expected to run substantially different algorithms with only C-level recompilation.

3.3 SpiNNaker2 Beyond Neuromorphic Workloads

SpiNNaker2 itself, as described in Section 2.2, was designed primarily as a platform for large-scale, energy-efficient spiking neural network simulation and event-based machine learning [2]. The platform’s designers have nevertheless repeatedly argued that its architectural properties are not inherently specific to neuroscience applications. Mayr et al. observe that, “apart from the platform’s traditional usage as a neuroscience exploration tool, the extended functionality [of SpiNNaker2] opens up new application areas such as automotive AI, tactile internet, industry 4.0 and biomedical processing” [28]. Gonzalez et al. similarly emphasize that SpiNNaker2’s combination of event-based and asynchronous processing “sets significantly different requirements for the development of learning algorithms” to mainstream dense, synchronous accelerators, and that, “despite the programming challenges, SpiNNaker2 paves the path for future fully event-based supercomputers with a negligible Amdahl limit and an energy-proportional operation beyond [deep neural networks]” [2]. The same discussion explicitly lists prospective non-neural application areas, including “the utilization of the massive parallelism in probabilistic computing and distributed drug discovery” [2], alongside large-scale brain simulation and event-based machine learning more narrowly.

These statements indicate that the platform is understood, by its own developers, as a general-purpose massively parallel MIMD system whose applicability extends beyond spiking neural networks. However, the published evaluations of SpiNNaker2 that we are aware of remain concentrated on neural workloads: spiking-network inference and training benchmarks, energy measurements for specific neuron and synapse models, and comparisons against other neuromorphic or deep-learning accelerators on tasks phrased as SNN simulation. We are not aware of a published, quantitative evaluation of SpiNNaker2 on a combinatorial optimization or graph-search workload expressed directly in terms of general-purpose C code, independent of the SNN abstractions provided by py-spinnaker2 (Subsection 2.2.6).

3.4 Gap Addressed by This Thesis

Taken together, the reviewed literature shows that ACO has been studied extensively on GPUs, where its irregularity constrains achievable speedups (Section 3.1); on FPGAs, where custom

data paths trade programmability for efficiency (Section 3.2); and on neuromorphic hardware, where pheromone dynamics are expressed through spiking abstractions rather than general-purpose code [29]. SpiNNaker2’s designers explicitly anticipate applications beyond neuroscience (Section 3.3), but, to our knowledge, no published work evaluates SpiNNaker2 as a general-purpose MIMD accelerator for an irregular combinatorial optimization problem such as the TSP, nor documents the software infrastructure required to do so outside the SNN-oriented tooling described in Subsection 2.2.6.

This thesis addresses that gap along two axes. First, it contributes *s2generic* (Section 4.1), a software layer that exposes SpiNNaker2’s interrupt, DMA, and semaphore facilities to arbitrary C applications rather than to SNN models specifically, filling the documentation and functionality gap identified in Section 4.1. Second, it contributes a memory-efficient ACS implementation and a multi-colony island model (Section 4.2) together with an experimental evaluation against an x86 baseline (Chapter 5, Chapter 6), providing - to our knowledge - the first detailed case study of a non-neural, irregular optimization workload on SpiNNaker2 hardware.

4 Methodology

This chapter describes how the goals of Section 1.2 are realized in practice. Section 4.1 presents the design of *s2generic*, the software layer used to develop and deploy general-purpose applications on SpiNNaker2. Section 4.2 then describes how the Ant Colony System is implemented on top of this layer, progressing from a memory-efficient representation of a single colony (Subsection 4.2.1, Subsection 4.2.2, Subsection 4.2.3) to a single-chip multi-colony island model (Subsection 4.2.4) and finally to a multi-chip board-scale implementation spanning 48 chips (Subsection 4.2.5).

4.1 Design of the *s2generic* SDK and Library

This section describes the design of *s2generic*, the SDK and library developed for this thesis to run general-purpose C applications on SpiNNaker2. It first motivates the need for such a layer and explains how applications are loaded and started (Listing 1). It then presents the individual mechanisms *s2generic* provides on top of the existing SpiNNaker2 software: typed host access to chip-side variables (Subsection 4.1.1), Regfile and Feedthrough interrupts (Subsection 4.1.2, Subsection 4.1.3), remote SRAM access (Subsection 4.1.4), hardware semaphores (Subsection 4.1.5), enabled-core discovery (Subsection 4.1.6), chip-side logging (Subsection 4.1.7), and the limitations we encountered regarding inter-chip communication (Subsection 4.1.8). These mechanisms are used throughout the ACS implementation described in Section 4.2.

The existing software ecosystem for SpiNNaker2, primarily the s2-sim2lab-app SDK and py-spinnaker2 library, present several significant hurdles for general-purpose application development. These tools are heavily optimized for Spiking Neural Network (SNN) workloads, resulting in a lack of documentation and functionality for generic use cases. It would still be technically possible to build a non-neural application directly on top of these libraries, since the underlying hardware access does not depend on the SNN abstractions themselves. In practice, however, this would require reverse-engineering internal data structures intended for neuron and synapse configuration, working around API calls that assume an SNN model is being deployed, and reimplementing basic facilities that are not exposed to application developers. Doing so for

every new generic application would be highly impractical and error-prone. Concrete examples of missing functionality include a general mechanism for reading and writing arbitrary, user-defined chip-side variables from the host (Subsection 4.1.1), abstractions for the Regfile-based interrupt and synchronization primitives (Subsection 4.1.2, Subsection 4.1.5), and a way to determine at runtime which cores of a chip are actually enabled (Subsection 4.1.6).

The development of an SDK for development of generic applications is thus necessary. We will call this SDK and associated library *s2generic*.

On the chip side we want to use as many functionalities from the s2-sim2lab-app SDK as possible and provide our own solutions and documentation where the existing SDK is lacking.

As the host side C++ code for interaction with the SpiNNaker2 hardware in s2-sim2lab-app seems to be broken, or at least is not usable for the deployment available to us, we need another solution for interacting with the SpiNNaker2 hardware. We suspect that this C++ tooling was primarily developed and tested against chips connected directly via Ethernet, whereas the deployment described in Section 5.2 connects boards through an STM card; the resulting differences in addressing and connection setup may explain why the provided host-side code does not function in our environment. In our testing we were able to get py-spinnaker2 applications, as introduced in Subsection 2.2.6, running. py-spinnaker2 utilizes the SpiNNMan2 python library for low-level interaction with the SpiNNaker2 hardware. SpiNNMan2 provides functionality for

- establishing a connection to a SpiNNaker2 board or chip,
- locking and resetting the hardware,
- loading a given compiled C application on selected cores,
- reading and writing to memory on the chips, and
- triggering Interrupts.

Many of these features require the *SpiNNaker Control and Monitor Processor* (SCAMP) to be running on the chips. SCAMP can be seen as a form of firmware, which is running on the periphery module and PEs 0 and 2 of QPE (5, 3) [30]. SpiNNMan2 can load and start SCAMP on the required chips automatically. The library also provides a description of the hardware layout.

We can thus use the SpiNNMan2 library on the host side for interaction with the hardware.

```

1 from pathlib import Path
2 from spinn_machine import CoreSubset, CoreSubsets
3 from spinn_machine.version.version_factory import SPIN2_GEN
4 from spinnman.scamp_constants import APPLICATION_CORES
5 from s2generic.session import EasyApp, easy_session
6 with easy_session(
7     board_type=SPIN2_GEN.SPIN2_48CHIP,
8     board_ips=[
9         # "192.168.1.17",
10        # "192.168.1.18",
11        # "192.168.1.19",
12        # "192.168.1.20",
13    ],
14    n_boards=1, # currently only one supported

```

 Python

```

15     force_reset=False, # set to True if board is having issues
16 ) as (session, hw_desc):
17     app = EasyApp(
18         session,
19         hw_desc,
20         cores="all",
21         # cores="chip00",
22         # cores=CoreSubsets(
23         #     [ # just selecting a small subset of cores
24         #         CoreSubset(0, 0, list(APPLICATION_CORES.keys())[0:10]),
25         #         CoreSubset(2, 1, list(APPLICATION_CORES.keys())[2:5]),
26         #     ]
27         # ),
28         memfile_path=Path("binaries/app.mem"),
29         memjson_path=Path("binaries/app.json"),
30         auto_load=True,
31         auto_start=True,
32         remove_known_faulty_cores=True,
33     )
34     # main() has now begun running on the cores
35     # we can now use 'app' to interact with the application

```

Listing 1: Using *s2generic* to load and start an application on SpiNNaker2 hardware.

In *s2generic* we provide `easy_session()` and `EasyApp()` to easily start a session and load and start an application, as shown in [Listing 1](#). The `cores` parameter of `EasyApp` selects onto which PEs the application will be loaded. Any known faulty cores will automatically be removed. `EasyApp` will upload a bit array of which cores are enabled on a chip to each selected core of that chip. Applications can use this data on the chip side to select cores for specific tasks without a need to hardcode exact cores.

The `app.mem` and `app.json` files are generated from the ARM ELF binary of the application, which is compiled from the chip side C source code of the application. `app.mem` describes which bytes should be loaded where into the SRAM of each enabled core.

4.1.1 Accessing Chip-Side Variables from Host

`EasyApp` uses `app.json`, which contains a description of the C symbols with their name, address in SRAM, and size in bytes. This enables us to use the proper names of C variables with some type safety and runtime assertions when reading and writing memory from the chips in Python, instead of using arbitrary addresses.

To prevent the compiler from optimizing reads and writes away, variables that should be accessible from the host must be declared as volatile. They must also be 4-byte aligned and their size in bytes must be an integer multiple of 4. This is required as the ARM Cortex-M4F core works on 32-bit words and has unexpected behavior on misaligned memory access [\[31\]](#). Multiple examples in `s2-sim2lab-app` also use 16-byte alignment for larger arrays, though we were unable to find a rationale for this.

To model C variables in Python we provide an abstract class `CVar[T]` in *s2generic*. `EasyApp` provides a function for creating an instance of a given subclass of a `CVar` and the name of

a variable. Subclasses need only to implement functions for encoding and decoding the given type `T` from and to bytes, as well as to string. We provide multiple useful subclasses, such as `CVarUInt32`, `CVarFloat`, and `CVarUInt32Array`. [Listing 2](#) and [Listing 3](#) demonstrate the usage of CVars.

`s2-sim2lab-app` and `s2generic` declare some variables internally, which can also be accessed from the host using CVars. One such variable is `uint32_t exit_code` which stores the return value of the `main()` function. While the program is running, or stuck in an infinite loop, the exit code is set to `0xDEADDEAD`. An exit code of `0xBADC0D3D` indicates an assertion failure. `0xBA11ADE5`, `0xBA5EBA11`, and `0xDEADC0DE` indicate hard fault, bus fault, and usage fault of the ARM Cortex-M4F core respectively.

Furthermore, we provide `CVarStructure[S]` and `CVarStructureArray[S]` to interact with C structs. `S` has to be a class implementing `ctypes.Structure` and `_fields_` set corresponding to the struct in C.

```

1 volatile uint32_t mynumber = 0;
2 volatile float myfloatarray[10] __attribute__((aligned(4))) = {0};
3
4 // this main() function gets run on every enabled core in parallel
5 int main() {
6     myfloatarray[5] = 123.456f;
7     while (mynumber != 42) {} // busy wait for mynumber to be set to 42
8     myfloatarray[5] = 654.321f;
9     return (int)0xcafebabe; // success
10 }
```

Listing 2: Chip side example of variables that are accessible by host.

```

1 mynumber = app.var(CVarUInt32, "mynumber")
2 myfloatarray = app.var(CVarFloatArray, "myfloatarray")
3 exit_code = app.var(CVarUInt32, "exit_code") # return of main(), or error code
4
5 # iterate over all enabled cores
6 for pe in app.iter_cores():
7     assert exit_code.read(pe) == 0xDEADDEAD # ensure main() is still running
8     print(myfloatarray.read(pe)[5]) # prints 123.45600128173828
9     mynumber.write(pe, 42) # write 42 into mynumber of the PE, causing the
    while loop to end
10
11 time.sleep(1) # not necessary, just to be really sure main() has time to return
12
13 for pe in app.iter_cores():
14     assert exit_code.read(pe) == 0xCAFEBADE # ensure main() returned
    successfully
15     print(myfloatarray.read(pe)[5]) # prints 654.3209838867188
```

Listing 3: Host side code for [Listing 2](#).

4.1.2 Regfile Interrupts

Regfile Interrupts are a type of interrupt fully managed through Regfile registers. Every QPE has 9 configurable Regfile Interrupt lines (0-8) for each of its four PEs. [14] Regfile Interrupt 8 is reserved internally for usage during DRAM access, thus we can utilize Regfile Interrupts 0 through 7 for usage in generic applications.

In conjunction with the **WFI** (wait-for-interrupt) instruction, which lets a core go into a low power state until an interrupt occurs [32], power efficient event based applications can easily be developed.

We provide functions to easily let a core subscribe and unsubscribe to and from its Regfile Interrupt lines. When subscribing to an Interrupt we assign a priority which is used by the NVIC of the ARM Cortex-M4F if multiple different interrupts occur at the same time. Lower priority values indicate a higher priority, with 0 being the highest priority and $(1UL \ll _NVIC_PRIO_BITS) = 32$ being the lowest priority. [11]

When Regfile Interrupt *X* is triggered on a PE, *s2generic* calls the corresponding `Regfile_INT_X_Handler_Callback()` function, assuming the PE subscribed to it first. As Regfiles are managed through the NoC a Regfile Interrupt can be triggered by any PE on the same chip as demonstrated in Listing 4 or by the host using `app.transceiver.interrupt_core(*pe, X)`.

The SpiNNaker2 chip design requires a Regfile Interrupt to be explicitly marked as handled. If the Interrupt is not marked as handled, then the handler will be called immediately again after it returns causing an infinite loop. To prevent this *s2generic* automatically marks a Regfile Interrupt as handled before calling the callback function. If the interrupt is triggered again while the callback is still running, the NVIC of the PE should mark the interrupt as pending and thus call the callback again once it returns. [33] The callback function must return a boolean indicating whether *s2generic* should tell NVIC to clear such a pending interrupt if it exists. The behavior of NVIC on pending interrupts seems to be unreliable in practice and thus should not be relied upon. We recommend using a Semaphore when each trigger of a Regfile Interrupt should cause its callback function to be called exactly once.

Our Regfile Interrupt related functions take multiple Regfile Interrupt lines at once as a mask for convenience.

```

1  #include "common.h" // contains regfile interrupt related functions and types
2
3  volatile uint32_t counter_Regfile_INT_2_Handler = 0;
4
5  // called when Regfile Interrupt 2 is triggered
6  bool Regfile_INT_2_Handler_Callback(Regfile_INT_Line self ALLOWUNUSED) {
7      counter_Regfile_INT_2_Handler++;
8      return false; // do not clear pending interrupts
9  }
10
11 // this main() function gets run on every enabled core in parallel
12 int main() {
13     // subscribe to Regfile Interrupt 2 with a low priority
14     // (technically only PE(1, 2, 3) must subscribe in this example)

```

```

15  rf_subscribe_regfile_interrupts(Regfile_INT_2_Line, (1UL << __NVIC_PRI0_BITS)
    - 2UL);
16
17  if (my_core_id == PEID(1, 2, 3)) {
18      // we are on PE(1, 2, 3)
19      while (counter_Regfile_INT_2_Handler < 10) {
20          __WFI(); // wait-for-interrupt, puts core in low power state until an
                interrupt occurs
21      }
22      // we reach this point only if Regfile Interrupt 2 was triggered at least 10
        times on this core
23      return (int)0xCAFEBAE;
24  } else {
25      // we are on any other core
26      rf_trigger_regfile_interrupts(1, 2, 3, Regfile_INT_2_Line); // send a NoC
        Packet to trigger Regfile Interrupt 2 on PE(1, 2, 3)
27      return (int)0xCAFEBAE;
28  }
29  }

```

Listing 4: Example usage of Regfile Interrupt 2 on the chip side. PE(1, 2, 3) only returns successfully if the Interrupt is triggered on it at least 10 times. All other cores try to trigger Regfile Interrupt 2 on PE(1, 2, 3). If at least 10 other cores are enabled on the chip and no Interrupt gets swallowed due to race conditions, then PE(1, 2, 3) will return successfully.

During our testing we found that there is a small delay between changes to the Regfile of a QPE and the NVIC of the PEs registering these changes. This may either be undocumented behavior, the delay introduced by Regfile changes going through the NoC router, or a hardware bug, but we found that this delay was not longer than 50 instructions. We thus add a small delay of 50 NOP instructions in the appropriate places of the Regfile Interrupt related functions to prevent undefined behavior.

4.1.3 Feedthrough Interrupts

In contrast to Regfile Interrupts, which are triggered on individual PEs, Feedthrough Interrupts are triggered on an entire chip. They are *feed through* all PEs on a chip and are thus received by all PEs on a chip at the same time. There are 4 Feedthrough Interrupt lines (0-3) on each chip. [14]

Feedthrough Interrupts can be used to synchronize all cores on a chip or to send very simple commands to each core, such as to proceed to the next stage of its program. An example of this can be seen in Listing 5.

Only the periphery module on a chip can trigger Feedthrough Interrupts. [14] Unfortunately the periphery module is fully managed by SCAMP. [30] As we were unable to obtain the source code nor any detailed documentation for SCAMP, we are unable to provide a method for triggering a Feedthrough Interrupt from the chip side.

