Analysis and Execution of a Data-Flow Graph Explicit Model Using Static Metaprogramming

Alexandre Bardakoff

▶ To cite this version:

Alexandre Bardakoff. Analysis and Execution of a Data-Flow Graph Explicit Model Using Static Metaprogramming. Distributed, Parallel, and Cluster Computing [cs.DC]. Université Clermont Auvergne, 2021. English. NNT : 2021UCFAC109 . tel-03813645

HAL Id: tel-03813645
https://theses.hal.science/tel-03813645

Submitted on 13 Oct 2022

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

Université Clermont Auvergne

École Doctorale des Sciences Pour l'Ingénieur

Analysis and Execution of a Data-Flow Graph Explicit Model Using Static Metaprogramming

Laboratoire d'Informatique, de Modélisation et d'Optimisation des Systèmes

(UMR 6158)

Alexandre Bardakoff

Directeurs de thèse: Bruno Bachelet, Loïc Yon
Encadrant: Walid Keyrouz

Président du jury: M. David Hill
Rapporteurs: M. Joël Falcou, M. Mamadou Kaba Traoré
Examinateurs: Mme Jian Ji Li, M. Shuvra S. Bhattacharyya
Invité: M. Timothy Blattner

2018 - 2021
Date de soutenance: 17 Décembre 2021

À mon grand père.

Acknowledgements

J'aimerais commencer par remercier mes directeurs de thèse : Bruno Bachelet, Walid Keyrouz et Loïc Yon pour avoir construit un cadre de travail qui a permis le succès de ce projet, sans eux, rien n'aurait été possible. Je tiens aussi à les remercier d'avoir cru en ce projet malgré les difficultés inhérentes au travail à distance entre deux continents, pour leurs conseils, pour leur support et pour les heures de relectures.

J'adresse mes remerciements à Joël Falcou et Mamadou Kaba Traoré d'avoir rapporté ce travail. Je tiens aussi à remercier Jian Ji Li et Shuvra S. Bhattacharyya pour avoir relu mon travail et pour leurs commentaires. I would like to thank Shuvra S. Bhattacharyya for reading this work and for his comments. Je n'en serai pas là sans David Hill qui m'a fait l'honneur d'accepter d'être mon président de jury.

I would like to thank specially Timothy Blattner for his assistance, insightful comments, and suggestions at every stage of the project. Je tiens à remercier Claude Mazel pour s'être rendu disponible et pour son aide. It's thanks to the support and help from all the ITL family that have made my studies and life in the USA a wonderful time. Merci aussi à François Delobel et Guenal Davalan de m'avoir enseigné ce qu'est l'informatique et l'importance de nos métiers.

Pour leur amour, pour leur soutien indéfectible et pour toujours avoir été présent pour moi, je remercie ma maman, mon frère et mes grands-parents; merci pour tout. J'ai une grande pensée pour toute ma famille; bien que je souhaite écrire pour chacun d'eux toute l'importance qu'ils ont pour moi, je ne me lancerai pas dans cet exercice par souci de concision, mais surtout, pour ne pas manquer sa réalisation. Je tiens à remercier Minli pour des raisons qui rempliraient un manuscrit bien plus long que celui-ci 我喜欢(爱?)你。

Contents

Introduction 14

I State of the art 17

1 Parallel runtime systems 18
1.1 Parallel programming . . . 18
1.1.1 Some definitions . . . 18
1.1.2 Complexity . . . 19
1.2 Flynn's taxonomy . . . 20
1.3 Thread, a low-level primitive . . . 20
1.4 Futures and promises, mid-level constructs . . . 22
1.5 Domain-specific parallel libraries . . . 24
1.5.1 Examples of libraries . . . 25
1.5.2 Implicit execution model . . . 26
1.5.3 Specificity of the libraries . . . 26
1.5.4 Matrix multiplication with domain-specific libraries . . . 27
1.6 Algorithmic skeletons . . . 27
1.6.1 C++ execution policies . . . 27
1.6.2 Intel Threading Building Blocks . . . 29
1.6.3 Composability . . . 29
1.6.4 Matrix multiplication with algorithmic skeletons . . . 30
1.7 Task-based approach . . . 30
1.7.1 Message-driven libraries . . . 30
1.7.2 Graph-based libraries . . . 31
1.8 Multi-paradigm models . . . 36
1.8.1 OpenMP . . . 36
1.8.2 OmpSs-2 . . . 36
1.8.3 Kokkos . . . 36
1.8.4 Parallelism considerations . . . 37
1.8.5 Matrix multiplication with OpenMP . . . 37
1.9 Parallel languages . . . 38
1.9.1 Chapel . . . 38
1.9.2 Rust . . . 38
1.9.3 Go . . . 38
1.9.4 Matrix multiplication with a parallel language . . . 39
1.10 Conclusion . . . 40

2 Metaprogramming techniques in C++ 41
2.1 Metaprogramming taxonomies . . . 41
2.1.1 Pasalic's taxonomy . . . 42
2.1.2 Sheard's taxonomy . . . 43
2.1.3 Damaševičius and Štuikys' taxonomy . . . 43
2.1.4 Lilis and Savidis' taxonomy . . . 44
2.1.5 Conclusion . . . 46
2.2 Trivial metaprogramming in C++ . . . 47
2.2.1 Metaprogramming with strings . . . 47
2.2.2 Metaprogramming with macros . . . 48
2.3 Template metaprogramming . . . 49
2.3.1 Fundamental template principles . . . 49
2.3.2 Variadic templates . . . 52
2.3.3 Type deduction . . . 54
2.3.4 C++ templates vs. Java generics . . . 55
2.3.5 Metafunction . . . 58
2.3.6 SFINAE . . . 60
2.3.7 Traits and helpers . . . 62
2.3.8 Constraints and concepts . . . 63
2.3.9 Computation with template metaprogramming . . . 67
2.4 Constant expressions . . . 71
2.4.1 Constant expressions in Lilis and Savidis' taxonomy . . . 75
2.4.2 `constexpr` vs. `consteval` . . . 75
2.4.3 Template metaprogramming and constant expressions . . . 75
2.4.4 Usage of metaprogramming . . . 79
2.5 Conclusion . . . 80

II Design 81

3 Hedgehog framework 82
3.1 HTGS . . . 82
3.1.1 Model and types of nodes . . . 82
3.1.2 Memory manager . . . 84
3.1.3 Graphical feedback . . . 84
3.2 HTGS limitations . . . 84
3.2.1 `IData` interface . . . 85
3.2.2 Graph manipulation . . . 85
3.2.3 Graph construction . . . 85
3.2.4 Bookkeeper broadcast . . . 86
3.2.5 Internals . . . 86
3.2.6 Memory manager . . . 86
3.2.7 Conclusion . . . 86
3.3 Early decisions . . . 86
3.3.1 Multiple inputs and output broadcast . . . 87
3.3.2 Major API terminology changes . . . 89
3.3.3 Key decisions from HTGS . . . 90
3.3.4 Conclusion . . . 90
3.4 Hedgehog model . . . 91

3.4.1 Hedgehog data-flow graph . . . 91
3.4.2 Data pipelining . . . 92
3.4.3 Output streaming . . . 93
3.4.4 Scheduler-free execution . . . 93
3.4.5 Conclusion . . . 93
3.5 Separations of concerns . . . 94
3.5.1 Computational task . . . 94
3.5.2 Management node . . . 98
3.5.3 Graph . . . 100
3.5.4 Execution pipeline . . . 102
3.6 Software architecture . . . 103
3.6.1 API . . . 103
3.6.2 Core . . . 112
3.6.3 Library extensibility . . . 114
3.7 Conclusion . . . 116

4 Experimentation for performance 117
4.1 Profiling and feedback . . . 117
4.1.1 Graphical representation . . . 119
4.1.2 Signal handler . . . 120
4.1.3 NVTX . . . 122
4.2 Intrinsic performance . . . 124
4.2.1 Data transfer latency . . . 124
4.2.2 Profiling cost . . . 125
4.3 Real-life experiments . . . 126
4.3.1 Matrix multiplication . . . 127
4.3.2 LU decomposition with partial pivoting . . . 139
4.4 Libraries based on Hedgehog . . . 141
4.4.1 HMBLib . . . 142
4.4.2 FastLoader . . . 143
4.5 Conclusion . . . 144

5 Hedgehog at compile-time 146
5.1 Conformity checking at graph construction . . . 146
5.1.1 Basic checking . . . 147
5.1.2 Input node connection . . . 147
5.1.3 Output node connection . . . 153
5.1.4 Edge connection . . . 155
5.1.5 Conclusion . . . 157
5.2 Static analysis of data-flow graph . . . 157
5.2.1 Limitations of `constexpr` . . . 159
5.2.2 Compile-time representation . . . 159
5.2.3 Compile-time tests . . . 162
5.2.4 From compile-time to runtime . . . 165
5.3 Experiments and results . . . 166
5.3.1 Compilation performance . . . 167
5.3.2 Overhead of conformity checking . . . 169
5.3.3 Overhead of graph analysis . . . 170
5.4 Conclusion . . . 175