However, SCAMP enables us to trigger Feedthrough Interrupts from the host side. In Python, we can trigger a Feedthrough Interrupt on all chips and thus all PEs by calling

`app.transceiver.global_interrupt(X)`, where `X` is the Feedthrough Interrupt line we want to trigger.

s2generic provides subscribe and unsubscribe functions for Feedthrough Interrupts similar to those for Regfile Interrupts. Each QPE also has a Regfile entry for each PE with a mask on which Feedthrough Interrupt lines should be “enabled” and thus be able to be received by the NVIC of said PE. We thus also provide functions to “enable” and “disable” the Feedthrough Interrupt lines on a PE.

When a Feedthrough Interrupt line is active and a PE is subscribed to that line and has that line “enabled”, then its corresponding `FT_INT_X_Handler()` function is called.

In our testing we found Feedthrough Interrupt lines to remain active for many instructions when triggered via Python. Thus, they are likely still active once the handler function on the PE has returned. The NVIC interprets the line being active as the interrupt still being triggered and thus will call the handler function many more times. [33] Feedthrough Interrupt handlers thus should be designed with this in mind and either

- disable or unsubscribe from that line to be called only once,
- run long enough to return after the line is no longer active, or
- perform only idempotent actions.

Feedthrough Interrupt line 0 is used internally by *s2generic*. Its handler can thus not be overwritten and not be used in generic applications. *s2generic* uses the Feedthrough Interrupt during startup to inform all cores that they have received some internal variables can now continue to initialize and call `main()`. This is necessary as in our testing we found that SRAM cannot be written to from the host before the application is actually running. The line may thus still be active when `main()` is called.

```

1  #include "common.h"
2  volatile bool running = true;
3
4  // handler of Feedthrough Interrupt 3
5  void FT_INT_3_Handler(void) {
6      rf_disable_feedthrough_interrupts(FT_INT_3_Line); // "disable" line in
        Regfile, causing handler to be called only once
7      running = false;
8  }
9
10 // this main() function gets run on every enabled core in parallel
11 int main() {
12     // subscribe to Feedthrough Interrupt 3 with a low priority
13     rf_subscribe_feedthrough_interrupts(
14         FT_INT_3_Line, // mask of lines to subscribe to
15         (1UL << __NVIC_PRI0_BITS) - 2UL, // priority
16         true // whether they should also be "enabled" in the Regfile
17     );
18
19     // wait until running is set to false by the handler of Feedthrough Interrupt
        3
20     while (running) {

```

```

21     __WFI();
22 }
23
24 return (int)0xCAFEBABE;
25 }

```

Listing 5: Example usage of Feedthrough Interrupts on the chip side. Every core subscribes to Feedthrough Interrupt 3 and waits until `running` is set to `false`. The value is set to `false` by the handler function of Feedthrough Interrupt 3 `FT_INT_3_Handler()`. `main()` will thus only return if `app.transceiver.global_interrupt(3)` is called in the Python code on the host. The handler function also “disables” the line for itself to prevent being called multiple times, so this would technically not be necessary in this example as the handler only performs idempotent actions.

4.1.4 Accessing Variables in SRAM of other PEs

The main method of communicating data between PEs on a chip is through direct memory access (DMA). Any PE can access the SRAM of another PE on the chip using NoC packets. [4]

PEs have the SRAM of all other PEs implicitly memory-mapped. `s2-sim2lab-app` provides a helper function `get_global_addr_pe_mem(tar_x, tar_y, tar_pe, tar_addr)` to retrieve the global memory-mapped address of the target address `tar_addr` on the SRAM of `QPE(tar_x, tar_y, tar_pe)`. [4] As *s2generic* places the same compiled binary with the same memory layout in SRAM of each PE, pointers to a volatile global variable can safely be used as target address when trying to access the same variable of another core. Listing 6 contains an example of accessing a variable on another PE through memory-mapping.

When accessing SRAM of another PE through memory-mapping all reads and writes are implicitly done through individual NoC packets with a maximum payload size of 1-word. To transfer larger chunks of contiguous data the helper functions `dma_wr()` and `dma_rd()` should be used instead, which are also provided by `s2-sim2lab-app`. They are able to use the full maximum payload size of 16-bytes per DNoC packet and reuse the NoC configuration between packets. [4]

The functions provided by `s2-sim2lab-app` are sufficient for usage in generic applications and no further abstraction is required by us.

In our testing we found that when accessing another PEs SRAM, the target PE is not blocked nor notified of this, even though the documentation states that should be possible. [4] Race conditions can be prevented by using a semaphore. A Regfile Interrupt can be used to notify the core of any potential change.

4.1.5 Semaphores

Every QPE has 4 counting Semaphores (0-3) in its Regfile. [14] These semaphores provide hardware-level synchronization primitives that can be used to coordinate access to shared resources between PEs on the same chip. Each semaphore maintains a count value in the Regfile that can be atomically incremented (released) or decremented (acquired).

As they are Regfile-controlled, any core on a chip or the host can acquire and release them using NoC packets. We provide functions in *s2generic* to easily configure, acquire, and release Semaphores. The acquire function takes a number of max attempts and returns a boolean, useful to handle deadlocks gracefully.

If s2-sim2lab-app's `init_dram_dma()` is called on a PE, then Semaphores 0 and 1 of that QPE are reserved for usage during DRAM access. Our functions contain assertions to catch if these semaphores are used anyway, preventing unexpected behavior.

On the host side, we provide a context manager in Python that can be used to safely acquire and release a semaphore of a given QPE and chip, ensuring proper cleanup even in error scenarios.

[Listing 6](#) demonstrates the usage of Semaphore 3 to protect a critical section where multiple PEs increment values in another PE's SRAM array. The semaphore ensures that only one PE at a time performs the read-modify-write operations, preventing race conditions.

```

1  #include "common.h"
2  volatile uint32_t myarray[100] = {0};
3
4  // this main() function gets run on every enabled core in parallel
5  int main() {
6      // configure Semaphore 3 on the current QPE to have 1 resource
7      // Technically this should only be done by one PE on each QPE, though
8      // there is no harm as long as semaphores are not used before every PE has
9      // run this.
10     configure_qpe_semaphore_3(my_qpe_x, my_qpe_y, 1);
11
12     // the global address of myarray on PE(1, 2, 3)
13     uint32_t *myarray_on_PE123 = (uint32_t *)get_global_addr_pe_mem(1, 2, 3,
14     (uint32_t)&myarray);
15
16     // block until we have a lock acquired on Semaphore 3 of QPE(1, 2)
17     acquire_qpe_semaphore_3(1, 2, UINT32_MAX);
18     // only one PE on a chip will be in this section at a time
19
20     // increment all values in myarray on PE(1, 2, 3) by 1
21     // (this would be prone to race conditions without the lock on the Semaphore)
22     for (int i = 0; i < 100; i++) {
23         // NOTE: any read and write to myarray_on_PE123 will be done through NoC
24         // packets
25         myarray_on_PE123[i] = myarray_on_PE123[i] + 1;
26     }
27
28     // release the lock, so that another PE may acquire it
29     release_qpe_semaphore_3(1, 2);
30
31     return (int)0xcafebabe; // success
32 }
```

Listing 6: Example usage of Semaphore 3 to protect a critical section. All enabled cores attempt to increment each element of `myarray` on PE(1, 2, 3) by 1. Semaphore 3 on QPE(1, 2) ensures mutual exclusion, preventing race conditions during the read-modify-write operations.

4.1.6 Enabled Cores

Not every application uses, or wants to use, every PE of a chip: some PEs may be marked as faulty and unavailable, an application may only require a subset of the available cores, and different applications may want to reserve different roles for specific cores without hardcoding their exact positions on the chip. Hardcoding a fixed set of PE identifiers inside chip-side C code would also make an application brittle, since the set of usable cores can change between deployments as faulty cores are identified. Applications therefore need a reliable, runtime-visible notion of which cores are currently enabled.

To provide this, EasyApp writes a bit field, `_enabled_cores_bit_field`, into the SRAM of every selected core at application start. Each bit of this field corresponds to one PE of the chip, using the numbering scheme introduced in [Subsection 2.2.2](#), and is set if and only if that PE was selected and successfully loaded by EasyApp ([Listing 1](#)). Because *s2generic* places an identical binary and memory layout on every selected core ([Subsection 4.1.4](#)), every enabled core on a chip observes the same bit field and therefore agrees on which of its neighbors are enabled.

s2generic provides the macro `FOR_EACH_ENABLED_CORE` to iterate over every core marked as enabled in this bit field, as used in the broadcast implementations of [Subsection 4.2.4](#) and [Subsection 4.2.5](#). Applications can also query whether a specific core is enabled, or determine the numerically first and last enabled core on the chip. This is used, for example, to select a single, well-defined “supervisor” core without hardcoding a specific PE identifier, as done in the board-scale implementation of [Subsection 4.2.5](#). Finally, an application can compare the number of enabled cores against a required minimum, allowing it to fail predictably rather than behave incorrectly if too few cores were successfully loaded for a given algorithm to run.

[Listing 7](#) demonstrates the resulting API on the chip side.

```

1  #include "common.h"
2
3  int main() {
4      // require at least two enabled cores: one supervisor, one worker
5      assert(num_enabled_cores >= 2);
6
7      if (my_core_id == first_enabled_core) {
8          // deterministically chosen supervisor core, regardless of its
9          // physical (x, y, pe) location on the chip
10         return supervisor_main();
11     }
12
13     uint32_t num_neighbors = 0;
14     FOR_EACH_ENABLED_CORE(core_id) {
15         if (core_id != my_core_id) {
16             num_neighbors++;
17         }
18     }
19
20     return worker_main(num_neighbors);
21 }
```

Listing 7: Example usage of enabled-core discovery. A deterministically chosen supervisor core is selected without hardcoding its identifier, and every worker counts its enabled peers.

4.1.7 Chip Side Logging and Assertions

Debugging an application running independently on over a hundred PEs, without an attached debugger or a conventional standard output, is difficult. *s2-sim2lab-app* addresses this by providing a `printf`-compatible logging function. *s2generic* configures a small ring buffer and initializes this `printf` implementation. Log messages written on a core, such as those used for debugging the broadcast logic in [Subsection 4.2.4](#), accumulate in this buffer instead of being transmitted immediately. On the host side, `pe.read_log()` retrieves the buffer for a given PE, allowing accumulated messages to be printed or stored once execution has stopped or during a monitoring pause. Because logging is asynchronous with respect to the host, it is not a substitute for the precise, low-latency synchronization mechanisms of [Subsection 4.1.2](#) and [Subsection 4.1.5](#); it is intended purely as a debugging aid.

s2-sim2lab-app also provides an `assert()` macro for chip-side code. When its condition is false, the macro sets the reserved `exit_code` variable introduced in [Subsection 4.1.1](#) to `0xBADC0D3D` and halts the core, allowing the host to detect the failed assertion by polling `exit_code` as shown in [Listing 3](#), together with any log messages recorded up to that point. This `assert()` macro is only usable because of the initialization of `printf` by us. *s2-sim2lab-app* does not document any of this properly. It should also be noted that the `printf` implementation used has several bugs when printing floating point numbers.

4.1.8 Inter-Chip Communication

The SpiNNaker2 Router introduced in [Subsection 2.2.5](#) is, in principle, capable of routing packets directly between chips over their SerIO links, which would allow the multi-colony island model of [Subsection 4.2.4](#) to be extended across chips without host involvement. In our deployment, however, we were unable to get this direct inter-chip communication working at all: packets addressed to a neighboring chip did not arrive, and neither the available documentation nor the SCAMP source code, which we could not obtain, allowed us to diagnose the cause. We treat this as a limitation of the specific deployment and software versions available to us rather than as a fundamental property of the SpiNNaker2 Router, and we return to its consequences in [Chapter 7](#).

Because direct inter-chip routing is unavailable, any exchange of information between chips must instead be mediated by the host. We implement this as a simple pull-and-forward mechanism: the host periodically reads the current best-known tour from one designated core per chip and, if it improves on the best tour known on another chip, writes it there in turn. This is the mechanism used by the board-scale implementation in [Subsection 4.2.5](#), where it is restricted to a single core per chip to keep the number of host-chip interactions manageable.

4.2 Implementing ACO on SpiNNaker2

This section describes how the ACS algorithm from [Subsection 2.4.3](#) is implemented on SpiNNaker2 using the *s2generic* facilities from [Section 4.1](#), [Subsection 4.2.1](#) and [Subsection 4.2.2](#) first develop a memory-efficient representation of a TSP instance and of ACS state that fits within the DTCM of a single PE. [Subsection 4.2.3](#) then implements ACS on one PE in isolation,

before [Subsection 4.2.4](#) extends the implementation to a multi-colony island model running across all PEs of a chip, and [Subsection 4.2.5](#) further extends this model to the full 48-chip board.

The implementation is constrained primarily by the small amount of local memory available to each PE ([Subsection 2.2.2](#)) and by the cost of accessing non-local memory. Although a PE contains 128 KiB of SRAM, only 96 KiB is available as Data Tightly Coupled Memory (DTCM) for application data. The implementation must therefore fit the TSP instance, pheromone trails, ant states, intermediate values, and stack into this memory.

The 2 GB LPDDR4 memory could provide substantially more capacity, but every access to it must traverse the NoC ([Subsection 2.2.3](#)). The data-dependent access patterns described in [Section 2.3](#) make it difficult to combine these accesses into predictable, contiguous transfers. Using external memory for the frequently accessed distance or pheromone matrices would consequently introduce NoC traffic into the innermost loop of the algorithm. The implementation therefore keeps all performance-critical data in the local SRAM of each participating PE.

We implement the Ant Colony System described in [Subsection 2.4.3](#) rather than the original Ant System. ACS generally requires fewer pheromone deposits because only the best-known tour participates in the global update. It also offers an explicit balance between exploration and exploitation and encourages diversity through local pheromone updates. These properties are useful when several independent colonies execute on different PEs.

4.2.1 Representation of TSP Instances

As defined in [Subsection 2.4.1](#), a TSP instance can be represented by a distance matrix D containing the distance between every pair of cities. For n cities, this representation requires $O(n^2)$ entries. A matrix using 32-bit floating-point values occupies

$$M_{D(n)} = 4n^2 \text{ bytes.} \quad (9)$$

A matrix for 154 cities already occupies

$$M_{D(154)} = 4 \cdot 154^2 = 94,864 \text{ bytes.} \quad (10)$$

It would therefore consume almost all available DTCM without accounting for pheromone trails, ant states, the stack, or other application data. Storing a complete distance matrix is consequently unsuitable for the implementation.

We restrict the implementation to symmetric, two-dimensional Euclidean TSP instances. Instead of storing all pairwise distances, each PE stores the Cartesian coordinates of every city and computes distances as required:

$$d_{ij} = \sqrt{(x_i - x_j)^2 + (y_i - y_j)^2}. \quad (11)$$

The storage requirement is thereby reduced from $O(n^2)$ to $O(n)$. Using two 32-bit floating-point coordinates per city requires eight bytes per city. For the maximum supported instance size of 256 cities, the complete coordinate array occupies only

$$256 \cdot 2 \cdot 4 = 2,048 \text{ bytes.} \quad (12)$$

This representation exchanges additional arithmetic for a substantial reduction in memory consumption. In particular, every distance query requires a subtraction, multiplication, addition, and square-root operation. The distance function is called frequently during city selection, but this cost is preferable to the capacity and NoC-access problems of a complete distance matrix.

The relevant declarations are shown in Listing 8. `NUM_CITIES` and `cities` are global and `volatile` because they are initialized by the host through the mechanism described in Subsection 4.1.1. All PEs receive the cities in the same order.

```

1  typedef struct {
2      float x;
3      float y;
4  } city_data_t;
5
6  #define NUM_CITIES_MAX (UINT8_MAX + 1)
7
8  volatile uint32_t NUM_CITIES;
9  volatile city_data_t cities[NUM_CITIES_MAX];
10
11 typedef uint8_t city_id_t;
12
13 __STATIC_INLINE float distance(
14     const city_data_t *a,
15     const city_data_t *b
16 ) {
17     float dx = a->x - b->x;
18     float dy = a->y - b->y;
19     return sqrtf(dx * dx + dy * dy);
20 }
21
22 __STATIC_INLINE float dist(city_id_t i, city_id_t j) {
23     return distance(
24         (city_data_t *)&cities[i],
25         (city_data_t *)&cities[j]
26     );
27 }

```

Listing 8: Representation of a two-dimensional Euclidean TSP instance.

Tours must be stored both as intermediate solutions and as results that can be transferred to other PEs or read by the host. A tour is represented by an array containing city identifiers in visitation order. The edge from the final array element back to the first is implicit and completes the tour. Storing the closing city a second time is therefore unnecessary.

Because every PE and the host use the same ordering of the coordinate array, a city's array index can serve as its identifier. A `uint8_t` can represent indices from 0 to 255, limiting an instance to 256 cities. This is considered an acceptable compromise because the TSP search space grows factorially with the number of cities, as discussed in Subsection 2.4.1, while the selected representation substantially reduces communication volume.

A complete tour requires at most 256 bytes. Since a DNoC packet can carry up to 16 bytes of payload, a maximum-size tour can be transferred in 16 packets:

$$\left\lceil 256 \frac{\text{bytes}}{16} \text{ bytes per packet} \right\rceil = 16 \text{ packets.} \quad (13)$$

In contrast, transferring the coordinates themselves as part of every tour would require eight times as much data. The compact representation is therefore particularly useful for exchanging best-known solutions between colonies.

The 256-city limit requires care when iterating over city identifiers. A loop variable of type `uint8_t` wraps from 255 to 0 and therefore cannot represent the terminating value 256. Loops whose upper bound may equal `NUM_CITIES_MAX` must use a wider integer type and cast to `city_id_t` only when indexing a city.

4.2.2 Memory-Efficient ACS State

Each colony requires three principal categories of state: the ACS parameters, the ant states, and the pheromone trails. Scalar parameters require little memory and are stored as 32-bit floating-point values. These include the heuristic exponent `BETA`, the exploration probability, the global and local update coefficients, and the initial pheromone value `TAU0`. The pheromone matrix is the dominant storage requirement.

The transition weight used by the implementation is

$$w_{ru} = T_{ru} \eta_{ru}^{\beta}. \quad (14)$$

Thus, the pheromone exponent α from the general ACO formulation is fixed to one. The parameter named `BETA` corresponds to β in [Subsection 2.4.3](#). `EXPLORATION_BIAS` stores the probability of performing probabilistic exploration; exploitation is selected with probability $1 - \text{EXPLORATION_BIAS}$.

A full pheromone matrix using 32-bit floating-point values would require

$$4 \cdot 256^2 = 262,144 \text{ bytes}, \quad (15)$$

which is substantially larger than the complete DTCM. The restriction to Euclidean TSP instances makes the distance matrix symmetric:

$$d_{ij} = d_{ji}. \quad (16)$$

Reversing a tour consequently does not change its length. The implementation also treats pheromone trails as undirected and enforces

$$T_{ij} = T_{ji}. \quad (17)$$

Only one triangular half of the symmetric pheromone matrix, including the diagonal, must therefore be stored. It contains

$$\frac{n(n+1)}{2} \quad (18)$$

entries. For 256 cities, this is 32,896 entries. The array index for an edge (r, u) , after exchanging the identifiers such that $r \leq u$, is calculated as

$$\text{index}(r, u) = rn - \frac{r(r+1)}{2} + u. \quad (19)$$

Using 32-bit floating-point values would still require 131,584 bytes. The implementation instead stores pheromone values as unsigned 16-bit integers, reducing the matrix to

$$2 \cdot \frac{256 \cdot 257}{2} = 65,792 \text{ bytes.} \quad (20)$$

The encoded values represent non-negative pheromone intensities below the compile-time limit "EDGE_MAX" = 4. For a conversion factor F , encoding and decoding are performed as

$$\text{encode}(T) = \lfloor TF \rfloor, \quad \text{decode}(e) = \frac{e}{F}. \quad (21)$$

In the implementation, F is calculated as `floor(UINT16_MAX / EDGE_MAX)`, which evaluates to 16,383 for an edge limit of four. The resulting resolution is approximately

$$\frac{1}{F} \approx 6.104 \cdot 10^{-5}. \quad (22)$$

A positive pheromone value smaller than this resolution would normally be truncated to zero. Such a value could make an otherwise feasible transition impossible and could eventually cause the sum of all transition weights to become zero. To prevent this, every positive value that would be encoded as zero is rounded upward to the smallest positive representation, namely the integer value one. The variable `edge_conversion_errors` counts how often this correction occurs.

The local ACS update is a convex combination of the current and initial pheromone values:

$$T' = (1 - \xi)T + \xi T_0. \quad (23)$$

It therefore remains between T and T_0 . For the canonical ACS global update,

$$T' = (1 - \rho)T + \frac{\rho}{L_{\text{best}}}, \quad (24)$$

the updated value likewise remains between T and $\frac{1}{L_{\text{best}}}$. Consequently, the fixed-point range is safe if both T_0 and all possible global deposits are below `EDGE_MAX`.

The compact pheromone representation is shown in [Listing 9](#).

```

1 volatile uint16_t edges[
2     NUM_CITIES_MAX * (NUM_CITIES_MAX + 1) / 2
3 ] __attribute__((aligned(16)));
4
5 #define EDGE_MAX 4.0f
6 #define EDGE_CONVERSION_FACTOR \
7     ((float)((uint16_t)((float)UINT16_MAX) / EDGE_MAX))
8
9 __STATIC_INLINE void set_edge_pheromones_tau(
10     city_id_t r,
11     city_id_t u,
12     float value
13 ) {
14     if (r > u) {
15         city_id_t temporary = r;
16         r = u;
17         u = temporary;
18     }

```

```

19
20     uint32_t index = r * NUM_CITIES;
21     index -= (uint32_t)r * ((uint32_t)r + 1) / 2;
22     index += u;
23
24     assert(value >= 0.0f && value < EDGE_MAX);
25
26     uint16_t encoded =
27         (uint16_t)(value * EDGE_CONVERSION_FACTOR);
28
29     if (value > 0.0f && encoded == 0) {
30         encoded = 1;
31         edge_conversion_errors++;
32     }
33
34     edges[index] = encoded;
35 }
36
37 __STATIC_INLINE float edge_pheromones_tau(
38     city_id_t r,
39     city_id_t u
40 ) {
41     if (r > u) {
42         city_id_t temporary = r;
43         r = u;
44         u = temporary;
45     }
46
47     uint32_t index = r * NUM_CITIES;
48     index -= (uint32_t)r * ((uint32_t)r + 1) / 2;
49     index += u;
50
51     return ((float)edges[index])
52         / EDGE_CONVERSION_FACTOR;
53 }

```

Listing 9: Triangular 16-bit representation of the symmetric pheromone matrix.

Each ant must record the cities it has visited, their visitation order, and the accumulated tour length. The ordered list uses the compact tour representation introduced in [Subsection 4.2.1](#). Once an ant has completed its tour, its state is consequently already in the format required for a DMA transfer or host read.

Checking whether a city has already been visited occurs in the innermost city-selection loop. Searching the partial tour for every check would reduce memory usage but introduce an additional linear scan. The implementation instead stores a separate `visited` array with one Boolean value per city. Although this array could be bit-packed into 32 bytes, the byte-per-

city representation permits direct indexing and avoids bit extraction instructions. This is a deliberate time-space trade-off.

An ant state is defined in [Listing 10](#).

```

1  typedef struct {
2      bool visited[NUM_CITIES_MAX];
3      uint32_t num_visited;
4      city_id_t tour[NUM_CITIES_MAX];
5      float tour_length;
6  } __attribute__((aligned(4))) ant_t;
7
8  #define MAX_NUM_ANTS 40
9
10 volatile uint32_t NUM_ANTS = 20;
11 ant_t ants[MAX_NUM_ANTS]
12     __attribute__((aligned(16)));
13
14 volatile city_id_t shortest_known_tour[
15     NUM_CITIES_MAX
16 ] __attribute__((aligned(16)));
17
18 volatile float shortest_known_tour_length = INFINITY;

```

Listing 10: State stored for each ant.

Under the target ABI, where `bool` occupies one byte, one `ant_t` occupies 520 bytes: 256 bytes for the visited set, four bytes for `num_visited`, 256 bytes for the tour, and four bytes for the tour length. The maximum of 40 ants therefore requires 20,800 bytes.

The principal statically allocated data structures for a maximum-size instance are:

Structure	Size
City coordinates	2,048 bytes
Pheromone matrix	65,792 bytes
40 ant states	20,800 bytes
Best tour and its length	260 bytes
Temporary transition weights	1,024 bytes

Together, these structures require approximately 89.9 kB before scalar variables, alignment padding, and the stack are considered. The maximum configuration therefore leaves only a small memory margin. The compile-time sizes of the structures and the final linker-reported DTCM usage must be checked whenever the data layout changes.

The maximum number of ants is fixed at compile time because dynamic allocation is avoided on the PEs. `NUM_ANTS` selects how many entries are active at runtime. Host-visible configuration and result variables are declared `volatile` and globally accessible as described in [Subsection 4.1.1](#). Internal ant state does not need to be host-visible and is therefore not declared `volatile`.

4.2.3 Single-PE Implementation

We first develop the algorithm for a single PE and only afterward extend it to communicate with other PEs. This staged approach lets us validate the memory layout and the correctness of the ACS logic from [Subsection 4.2.1](#) and [Subsection 4.2.2](#) in isolation, before introducing the additional complexity of inter-PE communication.

Before any computation can begin, the host must transfer the city coordinates and the ACS parameters described in [Subsection 4.2.2](#) to every participating PE using the mechanism from [Subsection 4.1.1](#). As this transfer happens from outside the running application, a PE has no direct means of detecting when the host-initiated writes to its SRAM have completed; the write itself does not generate any event that the core could wait on. We resolve this by having the host trigger Feedthrough Interrupt line 1 once all cities and parameters have been uploaded to every enabled core, as described in [Subsection 4.1.3](#). This lets every PE be informed of the same fact simultaneously with a single interrupt, avoids polling, and allows a core to remain in the low-power `__WFI()` state until the upload is confirmed complete, at which point the `FT_INT_1_Handler()` sets `running` to `true`.

Once released, a PE first calls `assert_aco_parameters()` to validate the parameters received from the host, resets `shortest_known_tour_length` to infinity, resets its iteration counters, and calls `reset_edges(TAU0)`. The latter initializes every entry of the triangular pheromone matrix from [Subsection 4.2.2](#) to the initial pheromone value T_0 , matching the initialization used by ACS in [Subsection 2.4.3](#).

The main loop, shown in [Listing 11](#), executes ACS iterations until one of two conditions is met. First, because ACS is a stochastic algorithm, no iteration can be identified in advance as final; we therefore approximate convergence by tracking the iteration of the last improvement of `shortest_known_tour_length` and terminating once at least `MIN_ITERATIONS` iterations have elapsed and no improvement has occurred for `MAX_ITERATIONS_WITHOUT_IMPROVEMENT` further iterations. Second, the host may request early termination at any time by triggering Feedthrough Interrupt line 3, whose handler clears `running`. Both conditions are checked at multiple points inside the loop body so that a PE can abort tour construction promptly rather than only between iterations.

Each iteration begins by resetting every ant with `reset_ant()` and placing it on a uniformly randomly chosen city via `random_city()`. Starting from diverse cities, rather than from a single fixed city, increases the diversity of the tours constructed within one iteration, which is desirable given that only a single tour is reinforced during the subsequent global update. Tour construction then proceeds in lockstep across ants: an outer loop advances the shared “step” counter from 0 to `NUM_CITIES - 2`, and an inner loop advances every ant by exactly one city per outer step using `choose_city_s()` (implementing the pseudo-random proportional rule of [Subsection 2.4.3](#)) and `travel_to_city()`. Interleaving the ants in this way keeps the algorithm’s structure simple, since it removes any need for one ant to be aware of the progress of another; the only shared ant-visible state is the pheromone matrix, which is intentionally updated immediately after every move through `perform_local_trail Updating()`. This ordering is precisely the local pheromone update of ACS described in [Subsection 2.4.3](#), and it lowers the attractiveness of just-used edges for the remaining ants of the same iteration, promoting exploration within a single colony.

Once every ant has completed its tour, the PE determines the best ant of the current iteration through a linear scan of `ants[].tour_length`. Following ACS, only this iteration-best tour

participates in the global pheromone update, performed by `perform_global_trail Updating()`. The PE additionally compares the iteration-best tour against the overall best tour found on this PE so far. If it is shorter, the compact tour representation from [Subsection 4.2.1](#) is copied into the global, `volatile` variable `shortest_known_tour`, and `shortest_known_tour_length` is updated accordingly, allowing the host to read both at any time as described in [Subsection 4.1.1](#). `shortest_known_tour_length` in particular lets the host decide, without transferring the entire tour, whether the larger `shortest_known_tour` array needs to be downloaded at all, reducing the amount of host–chip traffic required for monitoring.

Because `shortest_known_tour` and `shortest_known_tour_length` may be read by the host and written by the PE at any time, a concurrent read could otherwise observe a partially updated tour. We therefore protect this critical section with one of the four Regfile semaphores described in [Subsection 4.1.5](#), using `my_qpe_pe` to select the semaphore, giving each of the four PEs on a QPE its own dedicated semaphore and avoiding unnecessary contention between PEs that do not yet share any state at this stage of the implementation.

Strictly speaking, this synchronization is unnecessary when only a single core is executing, since the host does not need to read the intermediate results while the algorithm converges. However, it is required for benchmarking: as discussed later in [Chapter 5](#), the host must repeatedly sample the current best tour and its length at regular intervals to record the algorithm’s progress over time, and it may do so at essentially any point during execution. Even in this single-PE stage, running the implementation on multiple cores simultaneously is already useful, although the PEs are not yet aware of each other. Each core searches independently, and the host can simply select the best tour among all running cores once execution stops. Since all cores expose their state independently, the host can also use whichever core reaches a satisfactory solution first to instruct every core to terminate early, by triggering Feedthrough Interrupt line 3 on the whole chip.

4.2.4 Single-Chip Multi-Colony Island Model Implementation

The single-PE implementation of [Subsection 4.2.3](#) uses only one of the 148 PEs available on a chip. The next step is therefore to let all enabled PEs of a chip contribute to the search for the same TSP instance, rather than merely running many unrelated single-PE searches side by side.

Several designs could achieve this. One option is to broadcast every local pheromone update to all other PEs as soon as it occurs. This is unattractive because the local update in [Subsection 4.2.3](#) happens once per ant per city, i.e. up to $\text{NUM_ANTS} \times (\text{NUM_CITIES} - 1)$ times per iteration; broadcasting each of these updates to every other PE would generate an enormous number of NoC packets and would reintroduce exactly the kind of NoC traffic in the innermost loop that [Subsection 4.2.1](#) and [Subsection 4.2.2](#) were designed to avoid. A second option is to centralize the pheromone matrix in the chip’s shared LPDDR4 SDRAM and let every PE update it there. This again places NoC accesses into the innermost loop, and additionally introduces contention on the shared SDRAM controller of a QPE, which is used by all four of its PEs. A third option, following the multi-colony island model of [Subsection 2.4.4](#), is to let every PE keep its own independent pheromone matrix and broadcast the entire matrix to all other PEs whenever it finds an improved tour. Although this avoids per-move communication, a full pheromone matrix requires up to 65,792 bytes ([Subsection 4.2.2](#)), i.e. more than 4,000 DNoC packets, which is still costly to transmit on every improvement, and it would additionally destroy the very property that makes several independent colonies useful in

the first place: that each one explores the search space with its own, independently evolving pheromone landscape [22]. We therefore choose a fourth option, broadcasting only the compact best-known tour whenever a PE improves on its own record, keeping each PE’s pheromone matrix private and independently evolving. This corresponds directly to the multi-colony island model described in Subsection 2.4.4, with every enabled PE acting as one island, and it is a natural match for SpiNNaker2’s distributed-memory MIMD organization: no PE needs to be interrupted, synchronized, or centrally coordinated for the search itself to proceed, and communication occurs only sporadically and asynchronously, exactly matching the irregular communication pattern discussed in Section 2.3 and Subsection 2.4.5.

Rather than notifying a PE via a Regfile Interrupt whenever an improved tour arrives, we deliberately let each PE continue its own iteration undisturbed and simply use its own `shortest_known_tour` for the global pheromone update at the end of every iteration, regardless of whether that tour was found locally or received from another PE. This avoids interrupting a PE while it is in the middle of ant tour construction, where an asynchronous update to shared ant state could otherwise create data races, and it fits naturally with the fact that different PEs already progress through iterations at different, unsynchronized rates.

The broadcast itself, shown in Listing 12, is implemented using the list of enabled cores and the DMA primitives provided by *s2generic* (Subsection 4.1.6, Subsection 4.1.4). When a PE finds a new best tour, it iterates with `FOR_EACH_ENABLED_CORE` over every other enabled core on the chip and writes its `shortest_known_tour` and `shortest_known_tour_length` into that core’s memory-mapped SRAM. Because several PEs may now attempt to broadcast an improvement to the same target core at the same time, we extend the use of the per-PE semaphores introduced in Subsection 4.2.3: a broadcasting PE must acquire the target PE’s semaphore before writing to it, exactly as the host must for reading. Before performing the write, the broadcasting PE first reads the target’s current `shortest_known_tour_length` through the memory-mapped pointer obtained from `get_global_addr_pe_mem()`, which costs only a single-word NoC transaction. If the target already holds a tour that is at least as good, the broadcasting PE cancels its own write, since the target will soon receive an equal or better tour from whichever PE is responsible for it; this avoids overwriting an improvement with an inferior tour due to a race between two concurrent broadcasts. We use the chip-side logging facility of Subsection 4.1.7 to record, for debugging purposes, whether a broadcast completed, was pre-empted by a better remote tour, or timed out while waiting for a semaphore, which we treat as an indication of a potential deadlock.

4.2.5 Scaling Multi-Colony Island Model Implementation on 48-Node Board

Having distributed the search across all PEs of a single chip, the next step is to extend the multi-colony island model to all 48 chips of a board. As explained in Subsection 4.1.8, we were unable to establish direct inter-chip communication through the SpiNNaker2 Router, so any exchange of information between chips must be mediated by the host, which periodically pulls data from and forwards data to the chips. This restriction fundamentally shapes the design of the board-scale implementation.

A naive extension of the design from Subsection 4.2.4 would have the host periodically iterate over every one of the more than 7,000 cores on the board, read their current best tour, and rebroadcast the globally best tour to all of them. The link between host and chips is considerably slower than the on-chip NoC, so individually addressing thousands of cores at a sufficient

frequency is impractical. However, our chip-level implementation already lets the PEs of a single chip converge on a shared best tour among themselves through the broadcast mechanism of [Subsection 4.2.4](#). We can therefore reduce the number of cores the host must interact with from one per PE to one per chip, by reading from and writing to only a single designated core per chip and relying on that chip’s own PEs to propagate the tour further.