Conclusion **176**

Bibliography **178**

Appendices 187

A Pi approximation **187**

B Hedgehog latency measurement **190**

C HTGS latency measurement **192**

D Accuracy matrix multiplication **196**

E Execution of CPU-only matrix multiplication **199**

List of Figures

1.1 Simple matrix multiplication decomposition in blocks 35
1.2 Matrix multiplication with a graph approach 35

2.1 Damaševičius and Štuikys' taxonomy 44
2.2 Relations between structural and processing concepts 45
2.3 Lilis and Savidis' taxonomy 45

3.1 Task and bookkeeper in HTGS 83
3.2 Graph representation of the matrix multiplication algorithm in HTGS 87
3.3 Hedgehog data-flow graph model 91
3.4 Data pipelining 92
3.5 Hedgehog Multi-threaded task 95
3.6 Hedgehog task activity diagram 96
3.7 Relation between tasks and a state manager 100
3.8 UML sequence diagram of interactions between a state manager and a state 101
3.9 UML class diagram of the graph modeling [0] 104
3.10 UML class diagram of the task modeling [0] 106
3.11 UML class diagram of specialized tasks modeling [0] 108
3.12 UML class diagram of the execution pipeline modeling [0] 109
3.13 Simplified UML class diagram of the communication layer [0] 113
3.14 Simplified UML class diagram of the relation between the `AbstractTask` and `CoreTask` 115

4.1 DOT representation of the graph achieving a LU decomposition algorithm 118
4.2 Graphical representation showing an execution pipeline 119
4.3 Graphical output produced by the signal handler 121
4.4 Full NVTX profiling of a heterogeneous matrix multiplication implementation 123
4.5 Zoom into NVTX profiling of a heterogeneous matrix multiplication implementation 123
4.6 Hedgehog and HTGS latency for different task's number of threads . 125
4.7 Hadamard product using Hedgehog, with and without profiling . . . 126
4.8 Data-flow graph of the heterogeneous matrix multiplication implementation 134
4.9 Data-flow graph of the multi-GPU matrix multiplication implementation 137
4.10 Data-flow graph of the advanced multi-GPU matrix multiplication implementation 137
4.11 Advanced multi-GPU matrix multiplication of $64k \times 64k$ matrices decomposed in $8k \times 8k$ blocks 139

4.12 LU decomposition algorithm 139
4.13 Data-flow graph of LU decomposition with partial pivoting 140
4.14 CPU implementation performance of LU decomposition with partial pivoting 141
4.15 Block release time of CPU implementation of LU decomposition . . . 141
4.16 Performance of HMBLib matrix multiplication implementation 142
4.17 Block release time of CPU implementation of matrix multiplication with HMBLib 143
4.18 FastLoader data-flow graph 144

5.1 Input conformity check between a node and a graph 148
5.2 CLion IDE hint of Hedgehog v.1 static checking (failure of `enable_if`) 150
5.3 CLion IDE hint of Hedgehog v.2 static checking (failure of static assertion) 152
5.4 CLion IDE hint of Hedgehog v.2 static checking (full concept) 153
5.5 Output conformity check between a node and a graph 154
5.6 Edge conformity check between two nodes 155
5.7 CLion IDE hint of Hedgehog static checking of edge connection . . . 156
5.8 Simplified UML class diagram of HedgehogCX 158
5.9 Graph example for compile-time analysis 161
5.10 Compilation times of HTGS and both Hedgehog versions, for matrix multiplication and LU decomposition 167
5.11 Phase compilation times of HTGS and both Hedgehog versions, for matrix multiplication and LU decomposition 168
5.12 Step compilation times of HTGS and both Hedgehog versions, for matrix multiplication and LU decomposition 169
5.13 Impact of static conformity checking in both Hedgehog versions, for matrix multiplication and LU decomposition 169
5.14 Impact of the static analysis library on the compilation duration, for matrix multiplication and LU decomposition 170
5.15 Phase compilation times of the static analysis, for matrix multiplication and LU decomposition 170
5.16 Step compilation times of the static analysis, for matrix multiplication and LU decomposition 171
5.17 Path graph structure 171
5.18 Relative number of constant expression operations, depending on the path graph size 172
5.19 Relative number of constant expression operations, depending on the random graph size 172
5.20 Phase compilation times for analyzing path graphs of different sizes . 173
5.21 Phase compilation times for analyzing random graphs of different sizes 173
5.22 Step compilation times for analyzing path graphs of different sizes . . 174
5.23 Step compilation times for analyzing random graphs of different sizes 174

E.1 Performance of the CPU-only matrix multiplication algorithm implemented with Hedgehog 200

List of Tables

1.1 Situation of data race with different outcomes 19
1.2 Flynn's taxonomy 20
1.3 Execution time (ms) of nested execution policies over 20 iterations . . 28

2.1 Concepts related to metaprogramming from Damaševičius and Štuikys' analysis 42
2.2 Fold expression evaluation 53
2.3 Evaluation of the power function implementations for Clang 12.0.1 and GCC 11.1 (cf. Code 2.31) 69
2.4 Evaluation of the sine function implementations for Clang 12.0.1 and GCC 11.1 (cf. Code 2.32) 70

3.1 Relation between the nodes and their cores 112
3.2 Inheritance table of the cores and their interfaces 114

4.1 Properties of the nodes used in the advanced multi-GPU matrix multiplication implementation 138

Listings

1.1 Matrix multiplication with `std::thread` 21
1.2 Excerpt of synchronization code with futures for π estimation 23
1.3 Worker code for π estimation 23
1.4 Standard deviation computation for π estimation 23
1.5 Composition of algorithm with execution policies 27
1.6 Matrix multiplication with C++ algorithms library and execution policies 29
1.7 Matrix multiplication with TBB 30
1.8 Matrix multiplication with OpenMP 37
1.9 Matrix multiplication with Chapel 39

2.1 Metaprogram based on strings 47
2.2 Output of the previous code with "blabliblou" given as argument . . 47
2.3 Metaprogram based on macro 48
2.4 Code with the macro expanded 48
2.5 Metaprogram based on macro with side effects 49
2.6 Template definition and instantiation 50
2.7 Class and variable templates 51
2.8 Partial and full template specializations 51
2.9 NTTP usage 52
2.10 Variadic template and pack expansion 52
2.11 Output example after template instantiation 53
2.12 Recursive manipulation of pack elements 54
2.13 Difference of deduction between `auto` and `decltype` 54
2.14 Boxing in Java 56
2.15 Generics in Java 57
2.16 Code produced by the instantiation process of Java 57
2.17 Example of addition function and metafunction 58
2.18 Example of value metafunction (`integral_constant`) 59
2.19 Example of type metafunction composition 59
2.20 Usage of SFINAE to test if a member function exists 60
2.21 `enable_if` metafunction 61
2.22 Helper type and variable templates 63
2.23 Conjunction and disjunction of constraints 63
2.24 Concept definitions with a atomic constraint. 64
2.25 Different usages of concepts 65
2.26 Types of requirements 65
2.27 Concept example 66
2.28 SFINAE alternative to concept 67
2.29 Factorial computation with template metaprogramming 67
2.30 Assembly code of compile-time factorial 68

2.31 Different power function implementations 68
2.32 Different sine function implementations 69
2.33 If construct with template metaprogramming 70
2.34 OR statement with template metaprogramming 71
2.35 Factorial with `constexpr` specifier (using C++ 11 norm) 72
2.36 `"constexpr" new` support, direct use 73
2.37 `"constexpr" new` support, indirect use 73
2.38 `"constexpr" new` support, deallocation missing 74
2.39 `"constexpr" new` support, undefined behavior 74
2.40 `"constexpr" new` support, return of allocated memory 74
2.41 Removing duplicates from tuples, using constant expressions 75
2.42 Removing duplicates from tuples, using template metaprogramming . 76
2.43 Output from Code 2.27, with a faulty case 77
2.44 Output from Code 2.28, with a faulty case 77

3.1 `LoadMatrixTask` basic structure 88
3.2 `LoadMatrixTask` rewritten 88
3.3 `LoadMatrixTask executeTask` rewritten 88
3.4 Broadcast writing proposal 89
3.5 Block management of matrix A for the matrix multiplication algorithm 99
3.6 Smart pointer checking on Hedgehog API v.1 (C++ 17) 111
3.7 Smart pointer usage on Hedgehog API v.2 (C++ 20) 111

4.1 Usage of the signal handler for a graph with deadlock 122
4.2 Product task, from CPU matrix multiplication 128
4.3 Addition task, from CPU matrix multiplication 128
4.4 Input state, from CPU matrix multiplication 129
4.5 Row traversal task, from CPU matrix multiplication 130
4.6 Partial computation state, from CPU matrix multiplication 130
4.7 Partial computation state manager, from CPU matrix multiplication 131
4.8 Product task on GPU, from heterogeneous matrix multiplication . . . 132
4.9 Copy in GPU task, from heterogeneous matrix multiplication 133
4.10 Copy out GPU task, from heterogeneous matrix multiplication 133
4.11 Data-flow graph, from multi-GPU matrix multiplication 135
4.12 Usage of execution pipeline, from multi-GPU matrix multiplication . 135
4.13 Definition of an execution pipeline, from multi-GPUs matrix multiplication 136

5.1 Static checking at memory manager connection 147
5.2 Static checking of graph `input` method, using template metaprogramming 148
5.3 Helper to get `MultiReceivers` type from input type list 148
5.4 Metafunction checking if at least one of the set of types `T1` is included in set `T2` 149
5.5 Metafunction checking if type `T` is part of set `Ts` 150
5.6 Definition of concepts for Hedgehog nodes. 151
5.7 Static checking of graph `input` method, using concepts and static assertion 152
5.8 Static checking of graph `input` method, using concepts only 153

5.9 Static checking of graph `output` method, using template metaprogramming . . . 153
5.10 Static checking of graph `output` method, using concepts and static assertion . . . 154
5.11 Static checking of graph `addEdge` method, using template metaprogramming . . . 155
5.12 Static checking of graph `addEdge` method, using concepts and static assertion . . . 156
5.13 Definition of static node and graph . . . 160
5.14 Graph construction at compile-time . . . 161
5.15 Tests addition for compile-time analysis . . . 162
5.16 Partial code of the critical path test . . . 163
5.17 Tests execution and "defrosting" . . . 164
5.18 Conversion from static to runtime graph . . . 165

A.1 Monte Carlo simulation to estimate π with `std::async` . . . 187

B.1 C++ code to measure Hedgehog latency . . . 190

C.1 C++ code to measure HTGS latency . . . 192

Introduction

Nowadays, the hardware landscape has changed. It is not necessary to have a multi-node server to get tremendous computing power available. Massively parallel single nodes with multiple CPUs, accelerators, and GPUs become more and more available to a new public: small companies, non-expert academic teams, etc. Parallel computing comes with a set of challenges such as orchestrating concurrent computations, managing multiple memory resources, race conditions, deadlocks, and data motion costs. Therefore, all users, from library developers to end-users, need to face them. There are a lot of available solutions, their forms and levels of accessibility vary: from one-call parallel library that will hide completely the computation to exascale approach that offers a level of complexity difficult to apprehend.

At NIST, in the ITL group, a first effort called HTGS [Blattner et al., 2017] for "Hybrid Task Graph Scheduler", driven by Timothy Blattner and Walid Keyrouz, proposes a framework to tackle efficient parallel computation on a heavy and heterogeneous node. In this manuscript, we present the architecture of Hedgehog, the next version of HTGS, that targets more extensibility, more expressiveness, and more safety during the design of parallel code.

HTGS and Hedgehog are based on a simple and understandable execution model using a data-flow graph representation for expressing coarse grain parallel executions. It uses data-pipelining to fully exploit the available hardware by firing a task as soon as a piece of data is available. The threading model consists only of threads statically bound to persistent tasks during the whole runtime.

The developer is at the center of our approach, in order to provide him/her the keys to control the performance of the parallel execution. The first principle is to keep the data-flow graph model that the developer uses to express an algorithm, from the first steps of the conception to its effective execution. Then, costless visual feedback mechanisms are provided by the library to expose how the computation is conducted on a specific hardware. We profile at node level the computations and present in a graphical representation graphs with the different metrics gathered. The representation helps the developer, who at a glance can assess the computation and determine the bottleneck. With this help and the fact that the model remains unchanged, the end-user can iteratively improve the end-to-end performance, that is what we call *experimentation for performance*.

Building the data-flow graph is secured through compile-time checking that uses common template metaprogramming techniques of the C++ language combined with *concepts*, a recent addition to the language. It allows detecting incompatible or potentially unsafe elements in the graph structure at an early step of the design, thus providing some guaranty for runtime; and presenting clear error messages to the developer at compile-time, or even during development if the IDE (Integrated Development Environment) used is powerful enough.

The data-flow graph remaining unchanged from design to execution, it is fixed

at compile-time. We propose to use a recent addition to the C++ language, the *generalized constant expressions*, to write an additional library to Hedgehog that acts mainly at compile-time to represent the data-flow graph and to design complex algorithms to analyze its structure.

With this library, it is possible to test a graph structure at compile-time and thus to provide extended guarantees at runtime. This approach is totally different from the usual template metaprogramming techniques, as there is no fundamental difference between code written for compile-time analysis before execution, or for runtime directly. The library comes with out-of-the-box compile-time tests, but is extensible enough for an end-user to add other tests to broaden the variety of checking done at compile-time.

Chapter 1 presents existing parallel run-time systems. We discriminate among 6 different types of models for different kinds of parallelism, from the lowest level available like the threads to the highest level approaches like the task-based approach for clusters. These approaches have their own advantages and disadvantages that we analyze through code examples and with a user perspective. From this analysis, we propose the features that are important to us for a "good" parallel model.

Chapter 2 showcases metaprogramming techniques in C++ that are a cornerstone of our model implementation. In this chapter, we propose an overview of the available techniques in C++, with a strong focus on template metaprogramming and on the *generalized constant expressions* available in the latest C++ 20 standard. The focus on template metaprogramming concerns both the classic approach with metafunctions but also the newly introduced *concepts and constraints*.

Chapter 3 reveals the Hedgehog framework, both the execution model and the software architecture of the runtime system. Hedgehog results in a library for helping developers doing parallel computation on heterogeneous computing nodes. It is the successor of the HTGS library that we introduce in this chapter. We then develop our model based on the properties highlighted in Chapter 1. Hedgehog exploits a data-flow graph with data pipelining to get performance on heterogeneous nodes. It is different from traditional data-flow approaches as it does not have any scheduler to manage the computation. Moreover, we describe the library's architecture and the different ways the library can be extended by developers.

Chapter 4 elaborates on our *experimentation for performance* approach. First, we present our costless visual feedback and our integration of the NVTX library from NVIDIA to help assessing the computation; and then, we discuss different experiments and results. Notably, as Hedgehog is a library that succeeds to HTGS, it was important for us to have a system that is at least as-good-as the previous one, so we first compare their performance. Then, we propose to delimit the cost of the library and the performance achievable on real-life problems. To end this chapter, we present two libraries built with Hedgehog: an image processing and a linear algebra libraries.

Chapter 5 introduces the compile-time aspects of the Hedgehog library. Based on what is presented in Chapter 2, we propose two main systems to secure our Hedgehog model (presented in Chapter 3). The first consists of constraints on templates included in Hedgehog library to secure its API, notably during local graph interaction (like connecting nodes). We experiment with this first system in two versions, both using template metaprogramming, the first in C++ 17 with traditional techniques and the second in C++ 20 with concepts. The second system is a side

library called HedgehogCX that only exists for the second version of Hedgehog in C++ 20. This side library presents a limited Hedgehog graph representation at compile-time, allowing to test globally its structure during compilation, and to transform it afterwards into a completely operational Hedgehog graph. We close this chapter with an analysis of the compilation performance of these systems with the GCC and Clang compilers.

Part I

State of the art

Chapter 1

Parallel runtime systems

Achieving parallel computing is difficult: on top of the classical software engineering, the software design needs to use efficient parallel constructs to take advantage of the available hardware. It becomes even harder with high-end massively-parallel computing platforms: 64-core CPUs are currently available, with 80-core and 128-core CPUs promised to arrive soon; and accelerators such as GPUs extend this hardware parallelism both within a single accelerator and between multiple accelerators in the same compute node.

In order to explore the different software solutions, we first discuss a classification of computer architectures proposed by Michael J. Flynn in Section 1.2. Then we showcase different approaches available to the user to tackle parallelism. In Sections 1.3 and 1.4, we show two elementary mechanisms: the threads and the futures, respectively. These are parts of the fundamental tools to create parallel programs. In Section 1.5, we show domain-specific parallel libraries. They help users with no background in computer science to achieve efficient one-call functions on domain-specific problems. In Section 1.6, we introduce the algorithmic skeletons. They are a tool to construct complex algorithms with basic building blocks. In Section 1.7, we present two task-based approaches. They consider the algorithm as an interconnected set of tasks. In Section 1.8, we overview libraries offering multiple kinds of models. Notably, it presents OpenMP used mainly to parallelize loops and Kokkos proposing a task-based and an algorithmic skeleton approaches. In Section 1.9, we elaborate on parallel languages. This kind of languages considers concurrency as fundamental building blocks. Then in Section 1.10, we identify the challenges that we propose to tackle.

To compare the different strategies, we will illustrate, with pseudo codes or graphical representations, how to implement a matrix multiplication algorithm: $C' = C + \alpha * A * B + \beta$, where A, B, and C, are matrices, and α and β are scalars.

1.1 Parallel programming

1.1.1 Some definitions

An **accelerator** is a piece of hardware with a different architecture than the main processor, located on the same die or elsewhere; it is designed to answer a specific set of applications [Patel and Wen-mei, 2008]. We call a **node** a process entity separated from the other nodes (*e.g.*, a computer in a local network). A heterogeneous node is a processing unit containing at least a CPU and an accelerator. In the experiments

presented in this document, the accelerators are GPUs.

In the following sections, we use **concurrency** (sections of code that can run independently, *i.e.* they run alternatively in the same process unit) and **parallelism** (sections of code that run at the same time) [Silberschatz et al., 1991] without any distinction except when explicitly stated.

1.1.2 Complexity

When dealing with parallel computing, the first problem encountered is the expression of the parallelism, from low-level thread constructs (Section 1.3) to high-level domain-specific parallel libraries (Section 1.5), through mid-level constructs (Section 1.4).

After expressing the parallelism, problems specific to this type of computation arise. Usually, to safely use a resource in a concurrent way, the actor locks a resource access for its own usage, uses it and releases it, making it available to other actors. However, this process is not enough to guarantee the proper execution of the program. A deadlock can occur if multiple actors are holding resources and waiting for resources held by other actors.