Two additional requirements guide the design of this designated core. First, a tour that a worker PE has only just started broadcasting to its chip should not be missed by the host simply because the host happens to poll the designated core before the broadcast reaches it. Second, an improved tour written by the host to a chip should be propagated to all of that chip’s PEs as quickly as possible, which is not guaranteed if the designated core happens to already be occupied constructing its own tour when the host writes to it. We address both requirements by introducing a dedicated **supervisor** core on every chip, distinct from the **worker** cores that execute the ACS search described in [Subsection 4.2.3](#) and [Subsection 4.2.4](#). Instead of broadcasting an improvement directly to every other worker, as in [Subsection 4.2.4](#), each worker now sends its improved tour only to the supervisor core, and notifies the supervisor of the improvement using a Regfile Interrupt ([Subsection 4.1.2](#)). The supervisor core does not itself run the ACS algorithm and can therefore remain in a low-power `__WFI()` state while waiting for such interrupts, reacting immediately once one occurs. Upon being interrupted, the supervisor determines the best tour known among its workers and broadcasts it to every worker that does not yet hold it, using the same memory-mapped read-before-write check as in [Subsection 4.2.4](#) to detect and switch to an even better tour discovered by another worker while the broadcast is still in progress. As a result, the host now only needs to read from and write to the 48 supervisor cores of a board, rather than all worker cores. This comes at the cost of removing one PE per chip from the pool of PEs performing actual tour construction; we consider this an acceptable trade-off given the substantial reduction in the number of cores the host must communicate with at each polling interval.

[Listing 13](#) shows the resulting implementation, in which `main()` dispatches to either `supervisor_main()` or `worker_main()` depending on whether the PE was selected as the chip’s supervisor. We designate the first enabled core of a chip, as identified through the mechanism of [Subsection 4.1.6](#), as the supervisor, which requires at least two enabled cores per chip so that at least one worker remains. Because the supervisor performs some initialization - resetting its per-worker tour-length table and waiting to be signaled that the host has finished uploading cities and parameters - before workers may safely begin, workers must not start computing immediately once the `FT_INT_1` upload-complete interrupt from [Subsection 4.2.3](#) is received. Instead, workers wait for Regfile Interrupt line 0, which the supervisor triggers on every worker once its own setup is complete. A Feedthrough Interrupt would have been the more natural choice for this “start” signal, since it could be broadcast to the whole chip at once; however, as discussed in [Subsection 4.1.3](#), Feedthrough Interrupts can only be triggered by the peripheral module under control of SCAMP and not by an application core such as the supervisor, so a Regfile Interrupt sent individually to every worker is used instead.

Analogously to how a worker signals the supervisor, the host signals the supervisor with a second, distinct Regfile Interrupt line whenever it has written an improved tour discovered on another chip into the supervisor’s SRAM, prompting the supervisor to propagate it to its workers in the same way as a locally discovered improvement. Because both workers and the host may attempt to interact with the supervisor’s state concurrently, access is again protected by one of the QPE’s semaphores: a worker or the host must acquire the supervisor’s semaphore

before writing its data and triggering the corresponding interrupt, and the supervisor’s interrupt handler releases the semaphore once it has consumed the update, ensuring that the semaphore is held for exactly the duration of the pending notification.

Finally, because the convergence criterion from [Subsection 4.2.3](#) was evaluated locally on each core, and because the host must in any case collect data from the chips at regular intervals for the purpose of benchmarking (as discussed in [Chapter 5](#)), we move convergence detection from the worker cores to the host. The host observes the sequence of best tour lengths reported by the supervisor cores over time and decides when to terminate the run, at which point it triggers Feedthrough Interrupt line 3 to stop every chip, exactly as in the single-PE implementation.

5 Benchmarking

This chapter describes how the implementations from [Section 4.2](#) are benchmarked. [Section 5.1](#) introduces the TSPLIB instances used throughout the evaluation, [Section 5.2](#) describes the SpiNNaker2 hardware, software, and monitoring setup, [Section 5.3](#) describes how a common ACS parameter configuration is selected, [Section 5.4](#) defines the benchmarks run for every implementation and core count, and [Section 5.5](#) introduces the single-threaded x86 reference implementation used for comparison.

The experiments evaluate whether the implementations described in [Section 4.2](#) execute correctly, how solution quality changes with increasing parallelism, and how the computational cost scales with the size of the TSP instance. The emphasis is on comparing the implementation variants rather than establishing a new state of the art for ACS. All SpiNNaker2 variants therefore use a common parameter configuration, problem representation, and stopping procedure wherever their execution models permit this.

Because ACS is stochastic, a single run does not adequately characterize an implementation. The benchmarks are repeated, and both the final solution and the progression of the search are recorded. During a run, the host periodically samples the current iteration, the iteration of the most recent improvement, and the shortest known tour from the participating PEs. These snapshots make it possible to distinguish configurations that occasionally produce a good final result from configurations that improve reliably and early.

5.1 TSP Instances

The implementations require TSP instances of different sizes to evaluate how their computational and memory requirements scale with the number of cities. The instances are taken from TSPLIB, a standard collection of benchmark problems for the TSP and related optimization problems [\[34\]](#). TSPLIB also provides reference solution lengths, allowing the quality of a discovered tour to be compared with a known result.

The representation developed in [Subsection 4.2.1](#) restricts the implementation to symmetric, two-dimensional Euclidean instances containing no more than 256 cities. Within these constraints, the selected instances cover dimensions from 51 to 226 cities:

Instance	Cities
eil51	51

Instance	Cities
berlin52	52
rd100	100
kroA150	150
kroA200	200
pr226	226

The numerical suffix of each instance name gives its number of cities. The sample therefore includes two small instances of similar size, one intermediate instance, and three larger instances approaching the implementation limit. This range permits the effects of the quadratic pheromone representation and the repeated city-selection loop to be observed without changing the algorithm itself.

The TSPLIB reference values must be interpreted with respect to the distance convention used by the implementation. The `dist()` function in Listing 8 computes a single-precision Euclidean distance directly. If a TSPLIB instance defines additional rounding for its edge-weight type, its published optimum is not necessarily the exact optimum under this unrounded metric. The reference length remains useful as a consistent quality indicator, but it is only directly comparable if both the benchmark loader and the reference use the same distance convention.

5.2 SpiNNaker2 Experimental Setup

The SpiNNaker2 experiments use the deployment on the *GWDG Future Technologies Platform* [35]. The deployment consists of four 48-node boards (Subsection 2.2.5) connected through an STM card, following the recommended organization for a production deployment [15]. All reported runs use the fourth board to avoid introducing differences between boards into the results.

PE(3, 4, 1) on chip (0, 0) of this board is treated as faulty and is disabled in every benchmark. SCAMP occupies further PEs for management tasks. Consequently, the evaluated configurations contain one application PE, 147 application PEs on one chip, or 7,103 application PEs across the complete board. For the board-scale implementation, one enabled PE per chip acts as the supervisor described in Subsection 4.2.5 and does not execute ACS. A configuration described as using one worker therefore requires a second enabled PE for supervision.

All communication between the host and the SpiNNaker2 chips passes through the STM card. The host program runs on the FTP server inside the Apptainer image `/sw/containers/spinnaker/spinnaker-users-2026-05-28-working.sif` provided by the GWDG. The container supplies the board configuration and the required SpiNNaker2 software dependencies. The version of `s2-sim2lab-app` included in this image is used.

The chip-side C programs are compiled with Clang 21.1.8 for the ARM Cortex-M4F using `--target=arm-none-eabi -mcpu=cortex-m4 -mfpu=fpv4-sp-d16 -mfloat-abi=hard -mthumb -O2` and the system libraries from Arm GNU Toolchain 15.2.Rel1, build arm-15.86. Using the same compiler options for all SpiNNaker2 variants prevents compiler configuration from becoming an additional experimental variable.

During every run, the host periodically inspects the enabled PEs. It records the current iteration, the iteration of the most recent improvement, and the shortest known tour. For the board-scale implementation, the host communicates primarily with the supervisor PEs, as described

in [Subsection 4.2.5](#). The sampling process is required to reconstruct the development of solution quality over time, but it also introduces host traffic. The same monitoring policy is therefore retained across comparable runs.

Board power is sampled on the FTP server at an interval of approximately 0.3 seconds using `snmpwalk -v2c -c public 192.168.1.3 .1` as provided by the platform administrators. The returned data includes an integer-valued estimate of instantaneous power in watts. This measurement covers the board-level system rather than an individual PE and has limited temporal and numerical resolution. It can therefore reveal large differences in board power or long-running energy trends, but it is not sufficiently fine-grained to isolate the energy cost of one ACS iteration or one NoC transfer.

5.3 Selection of ACS Parameters

The purpose of the parameter search is to obtain a common, functional ACS configuration for the implementation benchmarks. It is not intended to optimize ACS independently for every TSP instance or hardware configuration. Per-instance tuning would make it difficult to attribute differences to the implementation, because an observed improvement could result from either greater parallelism or a more favorable parameter set.

The parameters described in [Subsection 2.4.3](#) and [Subsection 4.2.2](#) are constrained to

$$\begin{aligned}
 0 &< \beta \leq 5, \\
 0 &\leq q_0 \leq 1, \\
 0 &\leq \rho \leq 1, \\
 0 &\leq \xi < 1, \\
 0 &< T_0 < 4, \text{ and} \\
 1 &\leq k \leq 40.
 \end{aligned} \tag{25}$$

Here, k denotes the number of ants in a colony. It is distinct from the migration-interval notation used in the general description of island models in [Subsection 2.4.4](#).

A stochastic hill-climbing procedure searches this bounded parameter space. It begins with a uniformly generated random configuration and evaluates that configuration using the same monitoring procedure as the subsequent benchmarks. Each following trial perturbs the best configuration found so far. Every fifth trial instead generates an entirely new random configuration, reducing the risk that the search remains confined to one local region of the parameter space. The procedure is applied to each implementation using the `berlin52`, `eil51`, and `rd100` instances.

The optimization criterion is the shortest tour found during a trial. This criterion is appropriate for rejecting configurations under which ACS does not search effectively. It does, however, favor occasional high-quality outcomes and does not by itself optimize result variance, elapsed time, or energy consumption. The repeated-run evaluation in [Chapter 6](#) therefore examines these properties separately.

5.4 Common SpiNNaker2 Benchmarks

After selecting a common parameter configuration, each implementation is evaluated with the available levels of parallelism:

1. one application worker,
2. the 147 available application PEs of one chip, and
3. the 7,103 available application PEs of the complete board.

Not every combination represents a native use of an implementation. For example, the single-PE implementation can be loaded onto many PEs, but those PEs remain independent and exchange no tours. Conversely, the board-scale implementation requires a supervisor even when only one worker performs ACS. These configurations are nevertheless useful controls because they separate the effect of executing many independent searches from the effect of sharing improvements.

A benchmark is run for each relevant combination of TSP instance, implementation, and core count. Runs are repeated to measure the variability caused by random starting cities and probabilistic city selection. The recorded snapshots support four complementary analyses:

1. the number of iterations completed for different TSP dimensions,
2. the distribution of final tour quality,
3. the distribution of the iteration count at termination, and
4. the progression of the best-known solution during representative runs.

Iteration counts are useful as an architecture-local measure of algorithmic work. They are not equivalent to wall-clock time across implementation variants. An iteration becomes more expensive as the number of cities increases, and communication, host polling, interrupt handling, and semaphore contention add costs that are not represented by the iteration counter. Runtime or speedup claims must therefore use measured elapsed time rather than iteration count alone.

5.5 Single-Threaded x86 Reference

A single-threaded x86 version of the single-PE implementation provides a reference for validating the ACS computation outside SpiNNaker2. The algorithmic data structures and update rules are retained, while host interaction through direct SRAM access and interrupts is replaced with standard input and output. The program is compiled with GCC 15.2.0 using `-O2`.

The reference system is a Dell OptiPlex 5090 containing an Intel Core i5-11500T processor and running Ubuntu 24.04.4 LTS with Linux kernel 6.8.0-137-generic. System power is observed with `powerstat`, while unrelated applications are minimized.

Only the single-PE algorithm has a directly comparable x86 implementation. The single-chip and board-scale variants depend on SpiNNaker2-specific NoC addressing, DMA, Regfile interrupts, and hardware semaphores. Replacing these mechanisms with x86 threads or processes would create a different communication architecture and would no longer isolate processor performance. The x86 program is therefore primarily a correctness and single-thread performance reference, not a substitute for the scaling experiments on SpiNNaker2.

A valid comparison requires identical city coordinates, distance semantics, random-number behavior or controlled random seeds, ACS parameters, and stopping rules. In addition, elapsed time must exclude application deployment where appropriate and must state whether host monitoring is included. Without these controls, differences cannot be attributed solely to the Cortex-M4F or x86 processor.

6 Evaluation

This chapter presents the results of the benchmarks described in [Chapter 5](#). [Section 6.1](#) first reports the parameter configuration selected for all remaining experiments. [Section 6.2](#) then examines how the implementations behave as the TSP dimension increases, before [Section 6.3](#), [Section 6.4](#), and [Section 6.5](#) focus on the `rd100` instance to examine, respectively, the consistency of final solution quality, the consistency of termination behavior, and the development of solution quality over the course of a run. [Section 6.6](#) summarizes the resulting findings.

The evaluation first selects the shared ACS parameters and then examines problem-size scaling, final solution quality, convergence behavior, and the progression of representative runs. The plots distinguish the three implementation strategies from [Subsection 4.2.3](#), [Subsection 4.2.4](#), and [Subsection 4.2.5](#).

6.1 Selected Parameter Configuration

[Figure 3](#) shows the configurations that produce the shortest tours during parameter discovery. Each subplot corresponds to one TSP instance. Rows identify the implementation variants and the number of trials considered, while columns show the six ACS parameters and the shortest tour obtained. Color intensity is normalized separately within each column of each subplot; it therefore visualizes relative values and must not be compared as an absolute scale between columns.

No single configuration dominates every instance and implementation. This is expected because ACS parameters influence the balance between exploration and exploitation, and the effective balance also changes when many independent colonies contribute solutions. Selecting separate parameters for every row would improve individual benchmark outcomes but would confound the implementation comparison.

A variation of the strongest configuration identified for `rd100` is consequently used for all subsequent runs. The evaporation parameter is increased to a nonzero value of 0.01 so that global evaporation remains active. The colony size is increased from one ant to five ants so that local pheromone updates can diversify the choices of multiple ants within each colony. The common parameters are

$$\begin{aligned}
 \beta &= 4.595, \\
 q_0 &= 0.1041, \\
 \rho &= 0.01, \\
 \xi &= 0.2612, \\
 T_0 &= 0.9677, \text{ and} \\
 k &= 5.
 \end{aligned} \tag{26}$$

These values are not claimed to be optimal. They provide a fixed and empirically functional configuration with which the implementation variants can be compared.

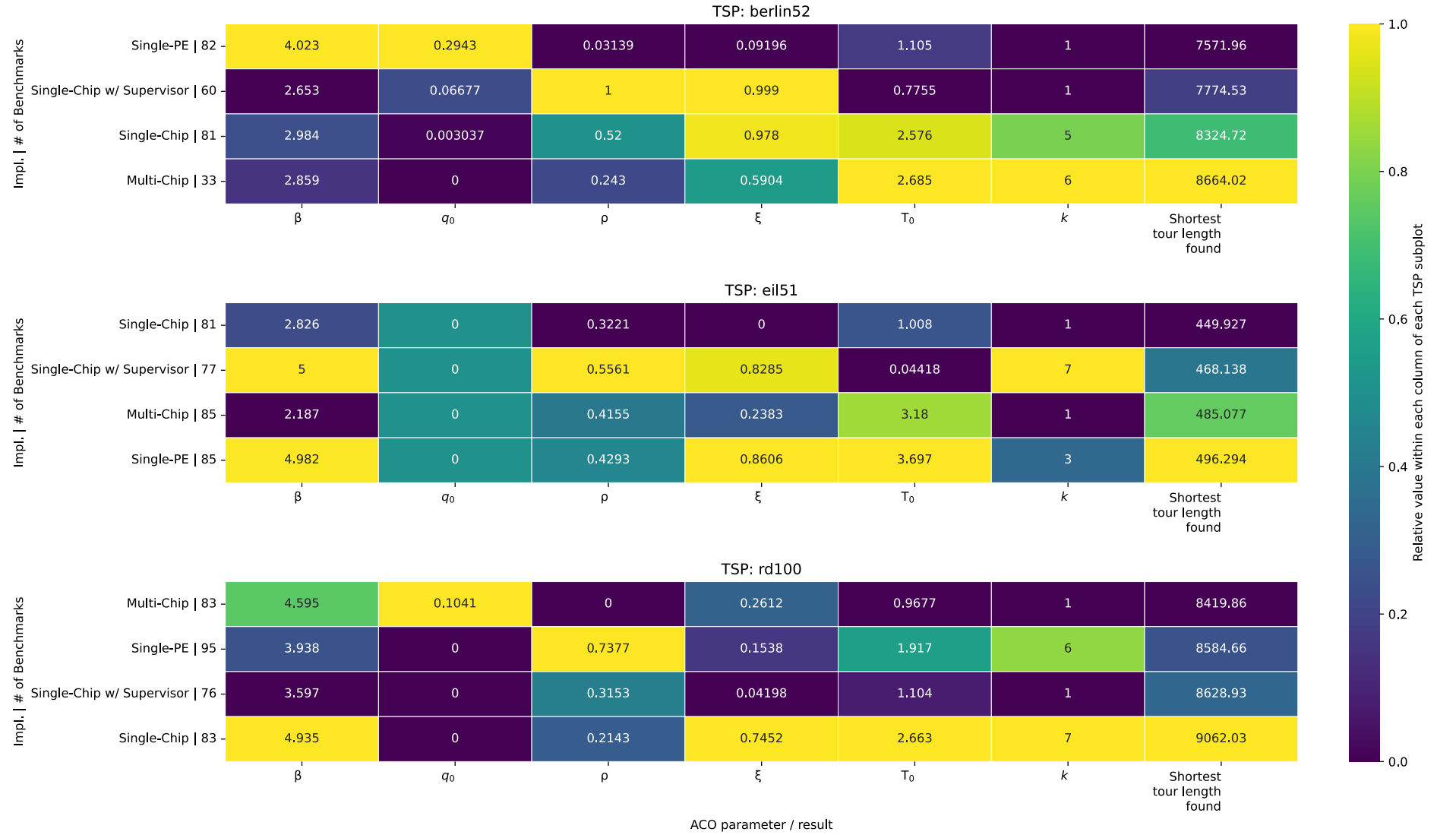


Figure 3: ACS parameter configurations yielding the shortest tours during parameter discovery for the selected TSP instances. Colors are normalized independently within each parameter column.

6.2 Scaling with TSP Dimension

The cost of one ACS iteration grows rapidly with the number of cities. For each of k ants, tour construction performs approximately $n - 1$ moves. Selecting a city may inspect up to $O(n)$ candidates, giving an approximate tour-construction complexity of $O(kn^2)$ per iteration. The triangular pheromone matrix also grows as $O(n^2)$, although the representation in [Subsection 4.2.2](#) reduces its constant factor.

[Figure 4](#) relates the TSP dimension to the maximum number of iterations completed by any participating core before the run terminates. Each point is annotated with the shortest tour found. The plot therefore combines two different quantities: iteration count indicates how long the stopping rule allows the search to continue in algorithmic steps, while the annotation indicates the quality obtained during those steps.

Most implementation and core-count combinations terminate after approximately 200 to 400 iterations for each instance. This range follows naturally from the stopping criterion: a run continues for at least 100 iterations and, for the single-PE and single-chip variants, remains active until a further interval without improvement has elapsed. The number of iterations is therefore influenced more directly by the timing of the last improvement than by the cost of the TSP instance.

The board-scale implementation with 7,103 enabled PEs produces several substantially larger iteration counts on the smaller instances. With thousands of independent colonies, improvements can continue to be discovered after a typical single colony has stagnated. Every newly shared improvement delays host-side termination. The outliers are therefore consistent with a larger aggregate search rather than necessarily indicating slower individual PEs. Their reduction for larger TSP dimensions has at least two plausible causes. First, each iteration becomes more expensive, reducing the number of iterations that can be observed within a comparable elapsed interval. Second, communication and host polling represent a smaller fraction of the work when tour construction is more expensive.

Iteration counts alone do not establish computational speedup. A 226-city iteration performs considerably more distance calculations than a 51-city iteration, and the board-scale variant additionally includes supervisor and host communication. The figure demonstrates that all selected instances fit and execute and that the stopping behavior remains broadly stable, but an elapsed-time plot is required to quantify throughput.

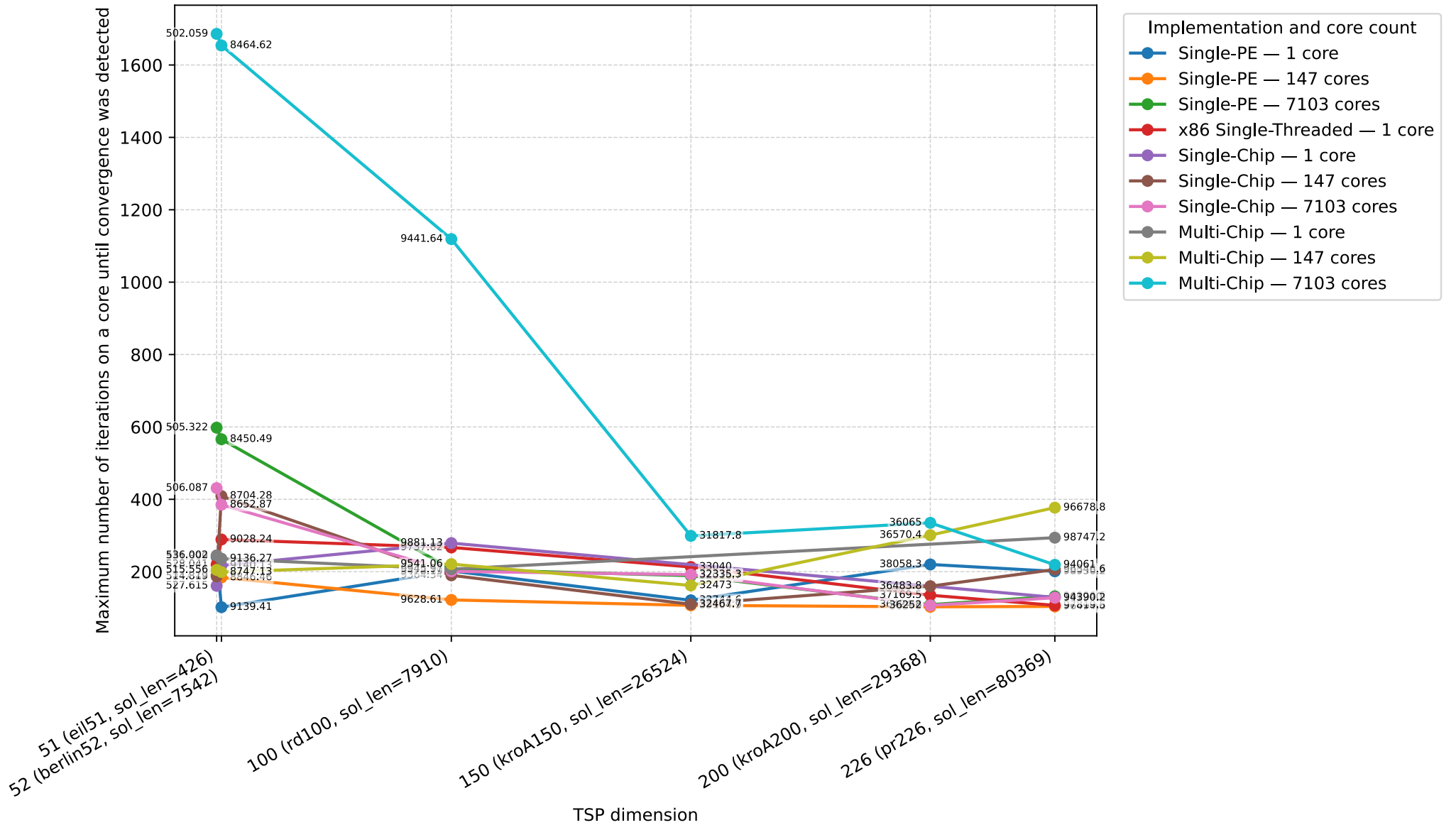


Figure 4: Maximum iteration count and shortest tour obtained as the TSP dimension increases. Iteration count measures algorithmic progress and is not a direct wall-clock runtime.

The remaining analyses focus on `rd100`. Its 100 cities are large enough to exercise the quadratic state and city-selection loops while leaving sufficient memory margin. It also lies between the smallest and largest instances and was included in parameter discovery.

6.3 Consistency of Solution Quality

[Figure 5](#) shows kernel density estimates of the shortest final tour lengths obtained across repeated `rd100` runs. Rows correspond to implementation variants and columns to the evaluated core counts. Rug marks show the individual observations. A narrow distribution close to the reference length of 7,910 indicates that a configuration repeatedly finds similar high-quality tours, whereas a broad distribution indicates greater sensitivity to random decisions.

All variants exhibit a broad distribution when only one worker executes ACS. This confirms that the stochastic search of one small colony may converge to substantially different tours in different runs. The result also demonstrates why a single benchmark run is insufficient for evaluating solution quality.

Increasing the number of workers to one chip reduces both the variance and the typical final tour length. The similarity between implementations at this scale indicates that much of this improvement results from having many independent colonies rather than from a specific communication protocol. Even independent searches allow the host to select the best result after the run, while the communicating variants additionally distribute improvements during the run.

The clearest improvement occurs for the board-scale multi-chip implementation using all available chips. Its distribution is shifted toward shorter tours and is more concentrated than the single-worker distributions. The supervisor-mediated island model therefore improves not only the best observed outcome but also the reliability with which a good outcome is obtained. This behavior is expected from parallel stochastic search: thousands of colonies explore different regions, and migration lets a strong tour influence the pheromone updates of colonies that did not discover it independently.

The improvement does not show that every participating colony becomes individually faster or more effective. It is partly a statistical consequence of drawing from many concurrent searches. The contribution of migration would have to be isolated by comparing the board-scale implementation against the same number of completely independent PEs under an identical elapsed-time budget.

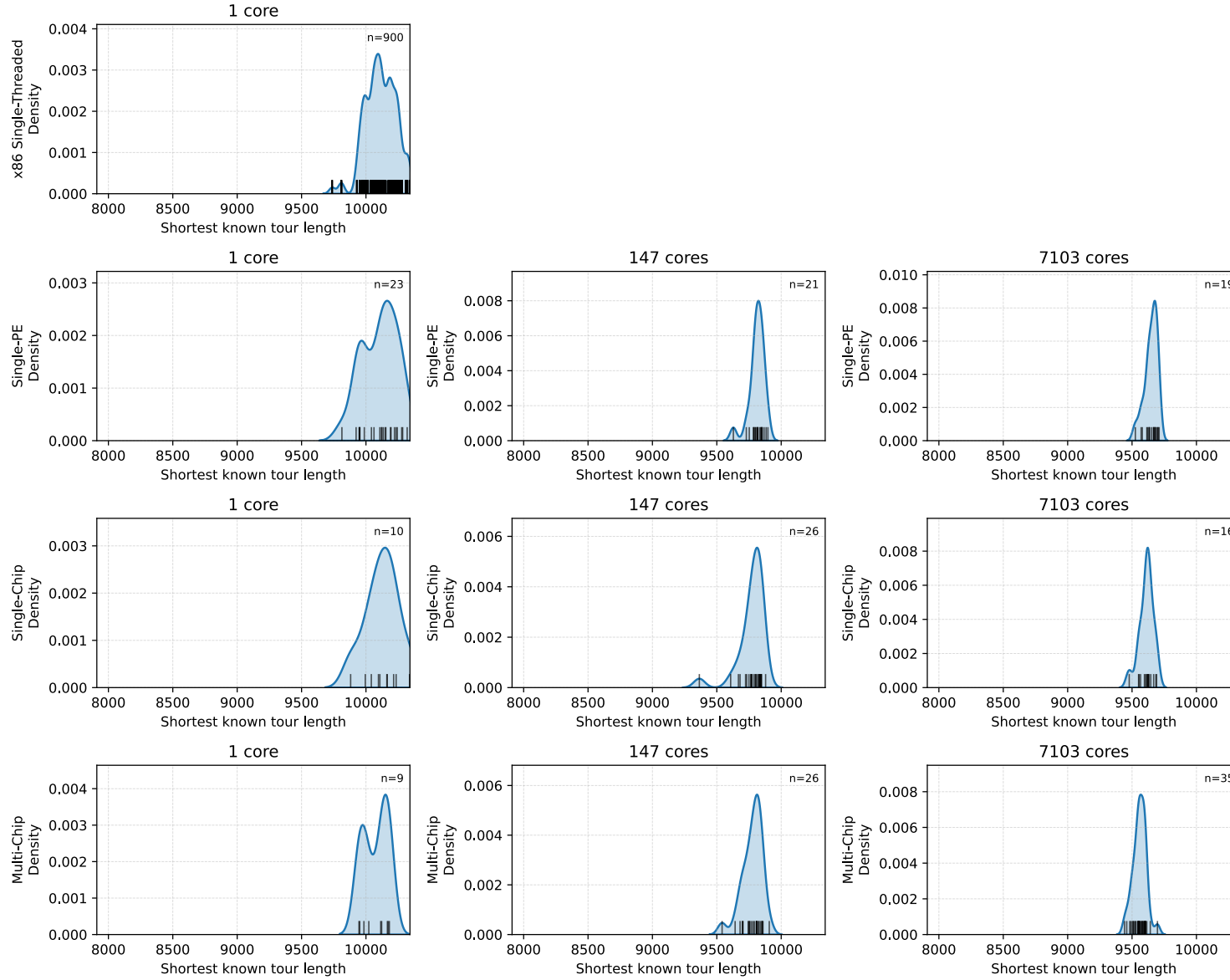


Figure 5: Distribution of the shortest tour lengths found by each implementation across repeated `rd100` benchmarks. Rug marks show the individual observations.

6.4 Consistency of Termination Behavior

[Figure 6](#) shows the distribution of the largest iteration counter reported during each `rd100` run. The layout matches [Figure 5](#). A narrow distribution indicates that improvements cease at a similar algorithmic stage in repeated runs, while a broad distribution indicates that the final improvement occurs at varying stages.

The single-worker configurations behave similarly across implementations. This is expected because their central ACS computation is derived from the same single-PE implementation and intercore communication has little or no role with one worker.

The board-scale multi-chip configuration exhibits a broader distribution and reaches larger iteration counts when more workers participate. This result is consistent with the solution-quality analysis. More colonies create more opportunities for a late improvement, and each propagated improvement can extend the host-side stopping procedure. The increased iteration count is therefore not necessarily undesirable: it accompanies the shorter and more consistent tours observed in [Figure 5](#).

The maximum over all cores is sensitive to stragglers and to variations in when the host obtains a snapshot. It also does not include the cost of an iteration. Consequently, the plot characterizes termination behavior rather than elapsed runtime. A higher maximum iteration count can indicate a longer run, but this implication is valid only when the per-iteration costs and monitoring intervals are comparable.

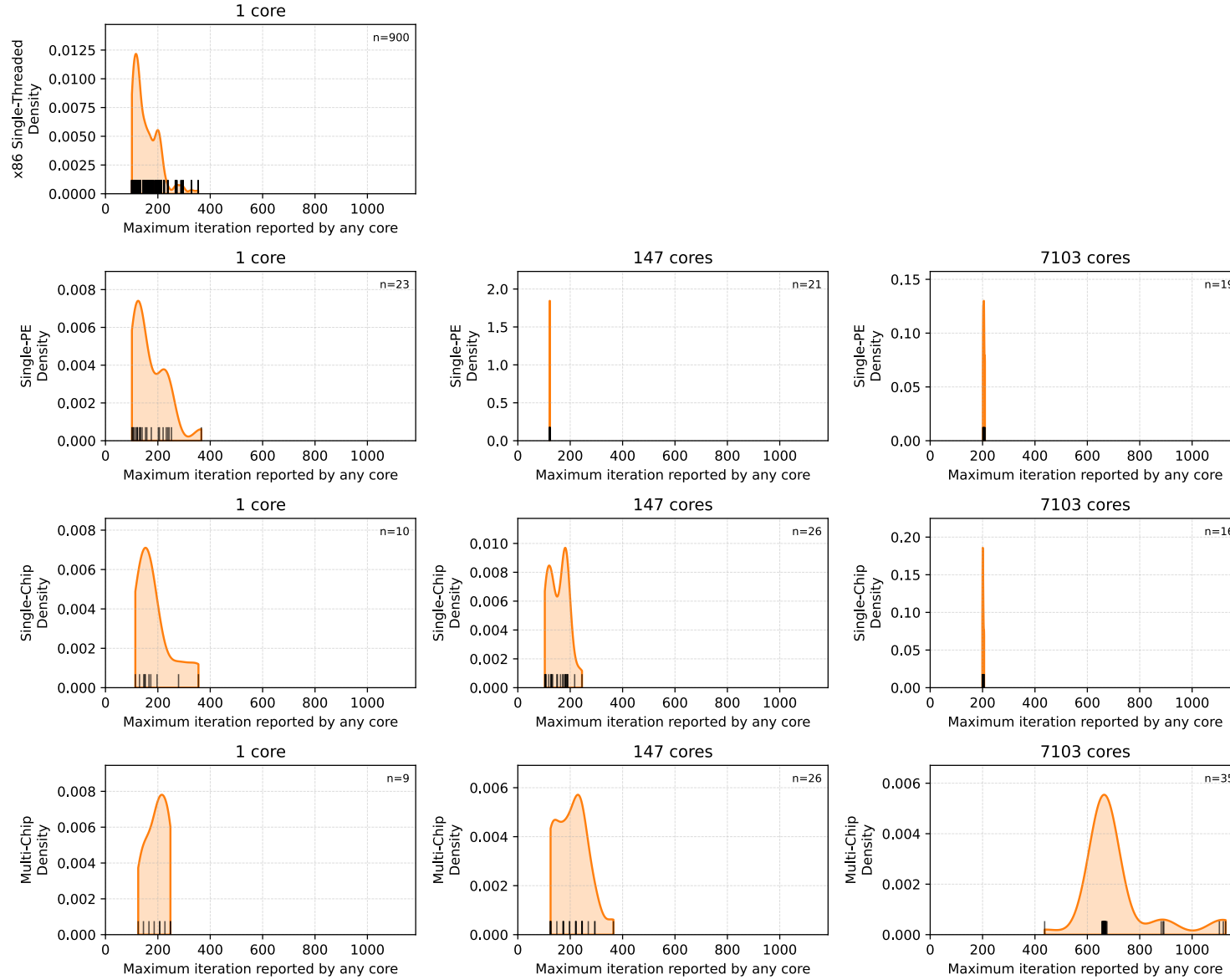


Figure 6: Distribution of the maximum iteration count reported during repeated `rd100` benchmarks. The measure characterizes stopping behavior rather than wall-clock runtime.

6.5 Development of Solution Quality

Final results do not show when a useful tour becomes available. This distinction matters for an anytime optimization algorithm: a configuration that finds a strong tour early may be preferable even if another configuration eventually reaches the same result.