If the locking mechanism is not implemented properly, two actors may have access at the same time to the same resource. We can overview two cases: if both actors read data of the resource, or if at least one of them writes data in the resource. In the first case, there is no problem, the data of the resource stay valid for both actors. In the second case, the order of operations, i.e. if the write operation occurs before or after the read of the other actor, changes the outcome of the process. In the example shown in Table 1.1, two actors read the same variable `val` and each one stores it in a (local) variable x, sets x to a different value (1 or 2, respectively), and writes x back in variable `val`. We observe that the final value of the variable depends on the order of execution of both actors, which is called a data race.

Time	**Actor 1**	**Memory**	**Actor 2**
0	x ← val	val = 0	
1	x ← 1		x ← val
2	val ← x	val = 1	x ← 2
3		val = 2	val ← x

Time	**Actor 1**	**Memory**	**Actor 2**
0		val = 0	x ← val
1	x ← val		x ← 2
2	x ← 1	val = 2	val ← x
3	val ← x	val = 1	

Table 1.1: Situation of data race with different outcomes

These problems are functional: they can change the proper execution of the algorithm or impact the performance of the computation. Moreover, when dealing with heterogeneous computing or cluster computing, we need to take into account the cost of moving data. In cluster computing, compute nodes are often connected through a network interface, and memory has to be sent in order to do the computation. The network interface in the DGX A100 of Nvidia has a $200\,\mathrm{Gbit\,s^{-1}}$ ($25\,\mathrm{GB\,s^{-1}}$) Ethernet connection. In heterogeneous computing, the GPUs have their own memory that we need to fill. A 4th generation PCI express has a bandwidth of $32\,\mathrm{GB\,s^{-1}}$ with 16 lines. In both cases, these costs are not trivial and need to be overlapped with computation to reduce the overall execution time.

1.2 Flynn's taxonomy

Back in 1972, Flynn presented a taxonomy to organize computer architectures [Flynn, 1972]. The way the computer is organized implies the way data move, how the processing units do their works, and what kind of parallelism is available. He categorizes the computer architectures following two axes, one is the number of instruction streams and second is the number of data streams as shown in Table 1.2.

	Single Data	**Multiple Data**
Single Instruction	SISD	SIMD
Multiple Instructions	MISD	MIMD

Table 1.2: Flynn's taxonomy

The number of instruction streams translates into the idea that each processing unit executes the same (single instruction) or different instructions (multiple instructions) during the same cycle. The number of data streams translates into the idea that each processing unit can operate on the same (single data) or on different data pieces (multiple data) per clock cycle. This taxonomy is still used to classify architectures, and libraries also use it to target such architectures and to exploit the underlying parallelism.

However, when building a library it is better to not target a specific architecture. This reflection results from our goal of creating a library that can tackle heterogeneous nodes with simple models to express coarse-grained parallelism structures. A user can use specialized libraries to exploit the underlying parallelism and tackle specific architectures. Therefore, we think possible to target different architectures with Intel CPUs and Nvidia GPUs seamlessly.

1.3 Thread, a low-level primitive

A thread is a "process within another process that uses the resources of the latter process" (ISO/IEC 2382:2015). It is a basic unit of execution, each thread possesses a program counter, a stack, some registers, and a thread identifier. Threads in the same program have access to the heap memory (and the access to this shared memory sometimes needs to be controlled).

The threads by themselves do not provide any guarantee on common parallel programming problems such as race condition or interlocking for example. Shared resource access and synchronization need to be managed by the developer; and other synchronization primitives like mutexes, semaphores, or condition variables, need to be used to guarantee the proper execution of the program.

A mutex (for mutual exclusion), is an object available in many languages that can be locked to protect a critical section of code (code that must only be executed by one thread at a time) from being accessed by multiple threads at the same time. Condition variables allow many threads to check a global condition (i.e., condition on a global resource that needs concurrent access control). A semaphore is a synchronization primitive with a counter. Unlike a mutex, it can allow more than one concurrent access to the same resource. A **barrier** is a coordination construct (that can be implemented with a condition variable) that will stop a group of threads at this point, until all of them reach the barrier.

The C++ thread is a direct encapsulation of the system thread. When creating a thread (an instance of class `std::thread`), it will be bound to a section of code that it will execute separately from the main thread. Its termination is usually checked by calling its `join` method that waits for this section of code to return. C++ 11 introduces a standard interface for creating and managing threads depending on what is available on the system. Since C++ 20, the semaphore and the barrier are available in the standard library.

Source Code 1.1: Matrix multiplication with `std::thread`

```
// Constants used in the algorithm
size_t const matSize = 10, blkSize = 3,
  nbBlk = (matSize + blkSize - 1) / blkSize,
  nbThreads = nbBlk * nbBlk;

float const alpha = 2, beta = 3;

// Matrices
std::array<float, matSize * matSize> A{}, B{}, C{};

// Block matrix multiplication algorithm
void blkMatMul(size_t const & row, size_t const & col) {
  auto maxRow = std::min(matSize, (row + 1) * blkSize);
  auto maxCol = std::min(matSize, (col + 1) * blkSize);

  // For all elements in a C block
  for(size_t i = row * blkSize; i < maxRow; ++i)
    for(size_t j = col * blkSize; j < maxCol; ++j) {
      // Along the common dimension of A and B
      for(size_t k = 0; k < matSize; ++k){
        // Does the multiplication
        C.at(i * matSize + j) +=
            alpha * A.at(i * matSize + k) * B.at(k * matSize + j);
      }
      // Adds beta
      C.at(i * matSize + j) += beta;
    }
}

int main() {
  size_t index = 0; // Threads counter
  std::array<std::thread, nbThreads> threads; // Array of threads

  // Generates randomly the matrices
  // ...

  // Loops over the blocks of C
  for(size_t rowIndex = 0; rowIndex < nbBlk; ++rowIndex)
    for(size_t colIndex = 0; colIndex < nbBlk; ++colIndex)
      // Creates a thread for each  C block
      threads[index++] = std::thread(blkMatMul, rowIndex, colIndex);

  // Waits for the threads to terminate
  for(auto & myThread : threads) myThread.join();
}
```

Code 1.1 presents a parallel implementation of the matrix multiplication using only the `std::thread` API. Each thread computes its own block of the C matrix. In this configuration, the matrices A and B are accessed concurrently by all the threads in read-only mode. Because each thread computes its own blocks with no overlapping with another one, there is no inter-thread communication, risk of data race, or interlocking possibility.

At lines 38 and 39, we iterate over the different block indexes in row and column dimensions. At line 41, we instantiate the threads bound to the function `blkMatMul` with the block indexes as arguments. The thread created is then stored in an array. At line 44, we join all the threads, meaning that we wait for all the block matrix multiplication to be done on separate regions of the matrix.

In order to help managing the parallelism and the computation synchronization, we introduce two possible common design patterns: the fork-join model that is a fundamental pattern when using threads, and the thread pool pattern, that is more dynamic (as the threads are not assigned on predetermined tasks) in order to achieve load-balancing.

In the **fork-join** model, multiple child threads are created (forked) from a main thread at one point in the execution to compute in parallel concurrent sections, and are joined when all the concurrent sections are done. So, only the main thread remains at the end.

In the **thread pool**, many threads are usually created ahead of time in a pool. When a section of concurrent code, a task, needs to be executed, it is submitted to a scheduler that assigns the task to an available thread. If no thread is available, the task is stored in a waiting queue for the next available thread. Each time a thread finishes a task, it becomes available for a new task. This mechanism naturally allows load balancing (which is interesting when the length of the tasks is not homogeneous).

1.4 Futures and promises, mid-level constructs

The concept of [a]synchronicity is orthogonal to the concept of concurrent programming. Synchronous tasks are executed in sequence. Each call is blocking, the execution of the following task is not started until the previous one is done. Asynchronous computations can start other computations before the previous ones are not completed.

In C++, `std::async` is a function that executes any callable (i.e., function, functor or lambda) asynchronously. The value returned by the callable is presented under the form of a *future* (`std::future` object). This is a difference with threads manipulated directly: it is possible to get a "return" value. With the thread mechanism alone, it is only possible to get a "return" value by passing an address or a reference of a value as argument, or putting the value in a shared memory location with a synchronization mechanism (e.g., condition variable) to alert when the information is available.

With futures, it is possible to wait for a value to become available. By calling its `get` method hiding the synchronisation mechanism, the current thread waits for the value associated with the future to be ready (another thread will update the value and alert that it is available). Automatically, `std::async` creates and manages a future object for the value returned by the callable it handles, but it is also possible to create promises (`std::promise` objects) that can be passed as arguments to the callable and, at any time, the callable can update the value and make it available. From outside the callable, one can associate a future to a promise and wait for its value to be ready.

We do not present a code for matrix multiplication with asynchronous constructs, the changes from Code 1.1 using threads will be minimal because of the way we do the computation. Instead, we have chosen to present a simple Monte Carlo simulation to estimate the π value in Appendix A. The algorithm considers a circular

sector inscribed in a unit square that forms a quadrant. It generates points in this space and counts the number of points inside the sector. The ratio of the quadrant area over the unit square area is $\pi/4$, we can then estimate the value of π from estimations of both areas by counting points put randomly in the unit square. The inputs of the algorithm are the number of samples (i.e., points) and the number of concurrent workers that will work in parallel. The output is the estimated value of π. An excerpt of the code in Appendix A is presented in Code 1.2.