[Figure 7](#) compares representative `rd100` runs. For each implementation and core-count configuration, the run whose final tour length is closest to the median of its repeated runs is selected. The horizontal axis gives the iteration count of the least-progressed active worker, and the vertical axis gives the best tour associated with the corresponding snapshot. Using the least-progressed worker provides a conservative measure of common algorithmic progress in an asynchronous system. Stepwise decreases represent newly observed improvements.

The board-scale multi-chip implementation already reports a strong solution in the first captured snapshot. That solution is better than the final result of several smaller configurations. The result demonstrates the principal benefit of large-scale parallel stochastic search: before the host performs its first poll, many colonies have already constructed tours concurrently, and the supervisor hierarchy has begun propagating the strongest discoveries.

The multi-chip curve also contains several meaningful later improvements. The initial quality is therefore not produced solely by fast startup; continued independent exploration still contributes after migration has distributed a good tour. In contrast, configurations with fewer workers improve more gradually and often terminate at a higher tour length.

The first plotted point is limited by the host sampling interval. It does not reveal exactly when the corresponding tour was found, nor whether communication had already propagated it to every worker. Likewise, the horizontal axis is not elapsed time. The figure supports the conclusion that increased parallelism improves solution quality per observed algorithmic stage, but it does not quantify a wall-clock speedup.

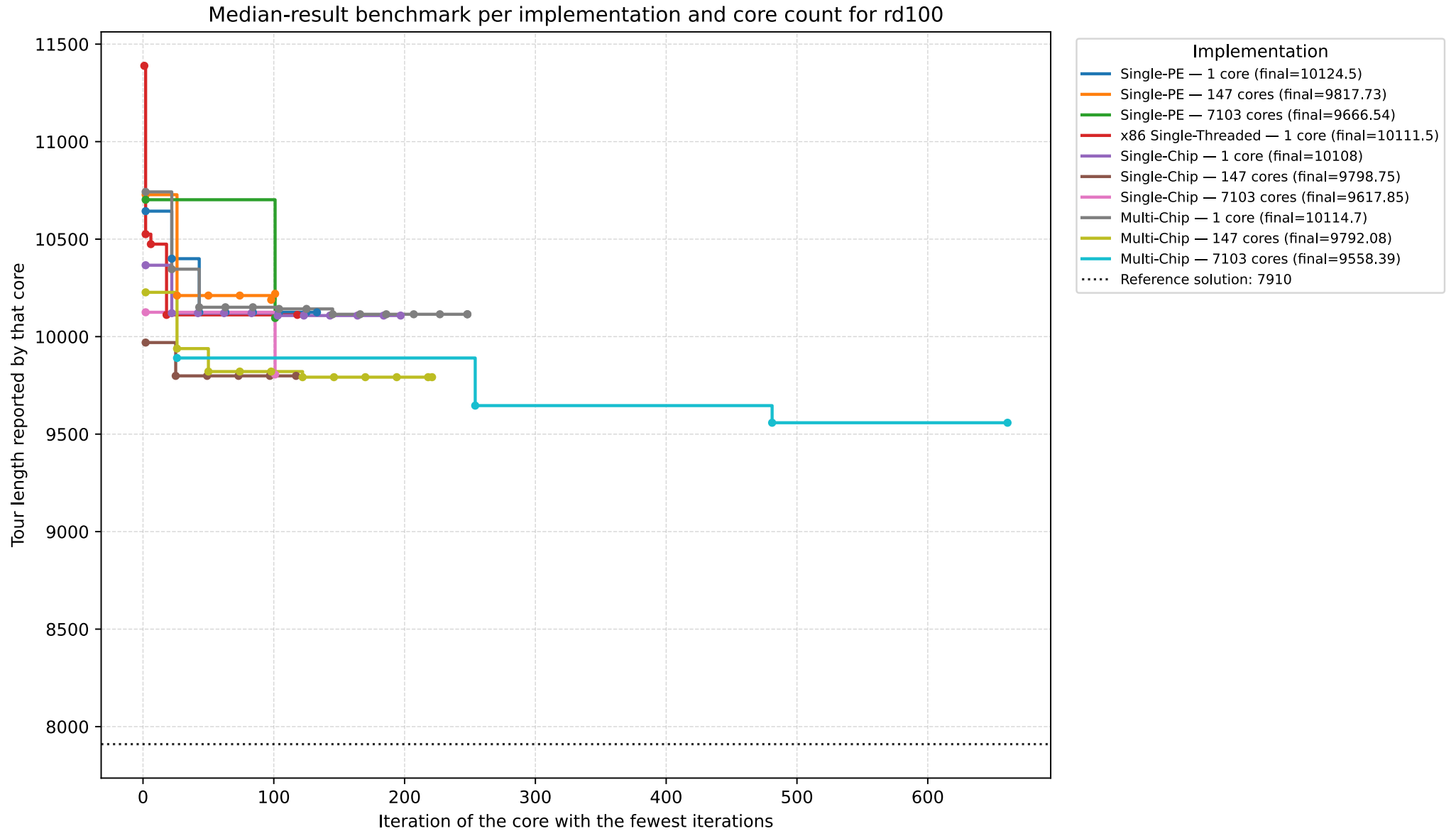


Figure 7: Development of solution quality in median-result **rd100** runs. The horizontal axis reports conservative algorithmic progress rather than elapsed time.

6.6 Summary of Experimental Results

The experiments yield four principal findings.

First, the compact representation permits ACS to execute entirely from local PE memory on all selected instances up to 226 cities. The implementation therefore avoids placing distance and pheromone accesses on the NoC in the inner city-selection loop. This validates the principal memory-design objective of [Subsection 4.2.1](#) and [Subsection 4.2.2](#).

Second, increasing the number of independently searching colonies improves final solution quality and reduces its variance. A single worker produces comparatively variable results, while one chip is more reliable. The full-board multi-chip implementation produces the strongest and most consistent `rd100` results among the evaluated configurations.

Third, the board-scale implementation continues to discover improvements for more iterations than smaller configurations. This is consistent with thousands of colonies exploring concurrently and with each improvement resetting the host-side convergence observation. The additional iterations are accompanied by improved tours rather than by stagnation alone.

Fourth, the presented iteration-based results do not establish wall-clock acceleration or energy efficiency relative to x86. Iteration counts omit the increasing per-iteration cost with TSP dimension and the communication overhead of the multi-colony variants. The available board-level power measurements are also too coarse to attribute energy to individual PEs or protocol operations. The experiments demonstrate effective parallel search and improved solution quality, but a quantitative speedup or energy-efficiency claim requires additional elapsed-time and power analysis.

7 Discussion

This chapter interprets the results of [Chapter 6](#) in light of the goals set out in [Section 1.2](#). It first assesses the general suitability of SpiNNaker2’s MIMD architecture for irregular search, then discusses the effectiveness of the memory representation from [Subsection 4.2.1](#) and [Subsection 4.2.2](#), the parallelization and communication strategy from [Subsection 4.2.4](#) and [Subsection 4.2.5](#), and the extent to which the observed solution-quality improvements can be attributed to increased parallelism versus migration between colonies. It closes by discussing the limits of the performance and energy claims supported by the experiments, correctness and reproducibility concerns, and threats to the validity of the results.

7.1 Suitability of SpiNNaker2 for Irregular Search

ACS exhibits the properties identified in [Section 2.3](#): decisions depend on random values and partial tours, pheromone and city accesses are data-dependent, and different colonies discover improvements at unpredictable times. SpiNNaker2 accommodates the control-flow component of this irregularity well. Each PE executes independently, so one colony’s choice does not force other colonies to execute the same branch. The full-board results are consistent with this architectural advantage: thousands of colonies can explore concurrently, and the aggregate search produces shorter and more consistent tours.

The results do not imply that MIMD execution removes every cost associated with irregularity. Each PE is a relatively small embedded processor, and local memory is severely constrained. The NoC offers flexible communication, but remote access is still more expensive than local SRAM access and requires explicit synchronization. SpiNNaker2 is therefore most suitable when a workload can be decomposed into largely autonomous tasks that perform substantial local computation between relatively small messages. The selected island model has precisely this structure: colonies keep their pheromone matrices local and exchange only compact tours after improvements.

A design in which all ants share one global pheromone matrix would fit the abstract ACS formulation more directly, but it would be a poor match for the hardware. It would place remote reads, writes, and synchronization inside the most frequently executed part of the algorithm. The successful design is consequently not simply a port of shared-memory ACS. It changes the parallel decomposition to preserve locality, accepting independently evolving pheromone states in exchange for much lower communication volume.

7.2 Effectiveness of the Memory Representation

The memory representation is a central contribution because the unmodified data structures do not fit into a PE's DTCM. Computing Euclidean distances from coordinates reduces distance storage from quadratic to linear. Representing only one triangular half of the symmetric pheromone matrix and encoding each entry in 16 bits reduces the largest structure to 65,792 bytes. Compact one-byte city identifiers further reduce ant state and communication volume.

These choices enable instances close to the 256-city design limit, but they also impose costs. Computing every distance on demand introduces floating-point arithmetic and a square root into the selection loop. A direct distance matrix would provide faster lookup if sufficient local memory were available. The fixed-point pheromone representation introduces quantization, and very small positive values are rounded upward rather than allowed to become zero. This preserves feasible transitions but changes the numerical behavior of ACS, particularly after many evaporation steps.

The maximum-size configuration also leaves little DTCM capacity for the stack, compiler-generated temporaries, and future extensions. The current limit is therefore a capacity boundary rather than a comfortable operating point. A change in ABI, alignment, logging, or library code can make a previously fitting binary exceed the available memory. Compile-time size assertions and linker-map checks are necessary to maintain this guarantee.

The 256-city limit follows from the one-byte city identifier rather than from an inherent limitation of ACS. Moving to 16-bit identifiers would increase tour and ant-state storage and double the size of migration messages, but it would allow larger instances if the pheromone representation were also changed. For the current design, the limit is reasonable because the pheromone matrix already approaches the available local-memory capacity.

7.3 Parallelization and Communication Strategy

The progression from independent PEs to a single-chip island model and finally to a supervisor-mediated board model illustrates the trade-off between decentralization and communication cost.

On one chip, direct tour broadcast minimizes the delay before an improvement reaches other colonies. The message contains at most 256 bytes and is small compared with a pheromone matrix. The read-before-write comparison also avoids replacing a better remote result with an inferior one. However, broadcasting from every improving PE to every other PE can generate bursts of traffic, and concurrent broadcasts require careful lock ordering. The implementation's semaphore timeouts reduce the risk that a broadcast blocks indefinitely, but they do not by themselves prove freedom from all races or deadlocks.

At board scale, the supervisor hierarchy substantially reduces host communication. The host interacts with 48 supervisors rather than more than 7,000 workers. The supervisor also reacts to local improvements independently of the host polling interval and can remain in `__WFI()` while idle. The cost is one non-computing PE per chip and a central point of serialization within each chip. If improvements occur frequently, the supervisor may spend considerable time transferring tours and processing interrupts. The design is therefore most effective once improvements become sparse relative to tour construction.

The host-mediated connection between chips is the largest architectural limitation of the evaluated system. It increases latency, makes migration dependent on polling, and prevents the board from operating as a fully autonomous distributed system. The resulting topology is hierarchical rather than a direct peer-to-peer island model. Working inter-chip routing would permit ring, mesh, or topology-aware migration without involving the host and would likely reduce both communication latency and host overhead.

7.4 Interpretation of Solution-Quality Improvements

The reduced variance and shorter tours obtained with more workers support the use of SpiNNaker2 as a platform for parallel stochastic search. Nevertheless, two effects must be separated.

The first is the *multiple-search effect*. Running more independent colonies increases the number of sampled tours, so selecting the best final result improves expected quality even without communication. The second is the *migration effect*. Sharing an improved tour changes subsequent pheromone updates on receiving colonies and can guide them toward promising regions.

The current evaluation shows the combined effect. The similarity among several one-chip configurations suggests that independent search accounts for a substantial fraction of the improvement at that scale. The stronger board-scale result is compatible with beneficial migration, but it does not isolate migration from the much larger number of workers. A controlled ablation should compare the supervisor-based implementation with 7,103 independent workers under equal wall-clock, iteration, random-seed, and host-monitoring budgets.

Migration can also reduce diversity. If every colony immediately reinforces the same globally best tour, their pheromone matrices may become increasingly similar and converge prematurely. The observed later improvements indicate that sufficient diversity remains in the evaluated configuration, but the fixed migration-on-every-improvement policy is not necessarily optimal. Delayed, probabilistic, neighborhood-based, or quality-threshold migration may preserve diversity while reducing communication.

7.5 Performance and Energy Claims

The experiments establish increased search breadth and improved solution quality, but they do not by themselves demonstrate acceleration in the conventional sense of reduced wall-clock time. An ACS iteration is not a constant unit of time across TSP dimensions or implementation variants. Its cost depends on the number of cities and ants, while the communicating variants additionally perform DMA, semaphore, interrupt, and host operations.

The maximum iteration count is especially unsuitable as a runtime substitute on thousands of asynchronous PEs. It is an extreme statistic and may be determined by one advanced worker, one delayed snapshot, or a late improvement. Likewise, the least-progressed worker used in [Figure 7](#) is useful for comparing common algorithmic progress but is not a clock.

A defensible speedup analysis requires elapsed-time measurements for clearly defined phases:

1. deployment and initialization,
2. host-to-chip parameter transfer,
3. ACS execution,
4. intercolony communication,
5. result retrieval, and
6. complete end-to-end execution.

Both execution-only and end-to-end results are relevant. Execution-only timing characterizes the processor and algorithm, while end-to-end timing captures the practical cost experienced by a user.

The same distinction applies to energy. The SNMP measurement reports integer-valued board power at approximately 0.3-second intervals and includes components not directly executing ACS. It is suitable for estimating board-level energy over sufficiently long runs by integrating power over elapsed time. It is not suitable for attributing energy to a core, DMA transfer, or interrupt. Comparing this result with `powerstat` on x86 additionally requires careful definition of the measured system boundaries. No claim that SpiNNaker2 consumes less energy than the x86 system is justified without such normalization.

7.6 Correctness and Reproducibility

The x86 port provides a useful independent execution environment, but numerical equivalence requires more than compiling similar C code. The ARM Cortex-M4F and x86 processor may differ in floating-point evaluation, random-number generation, and library implementation. Because ACS is stochastic, small numerical differences can alter a city choice and produce entirely different later tours. Correctness should therefore be tested at the level of invariants and controlled steps rather than by expecting complete stochastic runs to produce identical tours.

Important invariants include the following:

1. every completed tour contains each city exactly once;
2. the stored tour length equals a separately recomputed length;
3. encoded pheromone values remain in their allowed range;
4. the triangular index maps (i, j) and (j, i) to the same entry;
5. received tours are complete before their lengths become visible; and
6. global updates use the intended best tour.

There is also a distinction between iteration-best and global-best updating. Canonical ACS in [Subsection 2.4.3](#) reinforces the global-best tour. The single-chip and board variants use `shortest_known_tour`, including received improvements, whereas the single-PE listing passes the selected ant directly to the global update. This difference can affect convergence independently of communication. A fair implementation comparison should use the same update policy in every variant or explicitly treat the policy as an experimental factor.

Reproducibility further requires recording random seeds, compiler and library revisions, the exact enabled-core set, monitoring intervals, parameter files, and the commit used to generate each binary. The repository referenced in [Chapter B](#) is an appropriate place to preserve these artifacts.

7.7 Threats to Validity

Several limitations constrain the generality of the findings.

The evaluation uses one optimization algorithm and six symmetric Euclidean TSP instances containing at most 226 cities. Other irregular algorithms may have larger per-task states, less independent work, or more frequent communication. The results therefore demonstrate feasibility for one representative workload rather than universal suitability for irregular computing.

The common parameter configuration is derived partly from `rd100`, which is also used for the detailed analysis. This makes `rd100` a useful representative case but introduces a risk of parameter-selection bias. The use of the same parameters on the other instances limits overfitting, yet a separate validation set would provide stronger evidence.

Kernel density estimates can visually imply more certainty than the underlying number of repeated runs supports. Rug marks expose the individual observations, but medians, quantiles, sample counts, and confidence intervals should accompany the plots. Random seeds should also be paired across comparable configurations where possible.

The convergence rules differ between implementation stages. Single-PE and single-chip workers perform local convergence detection, whereas the board-scale design delegates termination to the host. This difference can alter both runtime and final quality. A common wall-clock budget or a common total number of constructed tours would provide a more direct comparison.

Host monitoring is part of the experimental system. Polling thousands of PEs can perturb NoC and host-link traffic, while polling only supervisors changes that cost. The monitoring interval also determines when improvements become visible in the recorded traces. Consequently, the benchmark framework must be considered part of the evaluated implementation rather than a perfectly passive observer.

Finally, the use of direct Euclidean floating-point distances must match the TSPLIB edge-weight convention before deviations from published reference lengths are interpreted as optimality gaps. Otherwise, the reference value is a useful landmark but not an exact objective value for the implemented metric.

8 Conclusions

This thesis investigates whether SpiNNaker2 can execute and accelerate an irregular, non-neural optimization workload. ACS for the TSP serves as the case study because it combines data-dependent control flow, irregular memory access, stochastic search, and asynchronous information exchange. The resulting implementation demonstrates that SpiNNaker2 can support this workload, provided that the algorithm is reorganized around local memory and coarse-grained communication.

The first goal is to establish a usable environment for general-purpose applications. This goal is achieved through *s2generic*. The environment provides host-side deployment and core selection, typed access to chip-side C variables, enabled-core discovery, logging, and abstractions for Regfile interrupts, Feedthrough interrupts, remote SRAM access, DMA, and hardware semaphores. These facilities make it possible to implement applications that do not follow the spiking-neural-network abstractions of the existing ecosystem.