Source Code 1.2: Excerpt of synchronization code with futures for π estimation

```
double approximatePi() {
  durations_.clear();
  durations_.reserve(numberWorkers_);

  // Creates a vector of futures to gather the number of inside
  ↪ points
  std::vector<std::future<unsigned long long int>> futures;
  futures.reserve(numberWorkers_);

  // Runs the different workers asynchronously
  for(int t = 0; t < numberWorkers_; ++t)
    futures.emplace_back(std::async(std::launch::async,
    ↪ &ExperimentPI::piMonteCarlo, this));

  // Gathers the results of each worker when they are done
  unsigned long long int numberPointsInside = 0;
  for(std::future<unsigned long long int>& f : futures)
  ↪ numberPointsInside += f.get();
}
```

Each worker has its own random generator, generates its amount of points, and has its local count of "inside points" which will be returned into the main method as shown in Code 1.3. Then every "inside points" are accumulated to estimate the value of π.

Source Code 1.3: Worker code for π estimation

```
[[nodiscard]] unsigned long long int piMonteCarlo() {
  std::random_device rd;
  std::mt19937 gen(rd());
  std::uniform_real_distribution<double> dis(0.0, 1.0);
  double x, y;
  unsigned long long int points_inside = 0;

  for(unsigned long long int i = 0; i < samplesPerWorker_; ++i) {
    x = dis(gen); y = dis(gen);
    if (x*x + y*y <= 1.0) ++points_inside;
  }

  return points_inside;
}
```

In order to get some insight on how the computation performed, we also gather the duration of the computation for each asynchronous worker. In order to achieve that, we have added a vector of durations, and to protect it from concurrent manipulation, the insertion into the vector is controlled by a mutex (cf. `registerDuration` in Appendix A).

And to compute the mean and standard deviation of the computation duration for each worker, we have used the `algorithm` library. The `std::transform` call in `double stdvDurationPerWorker()` (Code 1.4) method uses a lambda expression.

Source Code 1.4: Standard deviation computation for π estimation

```
double stdvDurationPerWorker() {
  std::chrono::duration<double> mean = meanDurationPerWorker();
  std::vector<double> diff(durations_.size());

  std::transform(
    durations_.cbegin(), durations_.cend(), diff.begin(),
    [mean] (std::chrono::duration<double>const & x)
    { return x.count() - mean.count(); }
  );

  double sq_sum =
    std::inner_product(diff.begin(), diff.end(), diff.begin(), 0.);

  return std::sqrt(sq_sum / durations_.size());
}
```

A lambda is a **closure**, a callable unnamed function object which can "capture" variables in the current scope [Järvi and Freeman, 2010]. A simple non template lambda, as the one used here (lines 7-8), consists of the following three parts: `[ part 1 ] ( part 2 ) { part 3 }`. The first part is the capture list, where one indicates which variables from the context (current scope) are accessible inside the lambda. The second part is the parameters list, that is where the lambda parameters are declared. In our case, `std::transform` will call in sequence the lambda for each element of the `durations_` vector. The third part is the body of the lambda: it is what is executed by the lambda for each call. For each element `x` of the `duration_` vector, the code `x.count()-mean.count()` is executed (notice that `mean` is captured from the context) and its result (return of the lambda) is written in `diff` vector.

To help manage the tasks and their synchronization induced by futures, it can be useful to have a graphical representation as a task graph (see Section 4.1.1). With futures, a task graph is implicitly created, it is a graph where nodes represent tasks and edges the connection between them (cf. Section 1.7.2). We can consider the asynchronous calls as the graph's nodes and the futures as the graph's edges symbolizing the transmission of data (the associated value is updated by one node and it is read by another one). At the opposite, it is possible, like MetaPASS (Section 2.4.4) to use `std::async` to implement an execution with futures based on a task graph representation.

1.5 Domain-specific parallel libraries

One approach to design parallel algorithms is to use specialized parallel libraries such as OpenBLAS (Section 1.5.1), OpenCV (Section 1.5.1), FFTW (Section 1.5.1), and libvips (Section 1.5.1). They allow developers to use parallel computing without even knowing it by invoking direct functions of a specific domain that are designed to perform efficient parallel computation.

The downside of this method is the implicitness of the mechanisms that are enabled to "optimize" the computation. This has multiple impacts: first, it is difficult to know how the computation is actually conducted because it is hidden by the simplicity of the single call. Second, it is often difficult to tweak, because of the lack of options in the parameters or in the general configuration. This is especially a problem when using special hardware that is not compatible with the

library to perform the computation or attempting to get better performance on current hardware. In order to get performance, such libraries will need to analyze the current platform and to choose parameters. It is difficult to imagine that they will be able to get the best performance on every possible type of platforms. Third, because the parallelization concerns only the library calls, there are implicit synchronization barriers between these invocations that slow down the whole computation or limit some more optimization. The advantages of this type of libraries are (1) the simplicity and (2) all the problems cited previously in Section 1.1.2 are directly tackled by the libraries.

1.5.1 Examples of libraries

OpenBLAS

OpenBLAS [Wang et al., 2013] is an open-source implementation of BLAS (Basic Linear Algebra Subprograms) specification and LAPACK specification in mainly C and Fortran. BLAS is decomposed into three levels. Level one offers an API to perform scalar, vector and vector-vector operations. Levels two and three present matrix-vector and matrix-matrix operations, respectively. LAPACK defines routines to solve "simultaneous linear equations, least-squares solutions of linear systems of equations, eigenvalue problems, and singular value problems. The associated matrix factorizations (LU, Cholesky, QR, SVD, Schur, generalized Schur) are also provided, as are related computations such as reordering of the Schur factorizations and estimating condition numbers. Dense and banded matrices are handled, but not general sparse matrices. In all areas, similar functionality is provided for real and complex matrices, in both single and double precision." [Angerson et al., 1990].

In order to get performance, the library can either use "basic" threads, OpenMP, or special kernels for some architectures.

OpenCV

OpenCV for Open Source Computer Vision is a library for image and video analysis [Culjak et al., 2012] developed in C and C++. It presents more than 2500 optimized algorithms and more than 40000 persons in the user group. The goal of the library is to help developers express complex image or video processing algorithms. OpenCV, during its compilation, uses existing parallel libraries such as OpenMP (Section 1.8.1) available on the system to enable parallelism in the library. It can also use SIMD processing or accelerators such as GPU if CUDA, OpenCL or OpenGL is available. Because it is such a large library, it is difficult to know how a particular functionality is implemented and if some parallelization techniques are used except by digging in the code itself. OpenCV can use OpenMP directives (Section 1.8.1) to parallelize some algorithms.

FFTW

FFTW (Fastest Fourier Transform in the West) [Frigo and Johnson, 1998], is a C subroutine library for computing the discrete Fourier transform. FFTW aims a narrower set of operations than OpenCV but shows performance comparable or superior to vendor algorithms [Frigo and Johnson, 1997].

In order to achieve high performance, the library proposes a pre-processing step to create a computation plan called *wisdom*. It is possible to choose the precision of the wisdom by allocating more or less time to create it. Such an approach allows one to have an efficient portable code where the only prerequisite is to re-compute the plan on each new platform.

Libvips

Libvips [Martinez and Cupitt, 2005] is a parallel large image processing library. When invoking the different routines, the library builds a processing pipeline. The input image will be automatically decomposed into a set of regions, allowing the library to do computations on very large images. The computation is conducted without lock because each used thread will be associated to a private copy of one of these tiles. It uses a mutex to read the input file and another one to write the output file. The computation kernel streams the different tiles thanks to the threads through the pipeline of functions. The library can use multiple CPUs by duplicating the pipeline into each of them.

1.5.2 Implicit execution model

Domain-specific parallel libraries only offer the level of details available in a one-call function or in library-wide parameters. This type of libraries presents a strong difference between the programming model and the execution model. The programming model is what is shown to the developer and the model he or she can interact with. In this case, it is just a set of calls that will do a complex algorithm. The execution model is the model used by the library to conduct the actual computation. Depending on the library, the computation can be represented with task graphs, with OpenMP loops, with vectorization instructions, etc.

The execution model used for executing the tasks is totally hidden by the API, and often no low-level details option is available to customize the computation. Because of the implicitness of the model, there is no way for a developer to analyze the computation that happened and to modify it in consequence.

An enlightened end-user may prefer an explicit model even if it is more involving for him or her. This user may want the ability to analyze the performances obtained by the computation, and if the solution allows it, to tweak the computation at a higher level to improve the performance.

1.5.3 Specificity of the libraries

A parallel library often targets a domain or a set of applications, for example linear algebra for OpenBLAS or image processing for libvips. Because these libraries propose only parallelism during the function calls, synchronization barriers appear possibly slowing down the overall computation.

It would be preferable to have a parallel approach that can be used independently of the domain of the computation. Such library could allow a global approach of the parallelism for the entire computation, and overlap domain specific library calls for specific tasks.

1.5.4 Matrix multiplication with domain-specific libraries

In order to compute the matrix multiplication of our example, we could use a parallel library specialized in linear algebra, e.g., OpenBLAS. It has a subroutine called *gemm*, with different versions handling the precision of the matrices values: `GEMM (&transa, &transb, &m, &n, &k, alpha, a, &lda, b, &ldb, beta, c, &ldc)`. This call here proposes different options about how the library will treat the pieces of contiguous memories `a`, `b`, and `c`: (1) if they need to be transposed (`transa`, `transb`) and (2) what are their leading dimensions (`lda`, `ldb`, `ldc`). `alpha` and `beta` are the scalars for the matrix multiplication ($C \leftarrow \alpha \times A \times B + \beta \times C$).

1.6 Algorithmic skeletons

Another method to design parallel programs is based on algorithmic skeletons. We only focus on the ones suited to C++ and more specifically on Intel TBB because it is well-known and is general purpose. An **algorithmic skeleton** is a piece of software representing a generic parallel pattern [Darlington et al., 1993]. The usage principle is to build a complex algorithm by composing common programming patterns. It is also possible to combine these simple patterns to build more complex ones. A few examples of skeletons are pipeline, farm, parallel composition, divide & conquer, and branch & bound. The algorithmic skeletons can be classified into task parallel and data parallel ones [Poldner and Kuchen, 2008]. In the first category, the parallelism is gained from efficient communication between sub-tasks. In the second category, the skeleton works by performing the same computation over part or all the data in a distributed data-structure. These patterns implicitly define how the parallelization and the synchronization are made when constructing the program which leads to potential optimizations in the scheduling. Intel TBB (section 1.6.2) is an example of a high-level library providing such patterns. Some languages or their standard library propose some form of skeletons. For example, C++ proposes such skeletons associated with the parallel execution policy directives since C++ 17 [Hoberock, 2016].

1.6.1 C++ execution policies

In C++ 17, some algorithms from the standard library can be parameterized with an execution policy. They are part of the *algorithms library* [Hoberock, 2016].

The algorithms are separated into categories: (1) non-modifying sequence operations, (2) modifying sequence operations, (3) partitioning operations, (4) binary search operations (on sorted ranges), (5) other operations on sorted ranges, (6) set operations (on sorted ranges), (7) heap operations, (8) minimum/maximum operations, (9) comparison operations, (10) permutation operations, (11) numerical operations, (12) operations on uninitialized memory, or (13) C library. In the past years, presentations [1] have presented the variety, possibilities, and usage of the algorithms library.

Source Code 1.5: Composition of algorithm with execution policies

[1] At CppCon, 2018: Jonathan Boccara `https://youtu.be/2olsGf6JIkU`, 2019: Conor Hoekstra `https://youtu.be/pUEnO6SvAMo` and `https://youtu.be/sEvYmb3eKsw`, 2020: Ben Deane `https://youtu.be/InMh3JxbiTs`

```
int main() {
  uint64_t sum;
  std::vector<std::vector<int>> vv;
  const size_t sizeVectors = 20000;

  for (int iteration = 0; iteration < 20; ++iteration) {
    vv.clear();
    for (auto i = 0; i < sizeVectors; ++i)
     ↪  vv.emplace_back(sizeVectors, 1);
    const auto startTime = high_resolution_clock::now();

    std::for_each(std::execution::seq, vv.begin(), vv.end(),
      [&sum](auto& v) {
        std::for_each(std::execution::seq,
        v.begin(), v.end(), [&sum](auto& value) {sum += value;}); }
    );

    std::cout << duration_cast<duration<double, milli>>(
      high_resolution_clock::now() - startTime).count() << ",";
  }
}
```

Execution policies are the standard library way to execute these algorithms in parallel. The standard library does not provide any guarantee about data races or deadlocks. We are a step away from the algorithmic skeletons because of the composability and we have to indicate by hand the parallelism. As an example, we present Code 1.5, in which we have two nested `for_each` calls (cf. lines 11-15).

The goal of the snippet of code is to sum up all elements in a matrix of 20000×20000 elements. The first `for_each` (line 11) traverses the matrix in rows and the second (line 13) traverses each row to get each element of the matrix and accumulates it in a variable `sum`. Each `for_each` can be parameterized either by a parallel or sequential policy. The first obvious consequence of this implementation is if at least one `for_each` has a parallel execution policy, there is a data race on the `sum` variable because multiple threads access concurrently to the variable in a write fashion.

Depending on the execution policies set for each loop, on a Windows computer with a Intel Core i7-9750H over 20 computations, we obtained the results summed up in Table 1.3.

Outer execution policy	**Inner execution policy**	**Execution time (ms)**
Sequential	**Sequential**	959.74 +- 23.35
Sequential	**Parallel**	1122.37 +- 24.40
Parallel	**Sequential**	379.66 +- 15.55
Parallel	**Parallel**	509.03 +- 11.69

Table 1.3: Execution time (ms) of nested execution policies over 20 iterations

The best times are with the outer `for_each` parallelized. In the case the outer algorithm is sequential, parallelizing the inner algorithm lowers the overall performance. This is counter-intuitive as one could think that such parallelized code should be faster than its sequential counterpart. So these results indicate that the parallel execution are applied mechanically without any considerations about the nesting. Furthermore, this technique does not provide any guarantee about data race.

The Code 1.6 shows an implementation of the matrix multiplication using the C++ algorithms library and execution policies. The parallelism is made element wise, the

std::for_each is parallelized by simply adding std::execution::par parameter. On a Windows computer with a Intel Core i7-9750H, we have been able to gain $\approx 4\times$ speedup by just adding the parallel execution policy (0.71 s $\pm$ 0.01 versus 2.61 s $\pm$ 0.02 with and without the parallel execution policy respectively, computed over 20 runs with MSVC++ version 14.26).

Source Code 1.6: Matrix multiplication with C++ algorithms library and execution policies

```
int main() {
  // Generates matrices A, B, and C
  // ...
  // Generates indexes
  std::iota(indexes->begin(), indexes->end(), 0);

  // Computes matrix multiplication
  std::for_each(std::execution::par,
    indexes->begin(), indexes->end(),
    [&A, &B, &C, &alpha, &beta] (auto & index) {
      auto col = index / matrixSize;
      auto row = index % matrixSize;

      for(auto i = 0; i < matrixSize; ++i)
        C->at(index) += alpha * A->at(row * matrixSize + i)
          * B->at(i * matrixSize + col);

      C->at(index) += beta;
    }
  );
}
```

1.6.2 Intel Threading Building Blocks

Intel Threading Building Blocks (TBB) [Kukanov and Voss, 2007] is a library which provides concurrent containers, synchronization primitives, and algorithmic skeletons for parallel execution to C++. These skeletons such as map, reduce, pipeline, and fork-join propose a high-level API to express parallel sections of complex algorithms. They work alongside TBB containers, an alternative version of the standard library's containers, which allows concurrent read/write actions. The library is optimized for Intel processors and targets only CPU. It is possible to express a computation with a data-flow graph where lambdas/functors describe tasks.

1.6.3 Composability

The main idea behind the algorithmic skeletons is the composability of skeletons to describe a whole computation. Task-based solutions (Section 1.7.2) can also use graph composability to connect different graphs together to formulate a multi-step algorithm. It should be possible to mimic such skeletons with task graphs to express these general algorithms and use graph composability to have the same behavior as composed skeletons. Internal graphs (or sub-graphs) can express logical parts of complex algorithms, such as communication and computation for the matrix multiplication on a GPU can be a sub-graph linked to two other sub-graphs, one for data decomposition on a CPU, and the other one for aggregation.

1.6.4 Matrix multiplication with algorithmic skeletons

We propose an implementation of the matrix multiplication algorithm with Intel TBB. Code 1.7 is inspired by code available online [2]. We are using two features from the library : the *range* and the *parallel for*. The range helps dividing C arrays (the matrix) into blocks. The `blocked_range` [Robison et al., 2008] is a "recursively divisible half-open interval". The `parallel_for` applies the lambda to each element of the blocks in parallel.

Source Code 1.7: Matrix multiplication with TBB

```
int main() {
  float alpha = 2, beta = 1;
  // Initializes buffers.
  // ...

  // Computes matrix multiplication
  parallel_for(
    blocked_range<int>(0, size),
    [&alpha, &beta] (blocked_range<int> r) {
      for (int i = r.begin(); i != r.end(); ++i)
        for (int j = 0; j < size; ++j) {
          for (int k = 0; k < size; ++k)
            c[i][j] += alpha * a[i][k] * b[k][j];
          c[i][j] += beta;
        }
    });
}
```

1.7 Task-based approach

In this section, we present two types of task-based libraries. The first one allows defining a set of tasks that communicate through a message passing system. The second one allows defining a set of tasks ordered in a graph representation.

1.7.1 Message-driven libraries

Charm++

Charm++ [Kale and Krishnan, 1993] is an object-oriented parallel message passing programming system based on C++. It follows an *asynchronous many-task* model where an application is decomposed into transferable units of work with their inputs. These units are called *chares*. They start their execution upon reception of a message. These chares are presented under the form of C++ objects with all the particularities brought by the language; they present encapsulation of data and inheritance capabilities. A chare will do its computation based on the data it embeds and transferred input data. Data safety is guaranteed by only running one chare on a piece of data. The execution system will associate each chare to a different execution unit thanks to a dynamic scheduler to maximize the performance. The overall workflow is described in a "charm interface" cross-compiled into C++ code [Bennett et al., 2015]. They claim portability without changes on all MIMD (multiple instruction, multiple data) computers.