The second goal is to design an ACS implementation compatible with the memory constraints of a PE. This goal is achieved by storing coordinates instead of a complete distance matrix, using one-byte city identifiers, representing symmetric pheromone trails as a triangular array, and encoding pheromone values with 16-bit fixed-point numbers. The resulting state fits in local DTCM for the supported instances and avoids non-local memory accesses in the inner selection loop. The implementation supports symmetric two-dimensional TSP instances containing up to 256 cities.

The third goal is to exploit parallelism across PEs and chips. On one chip, each PE executes an independent colony and broadcasts compact improved tours. Across the 48-chip board, one supervisor per chip aggregates worker improvements, distributes received tours locally, and provides one communication endpoint for the host. This hierarchy scales the search to the available application PEs despite the absence of direct inter-chip communication. It uses SpiNNaker2 as a distributed-memory MIMD system rather than as a neuromorphic accelerator.

The fourth goal is to evaluate scalability and solution quality. The experiments show that increasing the number of colonies reduces the variability of final tour quality and generally produces shorter tours. The full-board multi-chip implementation gives the strongest and most consistent results for the detailed *rd100* evaluation and discovers useful tours early in the recorded search. The evaluation therefore supports the claim that SpiNNaker2 can provide effective large-scale parallelism for this form of stochastic irregular search.

The evidence is more limited with respect to conventional performance and energy claims. Iteration counts characterize algorithmic progress but do not measure elapsed runtime, and the available board-level power samples do not isolate application energy. The x86 port supplies a reference environment, but a quantitative speedup and energy comparison requires stricter control of timing boundaries, distance semantics, random seeds, update policies, and system-level power measurements. Accordingly, the principal demonstrated benefit is improved search coverage and solution quality through massive parallelism, not a proven wall-clock or energy advantage over x86 or GPU systems.

Overall, SpiNNaker2 is promising for irregular algorithms whose state can be partitioned across autonomous workers and whose communication can be expressed as small, infrequent messages. Its independent instruction streams avoid SIMD branch-divergence penalties, while its event and packet mechanisms support asynchronous coordination. Its principal limitations for general-

purpose use are the small local memory, explicit synchronization requirements, immature generic software support, and the lack of functioning direct inter-chip communication in the evaluated deployment. The effectiveness of the architecture therefore depends as much on data representation and communication design as on the available number of PEs.

8.1 Future Work

The most immediate future work is a complete wall-clock and energy evaluation. Execution-only and end-to-end timings should be measured separately for the SpiNNaker2 and x86 implementations. Board energy should be calculated by integrating power over the same clearly defined execution interval, with idle power reported separately. A GPU implementation using equivalent ACS semantics would provide an additional comparison between MIMD and SIMT execution.

The implementation should next be validated with deterministic unit and integration tests. These tests should verify tour permutations, independently recompute tour lengths, test every triangular pheromone index, and compare fixed-point updates with a floating-point reference. The exact benchmark binaries should be checked to ensure that shortest-ant selection uses the correct comparison and that every implementation applies the same global-best or iteration-best update policy. Controlled random seeds would then permit paired comparisons across architectures and implementation variants.

Direct inter-chip communication through the SpiNNaker2 Router would remove the host from the migration path. With this facility, supervisors could communicate in a ring, mesh, or topology-aware neighborhood. This would reduce host traffic and make it possible to compare synchronous and asynchronous migration without polling. If direct routing remains unavailable, an event-driven host forwarding mechanism could reduce the latency of the current periodic pull-and-forward approach.

The migration policy itself offers several experimental parameters. Future implementations could vary the migration interval, suppress small improvements, communicate only with neighboring chips, or probabilistically accept incoming tours. These policies may reduce communication and preserve diversity. An ablation study should compare no migration, chip-local migration, and board-wide migration using the same number of workers and the same elapsed-time budget.

The local-memory design can also be extended. Bit-packing the visited sets would recover memory, while caching selected distances or candidate lists could reduce repeated square-root calculations. Alternative pheromone encodings could provide a wider dynamic range, for example through logarithmic or block-floating-point representations. For larger instances, the city identifier could be widened to 16 bits and the pheromone matrix could be sparsified by storing only candidate edges.

The SpiNNaker2 Machine Learning Accelerator may be investigated for batched distance or transition-weight calculations where the computation can be reorganized into suitable vector or matrix operations. Such use would not eliminate the irregular city choices, but it could accelerate regular subcomputations within each selection step.

Finally, `s2generic` should be developed beyond the requirements of ACS. Useful additions include documented timers and random-number facilities, topology-aware placement, safer lock abstractions, automated DTCM-size checks, structured inter-PE messaging, and systematic

fault handling. Applying the environment to graph traversal, discrete-event simulation, branch-and-bound search, and other irregular workloads would determine which conclusions of this thesis generalize beyond ant-colony optimization.

A Bibliography

- [1] M. Garland *et al.*, “Parallel Computing Experiences with CUDA,” *IEEE Micro*, vol. 28, no. 4, pp. 13–27, Jul. 2008, doi: [10.1109/MM.2008.57](https://doi.org/10.1109/MM.2008.57).
- [2] H. A. Gonzalez *et al.*, “SpiNNaker2: A Large-Scale Neuromorphic System for Event-Based and Asynchronous Machine Learning.” Accessed: Jul. 11, 2026. [Online]. Available: <http://arxiv.org/abs/2401.04491>
- [3] M. Flynn, “Very High-Speed Computing Systems,” *Proceedings of the IEEE*, vol. 54, no. 12, pp. 1901–1909, Dec. 1966, doi: [10.1109/PROC.1966.5273](https://doi.org/10.1109/PROC.1966.5273).
- [4] “DMA - SpiNNaker2 Developer Portal.” Accessed: Jul. 11, 2026. [Online]. Available: <https://spinnaker2.gitlab.io/external/documentation/hardware/7-dma/>
- [5] “Chip Topology - SpiNNaker2 Developer Portal.” Accessed: Jul. 11, 2026. [Online]. Available: <https://spinnaker2.gitlab.io/external/documentation/hardware/2-s2-chip-topology/>
- [6] “SpiNNaker Router - SpiNNaker2 Developer Portal.” Accessed: Jul. 11, 2026. [Online]. Available: <https://spinnaker2.gitlab.io/external/documentation/hardware/8-spinnaker-router/>
- [7] “SDRAM Interface - SpiNNaker2 Developer Portal.” Accessed: Aug. 11, 2026. [Online]. Available: <https://spinnaker2.gitlab.io/external/documentation/hardware/4-dram/>
- [8] “Cortex-M4 Technical Reference Manual R0p0.” Accessed: Aug. 11, 2026. [Online]. Available: <https://support.arm.com/documentation/ddi0439/b/Floating-Point-Unit/About-the-FPU>
- [9] “Random Number Generator - SpiNNaker2 Developer Portal.” Accessed: Jul. 11, 2026. [Online]. Available: <https://spinnaker2.gitlab.io/external/documentation/hardware/12-RNG/>
- [10] “ML Accelerator - SpiNNaker2 Developer Portal.” Accessed: Jul. 11, 2026. [Online]. Available: <https://spinnaker2.gitlab.io/external/documentation/hardware/1-ml-accelerator/>
- [11] “Running Multiple PEs - SpiNNaker2 Developer Portal.” Accessed: Jul. 11, 2026. [Online]. Available: <https://spinnaker2.gitlab.io/external/documentation/hardware/3-running-multiple-pes/>
- [12] “NoC - SpiNNaker2 Developer Portal.” Accessed: Jul. 11, 2026. [Online]. Available: <https://spinnaker2.gitlab.io/external/documentation/hardware/11-NoC/>
- [13] “Regfile Access - SpiNNaker2 Developer Portal.” Accessed: Jul. 11, 2026. [Online]. Available: <https://spinnaker2.gitlab.io/external/documentation/hardware/3-reg-file-access/>
- [14] “Marco Stolba / S2_rf_html · GitLab.” Accessed: Aug. 11, 2026. [Online]. Available: https://gitlab.com/mstolba/s2_rf_html
- [15] “48-Node Board - SpiNNaker2 Developer Portal.” Accessed: Jul. 11, 2026. [Online]. Available: <https://spinnaker2.gitlab.io/external/documentation/hardware/7-48-node-board/>
- [16] F. Negri *et al.*, “Py-Spinnaker2.” Accessed: Jul. 11, 2026. [Online]. Available: <https://zenodo.org/doi/10.5281/zenodo.10202109>
- [17] “Software - SpiNNaker2 Developer Portal.” Accessed: Jul. 11, 2026. [Online]. Available: <https://spinnaker2.gitlab.io/external/documentation/software/>

- [18] J. M. Cecilia, J. M. García, A. Nisbet, M. Amos, and M. Ujaldón, “Enhancing Data Parallelism for Ant Colony Optimization on GPUs,” *Journal of Parallel and Distributed Computing*, vol. 73, no. 1, pp. 42–51, Jan. 2013, doi: [10.1016/j.jpdc.2012.01.002](https://doi.org/10.1016/j.jpdc.2012.01.002).
- [19] Y. Zhou, F. He, N. Hou, and Y. Qiu, “Parallel Ant Colony Optimization on Multi-Core SIMD CPUs,” *Future Generation Computer Systems*, vol. 79, pp. 473–487, Feb. 2018, doi: [10.1016/j.future.2017.09.073](https://doi.org/10.1016/j.future.2017.09.073).
- [20] M. Dorigo, V. Maniezzo, and A. Coloni, “Ant System: Optimization by a Colony of Cooperating Agents,” *IEEE Transactions on Systems, Man, and Cybernetics, Part B (Cybernetics)*, vol. 26, no. 1, pp. 29–41, Feb. 1996, doi: [10.1109/3477.484436](https://doi.org/10.1109/3477.484436).
- [21] M. Dorigo and L. M. Gambardella, “Ant Colonies for the Travelling Salesman Problem,” *Biosystems*, vol. 43, no. 2, pp. 73–81, Jul. 1997, doi: [10.1016/S0303-2647\(97\)01708-5](https://doi.org/10.1016/S0303-2647(97)01708-5).
- [22] A. M. Mora, P. García-Sánchez, J. J. Merelo, and P. A. Castillo, “Pareto-Based Multi-Colony Multi-Objective Ant Colony Optimization Algorithms: An Island Model Proposal,” *Soft Computing*, vol. 17, no. 7, pp. 1175–1207, Jul. 2013, doi: [10.1007/s00500-013-0993-y](https://doi.org/10.1007/s00500-013-0993-y).
- [23] M. Craus and L. Rudeanu, “Parallel Framework for Ant-like Algorithms,” in *Third International Symposium on Parallel and Distributed Computing/Third International Workshop on Algorithms, Models and Tools for Parallel Computing on Heterogeneous Networks*, Jul. 2004, pp. 36–41. doi: [10.1109/ISPDC.2004.37](https://doi.org/10.1109/ISPDC.2004.37).
- [24] G. Shobaki *et al.*, “Instruction Scheduling for the GPU on the GPU,” in *2024 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*, Mar. 2024, pp. 435–447. doi: [10.1109/CGO57630.2024.10444869](https://doi.org/10.1109/CGO57630.2024.10444869).
- [25] Y. Zhao, C. Zhao, and Y. Liu, “FPGA-Based Hardware Modeling on Pigeon-Inspired Optimization Algorithm,” in *Methods and Applications for Modeling and Simulation of Complex Systems*, W. Fan, L. Zhang, N. Li, and X. Song, Eds., Singapore: Springer Nature, 2022, pp. 407–423. doi: [10.1007/978-981-19-9198-1_31](https://doi.org/10.1007/978-981-19-9198-1_31).
- [26] B. Scheuermann *et al.*, “FPGA Implementation of Population-Based Ant Colony Optimization,” *Applied Soft Computing*, vol. 4, no. 3, pp. 303–322, Aug. 2004, doi: [10.1016/j.asoc.2004.03.008](https://doi.org/10.1016/j.asoc.2004.03.008).
- [27] S. Zhang, Z. Huang, W. Wang, R. Tian, and J. He, “HACO-F: An Accelerating HLS-Based Floating-Point Ant Colony Optimization Algorithm on FPGA,” *International Journal of Performability Engineering*, vol. 13, no. 6, p. 854, Oct. 2017, doi: [10.23940/ijpe.17.06.p7.854863](https://doi.org/10.23940/ijpe.17.06.p7.854863).
- [28] C. Mayr, S. Hoepfner, and S. Furber, “SpiNNaker 2: A 10 Million Core Processor System for Brain Simulation and Machine Learning.” Accessed: Jul. 11, 2026. [Online]. Available: <http://arxiv.org/abs/1911.02385>
- [29] G. Verma, S. Soni, and B. K. Kaushik, “Spin Device-Based Image Edge Detection Architecture for Neuromorphic Computing,” *Nanotechnology*, vol. 35, no. 5, p. 55201, Nov. 2023, doi: [10.1088/1361-6528/ad0056](https://doi.org/10.1088/1361-6528/ad0056).
- [30] “Deploying an Application - SpiNNMan2.” Accessed: Jul. 11, 2026. [Online]. Available: https://spinnman2-spinnaker2-cfed5092d1bac452b1096ec19e2826cf9bbd0eade7.gitlab.io/spinnman_documentation/deploying_an_application/

- [31] “Unaligned Memory Access Behavior in ARM Cortex-M Processors - System on Chips.” Accessed: Aug. 04, 2026. [Online]. Available: <https://www.systemonchips.com/unaligned-memory-access-behavior-in-arm-cortex-m-processors/>
- [32] “Cortex-M4 Technical Reference Manual R0p0.” Accessed: Aug. 11, 2026. [Online]. Available: <https://support.arm.com/documentation/ddi0439/b/Nested-Vectored-Interrupt-Controller/NVIC-functional-description/Low-power-modes>
- [33] “Cortex-M4 Technical Reference Manual R0p0.” Accessed: Aug. 11, 2026. [Online]. Available: <https://support.arm.com/documentation/ddi0439/b/Nested-Vectored-Interrupt-Controller/NVIC-functional-description/Level-versus-pulse-interrupts>
- [34] “TSPLIB-A Traveling Salesman Problem Library,” *ORSA Journal on Computing*, Accessed: Aug. 11, 2026. [Online]. Available: <https://pubsonline.informs.org/doi/abs/10.1287/ijoc.3.4.376>
- [35] “SpiNNaker :.” Accessed: Jul. 11, 2026. [Online]. Available: <https://docs.hpc.gwdg.de/services/ftp/neuromorphic-computing/spinnaker/index.html>

B Source Code

The full source code of

- the *s2generic* SDK and library developed in [Section 4.1](#),
- the ACS implementations developed in [Section 4.2](#),
- the benchmarking methods used for [Chapter 5](#),
- as well as the raw data of the performed benchmarks, and
- the code for evaluating and plotting the benchmark data in [Chapter 6](#)

can be found in the Git Repository at:

<https://gitlab.gwdg.de/jake-uni-stuff/bachelor/experimenting>

Note that some the source code in that repo differs from the listings shown in this thesis, to improve comprehension for the reader in the context of this thesis. Further information on the hardware and software used for benchmarking can also be found in that repo.