[2] `http://blog.speedgocomputing.com/2010/08/parallelizing-matrix-multiplication_8641.html`

HPX

HPX [Kaiser et al., 2020] (for High Performance ParallelX) is a C++ library for exascale computation. Exascale computing is an architecture dealing with the parallelism and communication between nodes achieving 10^{18} Tflops. HPX AMT (Asynchronous Many-Task) runtime system presents an API conforming to the C++ standard for local or remote computation, and implements an asynchronous execution model that semi-automatically parallelizes user code.

HPX uses C++ futures (Section 1.4) to transform sequential algorithms into asynchronous executions with a wait-free property; computation and communication can be overlapped with the usage of C++ 20 `co_await` operator. They have created "local control objects" (LCO) for synchronization mechanisms. One of them, the data-flow, allows the execution of a piece of code on a separate thread when the values that it depends on become available. The thread usage allows to get a minimal overhead for synchronization and context switching.

The HPX scheduler comes with a work stealing algorithm and an automatic load balancer. HPX has its own C++ implementation of the C++ 17 algorithms (Section 1.6.1) and C++ 20 concurrency facilities. Their work served the design for the C++ parallelism technical specification [3]. Compared to other AMT systems, HPX brings a "future-proof C++ conforming API" and an exposed asynchronous programming model.

1.7.2 Graph-based libraries

Another model for parallel applications is based on a graph representation for a computation. We discuss the different graph models in the following section, with some examples of libraries based on graphs.

Type of graphs

The terms and notions introduced in this section come from two recognized sources: Timothy Mattson's book [Mattson et al., 2004] and Olivier Sennen's book [Sinnen, 2007]. In particular, we focus on two chapters of The Encyclopedia of Parallel Computing describing two types of graphs: (1) the task graphs [Robert, 2011] and (2) the data-flow graphs [Dennis, 2011].

The main model of graph is the *task graph*. The basic idea is to decompose a large problem into smaller interdependent chunks called tasks. Tasks, as graph nodes, are mainly defined by their inputs and outputs, what they receive and produce, and the computations they achieve. A task can only start when all of its antecedents have completed their work. This is the dependence relationship between tasks that is represented by graph edges. A task executes its kernel, terminates, and then the results are passed to its successors without being streamed.

A task graph needs to be mapped to the process unit (*e.g.*, a thread) into the computational node in order to be executed. That is the role of a scheduler to choose the process unit that will be in charge of running each task's kernel. This matching mechanism is complex and gathers multiple types of computation such as work stealing or work balancing. The work stealing is a process that allows an idle worker to steal work from another busy worker. The load balancing is a process

[3] `https://stellar-group.org/2016/03/hpx-and-cpp17/`

that balances the work among workers. StarPU 1.7.2 proposes different APIs to customize an application scheduling. Usually, task graph libraries use a *Directed Acyclic Graph* (DAG), a graph without directed cycle, in order to avoid deadlocks. A deadlock can occur with directed cycles because two tasks become inter-dependent and information from the computation in progress is needed to break the cycle.

The second kind of model is the data-flow graph which presents a different meaning of edges. In a data-flow graph the nodes are called actors. These actors will apply their kernels to the data they receive. The graph's edges represent the path for a data to move between two actors and also the dependency/precedence relationship. An actor is *enabled*, i.e. it processes data, when it receives them. Which means that more than one actor can be enabled at the same time.

For example, some libraries such as Legion (Section 1.7.2) add metadata to each input to determine if the data are ready to be computed. Others like HTGS and our proposal Hedgehog fire their tasks when one of their input data arrive in one of the task's queues. We will present HTGS and Hedgehog's model in Section 3. The data-flow graph computation representation presents naturally parallelism, because if several tasks have their dependencies met, they can be fired simultaneously.

In the following sections, we present different libraries using graphs.

Anthill

Anthill [Teodoro et al., 2012] is a data-flow graph library. It targets clusters of heterogeneous nodes. It decomposes its computation into computational nodes called *filters*. Each filter is duplicated into each node of the cluster. A filter presents multiple input streams with one input queue for each, creating their own events. An event scheduler associates a task to a filter when an event is received. The default policy follows a first-come first-served policy. Because the devices have different executions depending on the data they receive, the framework uses a *performance estimator* utilizing a model learning algorithm (k-nearest neighbors' algorithm) to improve performances in different runtime environments.

Qilin

Qilin [Luk et al., 2009] is a framework for heterogeneous single nodes. Qilin provides an API to parallelize sections of code. Through the API, an abstract directed acyclic graph is created to represent the whole computation.

The main proposition of Qilin is its automatic adaptive mapping. The goal of the map process is to associate a task ready to be executed to a processing element, a CPU or a GPU for example. The mapping bases its decision on (1) all previous executions of Qilin that store execution times of calls on the different processing elements, and (ii) a training drill when an execution first runs.

It also extracts automatically parallel sections and generates TBB (see the section 1.6.2) code for CPUs and Nvidia CUDA [Garland et al., 2008] code for GPUs that will be dynamically compiled at runtime. In the case of GPU computation, if Qilin detects that the memory used exceeds the available memory on the device, the graph is split into subgraphs that will be executed sequentially.

StarPU

StarPU [Augonnet et al., 2011] is a C library to execute parallel tasks over heterogeneous hardware on a single node with a task graph representation. It proposes a unified approach (a codelet) to implement a task. End-users can use additional libraries like Nvidia CUDA [Garland et al., 2008] or BLAS routines to implement the kernel inside the same codelet. The codelet will be then offloaded to the execution unit used, CPU or accelerators. In addition to the unified execution model, StarPU proposes a generic scheduling framework. It enables users to customize low-level scheduling decisions, such as work stealing or work balancing, with high-level calls or per-task performance models.

Legion

Legion [Bauer et al., 2012] is a task graph library for heterogeneous nodes. The library defines "logical regions" in memory as collections of objects. When creating a task, the end-user defines explicitly the task's input data plus the attached properties. The properties include the logical regions privileges (read, write, or both), the region organization (if it is an array of structures or a structure of arrays), the partitioning and the coherence (the types of treatments that a task can do in another task's logical region). The runtime system will use these different logical regions' properties to define the scheduling between these tasks and handle tasks reordering. It can also duplicate shared read-only logical regions to improve parallelism between multiple tasks.

Uintah

Uintah [Meng and Berzins, 2012] is a parallel asynchronous many-task runtime system with multiple simulation components made for exascale computation designed for multi-physics simulation. The simulation components as of January 2015 are *ARCHES* (combustion simulation component), *ICE* (compressible flows component), *MPM* (particle-based for structural mechanics component), and *MPM-ICE* (fluid structure interactions simulation) [Holmen et al., 2017].

The simulation spatial grid is divided into patches by a scalable regridder. Patches are assigned to nodes thanks to a measure-based load-balancer. A patch is a chunk of data used by a node and shares ghost values (interface data) computed by other nodes. The patch is stored and managed by a local data warehouse; it is updated by MPI calls.

The computation is represented with a task graph (a directed acyclic graph) where the task dependencies are determined by the required data and data usage (read, write, and read/write) for individual tasks. The Uintah runtime system schedules the tasks on the patches.

At each tick of the simulation, the data warehouse is updated with the computed data and old data can be removed. The MPI scheduler creates multiple worker threads on each multi-core node with one MPI rank per CPU core. All workers in a node have access to all variables contained in the node. A *unified scheduler* supports GPU tasks with a combination of MPI, POSIX Threads, and CUDA.

HTGS

HTGS [Blattner et al., 2017] is the data-flow library that inspired Hedgehog. As such, they share some important concepts and design decisions. It was developed at NIST by Timothy Blattner and Walid Keyrouz.

It is based on an explicit data-flow graph representation where each vertex has a defined role. Its execution relies only on the OS scheduler, no external mapper or scheduler are used. The explicitness of the model and a cost-free feedback mechanism allow the user to understand how the computation is conducted and to improve it. This is made possible because the designed model is exactly the one being executed. Details about the architecture and the model are given in Section 3.

Threading model

We have observed that the libraries in this section base their representation on task graphs. Their execution mechanism maps dynamically, at runtime, a task ready to be executed to a thread from a pool with a scheduler. The decision is made from a variety of properties depending on the library, state of needed data, communication costs, etc. In some cases, other algorithms like work stealing or load balancing help scheduling decisions. Multiple schedules are possible for a given algorithm; the scheduler aims to find a schedule that minimizes overall execution time. The general decision problem associated with the scheduling problem is a NP-complete problem [Sinnen, 2007].

Another solution, used in HTGS, would be to statically bind a thread to each task. In most cases, the scheduler will do the association between a task and a thread, and then the operating system schedules the thread. With this method, no external scheduler is needed, the sole OS scheduler can be used to organize the tasks on the available hardware. There is a major advantage to this technique: no extra overhead is added by a scheduler.

Directed cycles in task graphs

The other common property of these libraries is the usage of directed acyclic graphs. Directed cycles could be problematic as they are a cause of deadlocks between tasks. The graph can only terminate if all its vertices terminate. A node termination is linked to its predecessor vertex; it can usually terminate if all its predecessor nodes terminate. If there is a directed cycle, there is always an alive predecessor for each vertex of the cycle.

We advocate that an explicit model that is easy to understand could allow the end-user to (1) create directed cycles in his or her graph to increase the expression capabilities of the library, notably useful to model a loop (cf. matrix multiplication in Section 1.7.2) and (2) express termination conditions to break the cycles depending on the algorithm implemented (in the same example, to stop the loop).