The Cake Is A Lie!

```

1  #include "common.h" // s2generic
2
3  #include "aco.h" // ACS and TSP related types, functions, and variables
4
5  // minimum iterations to run
6  #define MIN_ITERATIONS 100
7  // maximum iterations to run after the last improvement was found
8  #define MAX_ITERATIONS_WITHOUT_IMPROVEMENT 100
9
10 volatile bool4 running = false; // uint32_t
11 volatile uint32_t last_improvement_iteration = 0;
12 volatile uint32_t iteration = 0;
13
14 // FT_INT_1 is triggered by the host, when all cities and parameters have been
   uploaded
15 void FT_INT_1_Handler(void) {
16     rf_disable_feedthrough_interrupt_lines(FT_INT_1_Line); // disable handler
17     running = true;
18 }
19
20 // FT_INT_3 is triggered by the host, when the core should end the program
   early
21 void FT_INT_3_Handler(void) {
22     rf_disable_feedthrough_interrupt_lines(FT_INT_3_Line); // disable handler
23     running = false;
24 }
25
26 int main() {
27     // subscribe to feedthrough interrupts we want to use
28     rf_subscribe_feedthrough_interrupts(FT_INT_1_Line //

```

```

29         | FT_INT_3_Line, //
30         (1UL << __NVIC_PRI0_BITS) - 2UL, true);
31
32     // configure semaphores
33     configure_qpe_semaphore_0(my_qpe_x, my_qpe_y, 1);
34     configure_qpe_semaphore_1(my_qpe_x, my_qpe_y, 1);
35     configure_qpe_semaphore_2(my_qpe_x, my_qpe_y, 1);
36     configure_qpe_semaphore_3(my_qpe_x, my_qpe_y, 1);
37
38     // wait for FT_INT_1, indicating cities have been uploaded
39     while (!running) {
40         __WFI(); // wait-for-interrupt (low power until interrupt occurs)
41     }
42
43     assert_aco_parameters();
44
45     shortest_known_tour_length = INFINITY;
46     iteration = 0;
47     last_improvement_iteration = 0;
48
49     reset_edges(TAU0); // set all edges of pheromone trail to TAU0
50
51     // main loop
52     while (running) {
53         // reset ants with random starting cities
54         for (uint32_t i = 0; i < NUM_ANTS && running; i++) {
55             ant_t *ant = &ants[i];
56             reset_ant(ant, random_city());
57         }
58
59         // process ants until all ants have a completed tour
60         for (uint32_t cities_visited = 0; cities_visited < NUM_CITIES - 1 &&
             running; cities_visited++) {
61             for (uint32_t i = 0; i < NUM_ANTS && running; i++) {
62                 ant_t *ant = &ants[i];
63                 city_id_t r = current_city(ant);
64                 city_id_t s = choose_city_s(ant);
65                 travel_to_city(ant, s);
66                 perform_local_trail Updating(r, s);
67             }
68         }
69
70         // determine shortest tour
71         ant_t *best_ant = &ants[0];
72         for (uint32_t i = 0; i < NUM_ANTS && running; i++) {
73             ant_t *ant = &ants[i];

```

```

74         if (ant->tour_length >= best_ant->tour_length) {
75             best_ant = ant;
76         }
77     }
78
79     if (!running)
80         break; // host requested exit, but didn't complete the iteration.
                data is worthless
81
82     // global trail updating of pheromones
83     perform_global_trail_updating(best_ant->tour, best_ant->tour_length);
84
85     acquire_qpe_semaphore((semaphore_t)my_qpe_pe, my_qpe_x, my_qpe_y,
        UINT32_MAX); // prevent race conditions, when host accesses
        shortest_known_tour while running
86
87     // update shortest_known_tour
88     if (best_ant->tour_length < shortest_known_tour_length) {
89         memcpy((void *)shortest_known_tour, best_ant->tour,
90             sizeof(city_id_t) * NUM_CITIES);
91         shortest_known_tour_length = best_ant->tour_length;
92         last_improvement_iteration = iteration;
93     }
94
95     iteration++;
96
97     release_qpe_semaphore((semaphore_t)my_qpe_pe, my_qpe_x, my_qpe_y);
98
99     // detect algorithm convergence
100    // (in theory better tours could still be found, but the algorithm is
        stochastic so we have to finish at some point)
101    if (iteration >= MIN_ITERATIONS && iteration -
        last_improvement_iteration > MAX_ITERATIONS_WITHOUT_IMPROVEMENT) {
102        running = false;
103    }
104
105    return (int)0xcafebabe; // application finished successfully
106 }

```

Listing 11: ACS Implementation for running on a single PE. The included `aco.h` contains the code discussed in [Subsection 4.2.1](#) and [Subsection 4.2.2](#) in addition to some further related helper functions and global variable declarations.

```

1 // [...unchanged... ]
2
3 volatile float check_shortest_known_tour_length = INFINITY;
4

```

C

```

5  int main() {
6
7      // [...beginning of main is unchanged...]
8
9      check_shortest_known_tour_length = INFINITY;
10
11     reset_edges(TAU0);
12
13     // main loop
14     while (running) {
15
16         // [...beginning of main loop is unchanged...]
17
18         // ['perform_global_trail Updating(best_ant->tour, best_ant-
19         >tour_length);' has been removed, everything after this point is
20         changed]
21
22         acquire_qpe_semaphore((semaphore_t)my_qpe_pe, my_qpe_x, my_qpe_y,
23         UINT32_MAX); // prevent race conditions, when host or other cores access
24         shortest_known_tour
25
26         // update shortest_known_tour
27         if (best_ant->tour_length < shortest_known_tour_length) {
28             memcpy((void *)shortest_known_tour, best_ant->tour,
29             sizeof(city_id_t) * NUM_CITIES);
30             shortest_known_tour_length = best_ant->tour_length;
31
32             // tell other cores about this improvement
33             bool desire_break = false;
34             FOR_EACH_ENABLED_CORE(core_id) { // iterate over each enabled core
35                 if (core_id == my_core_id)
36                     continue; // no need to inform ourselves
37                 uint8_t qpe_x, qpe_y, qpe_pe;
38                 UNPACK_COREID_INT0(core_id, qpe_x, qpe_y, qpe_pe); // helper
39                 macro in s2generic
40                 if (!acquire_qpe_semaphore((semaphore_t)qpe_pe, qpe_x, qpe_y,
41                 100000)) { // try to acquire semaphore of other core with up to
42                 100000 times
43                     printf("D%d", (int)iteration); // log deadlock detection
44                     goto cleanup_loop; // continue to next core
45                 }
46                 // get pointer to memory-mapped location of
47                 shortest_known_tour_length on other core
48                 float *remote_tour_length = (float
49                 *)get_global_addr_pe_mem(qpe_x, qpe_y, qpe_pe,
50                 (uint32_t)&shortest_known_tour_length);
51                 if (*remote_tour_length > shortest_known_tour_length) { //
52                 compare remote to local shortest_known_tour_length

```

```

41         // Detected that we have a better tour.
42         // Send out shortest_known_tour to the other core.
43         dma_write_variable_to_other_core(core_id, (void
44         *)&shortest_known_tour, pad_to_divisible_by_4(NUM_CITIES));
45         *remote_tour_length = shortest_known_tour_length; // update
46         remote shortest_known_tour_length
47     } else {
48         // Someone else already found a better solution. They will
49         broadcast theirs to us soon.
50         desire_break = true; // cancel sending our inferior solution
51         to other cores
52         goto cleanup_loop;
53     }
54 cleanup_loop:
55     release_qpe_semaphore((semaphore_t)qpe_pe, qpe_x, qpe_y);
56     if (desire_break)
57         break;
58 }
59 if (desire_break) {
60     // log that we found a better tour on another core
61     printf("B%d", (int)iteration);
62 } else {
63     // log that we successfully broadcasted our tour
64     printf("S%d", (int)iteration);
65 }
66 }
67
68 if (shortest_known_tour_length < check_shortest_known_tour_length) {
69     check_shortest_known_tour_length = shortest_known_tour_length;
70     last_improvement_iteration = iteration;
71 }
72
73 iteration++;
74
75 // global trail updating of pheromones
76 perform_global_trail_updating((city_id_t *)shortest_known_tour,
77 shortest_known_tour_length);
78
79 release_qpe_semaphore((semaphore_t)my_qpe_pe, my_qpe_x, my_qpe_y);
80
81 // detect algorithm convergence
82 if (iteration >= MIN_ITERATIONS && iteration -
83 last_improvement_iteration > MAX_ITERATIONS_WITHOUT_IMPROVEMENT) {
84     running = false;
85 }
86 }
87 }

```

```

82     return (int)0xcafebabe; // application finished successfully
83 }

```

Listing 12: Broadcasting an improved tour to every other PE on a single chip. Changes are relative to [Listing 11](#).

```

1  #include "common.h"
2
3  #include "aco.h"
4
5  volatile bool4 running = false;
6  volatile uint32_t last_improvement_iteration = 0;
7  volatile uint32_t iteration = 0;
8
9  volatile float shortest_known_tour_lengths[256] __attribute__((aligned(16))) =
    {0};
10
11 volatile bool4 is_supervisor_core = false;
12 volatile coreid_t supervisor_core_id;
13 volatile uint8_t supervisor_qpe_x, supervisor_qpe_y, supervisor_qpe_pe;
14
15 void supervisor_update_all_worker_cores(bool force);
16
17 // FT_INT_1 is triggered by the host, when all cities and parameters have been
    uploaded
18 void FT_INT_1_Handler(void) {
19     rf_disable_feedthrough_interrupt_lines(FT_INT_1_Line); // disable handler
20     running = true;
21 }
22
23 // FT_INT_3 is triggered by the host, when the algorithm has converged or
    should otherwise exit the program
24 void FT_INT_3_Handler(void) {
25     rf_disable_feedthrough_interrupt_lines(FT_INT_3_Line); // disable handler
26     running = false;
27 }
28
29 // triggered by the supervisor on the worker cores, indicating they can begin
    computation
30 bool Regfile_INT_0_Handler_Callback(Regfile_INT_Line self ALLOWUNUSED) {
31     running = true;
32     return true;
33 }
34
35 // triggered by a worker on the supervisor cores, indicating they just uploaded
    an improved tour to the supervisors SRAM
36 bool Regfile_INT_1_Handler_Callback(Regfile_INT_Line self ALLOWUNUSED) {
37     supervisor_update_all_worker_cores(false);

```

```

38     // semaphore was acquired by the core that triggered the interrupt
39     release_qpe_semaphore((semaphore_t)supervisor_qpe_pe, supervisor_qpe_x,
supervisor_qpe_y);
40     return false;
41 }
42
43 // triggered by the host on the supervisor cores, indicating a better tour was
found on another chip and just uploaded to the supervisors SRAM by the host
44 bool Regfile_INT_2_Handler_Callback(Regfile_INT_Line self ALLOWUNUSED) {
45     supervisor_update_all_worker_cores(true);
46     // semaphore was acquired by the host
47     release_qpe_semaphore((semaphore_t)supervisor_qpe_pe, supervisor_qpe_x,
supervisor_qpe_y);
48     return false;
49 }
50
51 void supervisor_update_all_worker_cores(bool force) {
52     coreid_t found_improvement_on_core = NO_CORE_FOUND;
53     // determine which worker has found an improvement
54     FOR_EACH_ENABLED_CORE(core_id) {
55         if (core_id == supervisor_core_id)
56             continue; // skip supervisor core
57         if (shortest_known_tour_lengths[core_id] < shortest_known_tour_length)
58         {
59             shortest_known_tour_length = shortest_known_tour_lengths[core_id];
60             found_improvement_on_core = core_id;
61         }
62     }
63     // broadcast improvement to all worker cores, until we are certain that all
cores have the best known tour
64     while (found_improvement_on_core != NO_CORE_FOUND || force) {
65         force = false;
66         if (found_improvement_on_core != NO_CORE_FOUND) {
67             // read shortest known tour from the core that has it
68             uint8_t qpe_x, qpe_y, qpe_pe;
69             UNPACK_COREID_INT0(found_improvement_on_core, qpe_x, qpe_y,
qpe_pe);
70             acquire_qpe_semaphore((semaphore_t)qpe_pe, qpe_x, qpe_y,
UINT32_MAX);
71             dma_read_variable_from_other_core(found_improvement_on_core, (void
*)&shortest_known_tour, pad_to_divisible_by_4(NUM_CITIES));
72             dma_read_variable_from_other_core(found_improvement_on_core, (void
*)&shortest_known_tour_length, sizeof(shortest_known_tour_length));
73             release_qpe_semaphore((semaphore_t)qpe_pe, qpe_x, qpe_y);
74             found_improvement_on_core = NO_CORE_FOUND;
75         }
76     }

```

```

76     // broadcast to all worker cores
77     FOR_EACH_ENABLED_CORE(core_id) {
78         if (core_id == supervisor_core_id)
79             continue; // skip supervisor core
80         if (shortest_known_tour_lengths[core_id] <=
            shortest_known_tour_length) {
81             continue; // core already has the best tour
82         }
83         uint8_t qpe_x, qpe_y, qpe_pe;
84         UNPACK_COREID_INT0(core_id, qpe_x, qpe_y, qpe_pe);
85         float *remote_shortest_known_tour_length = (float
            *)get_global_addr_pe_mem(qpe_x, qpe_y, qpe_pe,
            (uint32_t)&shortest_known_tour_length);
86         acquire_qpe_semaphore((semaphore_t)qpe_pe, qpe_x, qpe_y,
            UINT32_MAX);
87         if (*remote_shortest_known_tour_length <
            shortest_known_tour_length) {
88             // found a better core
89             found_improvement_on_core = core_id;
90             release_qpe_semaphore((semaphore_t)qpe_pe, qpe_x, qpe_y);
91             break; // restart broadcasting with new knowledge of which core
                has best tour
92         } else {
93             // send the tour to the core
94             dma_write_variable_to_other_core(core_id, (void
                *)&shortest_known_tour, pad_to_divisible_by_4(NUM_CITIES));
95             *remote_shortest_known_tour_length =
                shortest_known_tour_length;
96             release_qpe_semaphore((semaphore_t)qpe_pe, qpe_x, qpe_y);
97         }
98     }
99 }
100 }
101
102 int supervisor_main() {
103     // setup
104     for (int i = 0; i < 256; i++) {
105         shortest_known_tour_lengths[i] = INFINITY;
106     }
107
108     // tell all cores they can start computation now
109     FOR_EACH_ENABLED_CORE(core_id) {
110         if (core_id == supervisor_core_id)
111             continue; // skip supervisor core
112         uint8_t qpe_x, qpe_y, qpe_pe;
113         UNPACK_COREID_INT0(core_id, qpe_x, qpe_y, qpe_pe);

```

```

114     rf_trigger_regfile_interrupts(qpe_x, qpe_y, qpe_pe,
115     Regfile_INT_0_Line);
116 }
117 // main loop
118 while (running) {
119     __WFI(); // wait-for-interrupt in lower power state
120 }
121
122 return (int)0xcafebabe;
123 }
124
125 int worker_main() {
126     reset_edges(TAU0);
127
128     // main loop
129     while (running) {
130         // reset ants with random starting cities
131         for (uint32_t i = 0; i < NUM_ANTS && running; i++) {
132             ant_t *ant = &ants[i];
133             reset_ant(ant, random_city());
134         }
135
136         // process ants until all ants have a completed tour
137         for (uint32_t cities_visited = 0; cities_visited < NUM_CITIES - 1 &&
138             running; cities_visited++) {
139             for (uint32_t i = 0; i < NUM_ANTS && running; i++) {
140                 ant_t *ant = &ants[i];
141                 city_id_t r = current_city(ant);
142                 city_id_t s = choose_city_s(ant);
143                 travel_to_city(ant, s);
144                 perform_local_trail Updating(r, s);
145             }
146
147             // determine shortest tour
148             ant_t *best_ant = &ants[0];
149             for (uint32_t i = 0; i < NUM_ANTS && running; i++) {
150                 ant_t *ant = &ants[i];
151                 if (ant->tour_length >= best_ant->tour_length) {
152                     best_ant = ant;
153                 }
154             }
155
156             if (!running)
157                 break; // didn't complete the iteration. data is worthless

```

```

158
159     acquire_qpe_semaphore((semaphore_t)my_qpe_pe, my_qpe_x, my_qpe_y,
160                           UINT32_MAX);
161
162     // update shortest_known_tour
163     if (best_ant->tour_length < shortest_known_tour_length) {
164         memcpy((void *)shortest_known_tour, best_ant->tour,
165               sizeof(city_id_t) * NUM_CITIES);
166         shortest_known_tour_length = best_ant->tour_length;
167         last_improvement_iteration = iteration;
168     }
169
170     // global trail updating of pheromones
171     perform_global_trail_updating((city_id_t *)shortest_known_tour,
172                                   shortest_known_tour_length);
173
174     release_qpe_semaphore((semaphore_t)my_qpe_pe, my_qpe_x, my_qpe_y);
175
176     if (iteration == last_improvement_iteration) {
177         // tell supervisor core that we found an improvement
178         acquire_qpe_semaphore((semaphore_t)supervisor_qpe_pe,
179                               supervisor_qpe_x, supervisor_qpe_y, UINT32_MAX);
180         if (iteration > 0)
181             printf("I%u", iteration);
182         shortest_known_tour_lengths[my_core_id] =
183             shortest_known_tour_length;
184         dma_write_variable_to_other_core(supervisor_core_id, (void
185 *)&shortest_known_tour_lengths[my_core_id],
186                                           sizeof(shortest_known_tour_lengths[my_core_id]));
187         rf_trigger_regfile_interrupts(supervisor_qpe_x, supervisor_qpe_y,
188                                       supervisor_qpe_pe, Regfile_INT_1_Line); // the supervisor core will
189                                           release the semaphore
190     }
191
192     iteration++;
193
194     // if (iteration >= MIN_ITERATIONS && iteration -
195     // last_improvement_iteration > MAX_ITERATIONS_WITHOUT_IMPROVEMENT) {
196     //     running = false;
197     // }
198 }
199
200 return (int)0xcafebabe; // application finished successfully
201 }
202
203 int main() {
204     // subscribe to interrupts we want to use
205     rf_subscribe_feedthrough_interrupts(FT_INT_1_Line //

```

```

196         | FT_INT_3_Line, //
197         (1UL << __NVIC_PRI0_BITS) - 2UL, true);
198
199     rf_subscribe_regfile_interrupts(Regfile_INT_0_Line //
200         | Regfile_INT_1_Line //
201         | Regfile_INT_2_Line //
202         ,
203         (1UL << __NVIC_PRI0_BITS) - 2UL);
204
205     configure_qpe_semaphore_0(my_qpe_x, my_qpe_y, 1);
206     configure_qpe_semaphore_1(my_qpe_x, my_qpe_y, 1);
207     configure_qpe_semaphore_2(my_qpe_x, my_qpe_y, 1);
208     configure_qpe_semaphore_3(my_qpe_x, my_qpe_y, 1);
209
210     // we select the first enabled core as the "supervisor" core
211     supervisor_core_id = first_enabled_core;
212     UNPACK_COREID_INT0(supervisor_core_id, supervisor_qpe_x, supervisor_qpe_y,
213         supervisor_qpe_pe);
214     is_supervisor_core = (my_core_id == supervisor_core_id);
215     assert(first_enabled_core != last_enabled_core); // need at least two cores
216     // to have one worker core to perform computations
217
218     while (!running) {
219         __WFI(); // wait for FT_INT_1, indicating cities and parameters have
220         // been uploaded
221     }
222
223     assert_aco_parameters();
224
225     shortest_known_tour_length = INFINITY;
226     iteration = 0;
227     last_improvement_iteration = 0;
228
229     // run core dependent main
230     if (my_core_id == supervisor_core_id) {
231         // we are the supervisor core
232         return supervisor_main();
233     } else {
234         // we are a worker core
235         // wait for Regfile_INT_0, indicating supervisor core is ready and thus
236         // we can start computation
237         running = false;
238         while (!running) {
239             __WFI();
240         }
241         return worker_main();

```

```
238     }  
239 }
```

Listing 13: Multi-colony island model with a supervisor core per chip, scaling the implementation across a 48-chip board.