Matrix multiplication with a graph

In this section, we will give an overview of how to represent the parallel execution of the matrix multiplication of our example based on a data-flow graph as in HTGS. We will discuss this approach with much more details in Chapter 3.

For this example, we need to decompose the matrices into blocks to operate on them in parallel as shown in Figure 1.1. This allows us to decompose the algorithm into independent tasks and to manage the production order of blocks.

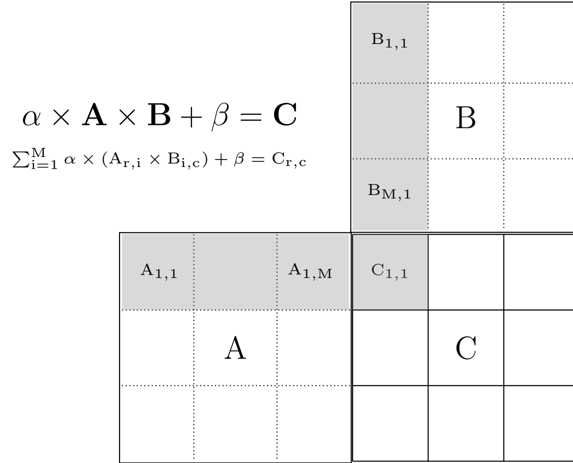


Figure 1.1: Simple matrix multiplication decomposition in blocks

This workflow is presented in the Figure 1.2 (circles are tasks and diamonds are management steps). The algorithm is decomposed into three tasks: (1) the matrices decomposition into blocks, (2) a multiplication between matrices A and B blocks and (3) an accumulation into blocks of matrix C. In order to compute a block of C at position (r, c), we need to multiply appropriate blocks of A and B (all $A_{r,i}$ and $B_{i,c}$) together with the α value creating temporary blocks. We can then accumulate these temporary blocks into the corresponding block of C plus at the end the β value.

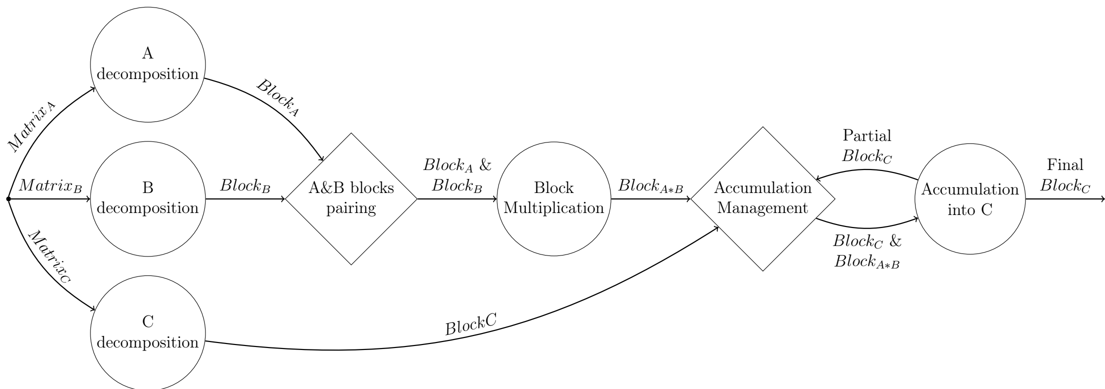


Figure 1.2: Matrix multiplication with a graph approach

However, the tasks may receive decomposed blocks of matrices A, B, and C out of order if the productions of blocks from these matrices are made in parallel. In that case, additional management phases would be required. One will be in front of the block multiplication to gather compatible blocks of A and B. The other will be in front of the final accumulation to manage the different aggregations of the temporary blocks into C.

There are three major steps: (1) decomposition of matrices A and B, (2) multiplication of block matrices and (3) accumulation into C matrix, plus two management phases: (1) gather compatible blocks of A and B and (2) manage accumulation into C.

1.8 Multi-paradigm models

1.8.1 OpenMP

The OpenMP Task Model [Dagum and Menon, 1998] is a multi-faceted library born in 1997. At its core, it proposes to parallelize loops on a single node through pragma annotation of C/C++/Fortran sequential code. It offers a large variety of options to customize the access, computation and synchronization on these loops. Internally, it uses a fork-join model to handle the threads and master/minion design to manage the computation.

This methodology comes with some drawbacks. First, it needs a sequential code that will be later augmented. Parallelism is not considered as a first-class citizen. Second, if parallelism is considered, the algorithm needs to be mainly centered around loop constructs. Complex algorithms may be difficult to design that way. Third, synchronization barriers appear between parallel sections because OpenMP only considers the fork-join model for those loops. Finally, the library presents some complexity due to its options. In order to get maximum performance, deep knowledge is required to fully exploit the library.

Last versions of the library provide accelerator support through direct compiler code generation and a task-based approach to represent parts of the algorithm with lambdas.

1.8.2 OmpSs-2

OmpSs-2 [Catalán et al., 2019] is a programming model built upon OpenMP (Section 1.8.1) among others. The usage is close to OpenMP, the library works by annotating sequential code. The main differences come from the internals, where OpenMP uses a fork-join model, OmpSs-2 uses a pool of threads. The support for accelerators is achieved by calling kernels' implementation for these targets. The library proposes a more implicit programming model. The end-user does not need to declare the parallel regions because the parallelism is created from the beginning of the program.

1.8.3 Kokkos

Kokkos [Edwards et al., 2014a] is a C++ library targeting manycore architecture. Kokkos focuses on the parallelism and on the data access pattern to get performance. The programming model consists of two parts: (1) the thread parallel execution (the execution space) and (2) the multidimensional arrays (the memory space).

It targets performance portability, a user code will be compiled specifically for various devices to get performance equivalent to hand tuned code. This is possible due to a unified abstraction for CPU and GPU architectures. Kokkos achieves this by using different back-ends depending on the targeted hardware: CUDA for Nvidia GPUs, POSIX threads (Section 1.3) or OpenMP (Section 1.8.1) for CPUs.

The Kokkos multidimensional arrays have a polymorphic data layout. The data layout can be changed from an array of structures to a structure of arrays to improve the performance on a specific target.

The Kokkos abstraction for an HPC environment is a network of computing nodes with manycore devices. They therefore propose two levels of parallelism, one

at the memory distribution level through MPI, and the other one at the thread level in each of the node.

Kokkos proposes two types of parallel patterns: (1) parallel algorithmic constructs with `parallel_for`, `parallel_reduce`, and `parallel_scan`, and (2) parallel tasks. These Kokkos tasks (2) form implicit directed acyclic graphs with Kokkos future constructs.

1.8.4 Parallelism considerations

OpenMP relies on a sequential code that is later augmented with directives. This means that the base computation needs to be sequential and the parallelism is considered as an afterthought. If the parallelism is taken into account during development, the code needs to be written around array structures to benefit from the API.

It seems preferable to have access to a library with an explicit model for the parallelism that drives the end-user to express his/her algorithm with a set of concurrent pieces of code. This way, the algorithm can naturally expose parallelism and can use special data structures inside the concurrent sections to express more fine-grained parallelism.

1.8.5 Matrix multiplication with OpenMP

To implement a version of matrix multiplication with OpenMP, we started with a triple loop implementation as shown in Code 1.8. Then, we added 2 directives: (1) for defining a parallel section and how the variables are shared among the threads, and (2) for parallelizing the outer loop with its static scheduling (the library will divide the outer loop into chunks that will be distributed to each thread in a circular order).

Source Code 1.8: Matrix multiplication with OpenMP

```
1  int main () {
2    const int size = 10;
3    float alpha = 2, beta = 3;
4    int i,j,k;
5
6    float a[size][size];
7    float b[size][size];
8    float c[size][size];
9
10 #pragma omp parallel shared(a,b,c) private(i,j,k)
11   {
12 #pragma omp for schedule(static)
13     for (i=0; i<size; i=i+1)
14       for (j=0; j<size; j=j+1) {
15         for (k=0; k<size; k=k+1)
16           c[i][j] += (a[i][k])*(b[k][j]) * alpha;
17         c[i][j] += beta;
18       }
19
20   }
21 }
```

1.9 Parallel languages

Parallel computing can be achieved using parallel programming languages where parallelism is a first-class citizen, they define constructs to express parallelism. Chapel (Section 1.9.1), Rust (Section 1.9.2), and Go (Section 1.9.3) are three examples.

1.9.1 Chapel

Chapel [Gmys et al., 2020] (Cascade High Productivity Language) is a parallel programming language aiming at single node and cluster computations. It mainly uses a task model to build parallel constructs. At runtime, Chapel will spawn a task and run the main function. To generate parallel executions, it is possible to use *unstructured tasks*, *structured tasks* or to exploit data parallelism. An unstructured task will execute its kernel and then terminate. At the same time, the main task continues. For a structured task, the parent task will wait for its child tasks to complete before continuing. Data parallelism (distribution of data across different execution units executing in parallel) is achieved through *forall-loops*, *ranges*, *domains*, and *arrays* constructs. Task parallelism comes with problems such as data races and deadlocks that the user needs to tackle. It provides work stealing schemes to improve task scheduling.

1.9.2 Rust

Rust is a recent multi-paradigm language that first appeared in 2010. One of its main concerns is to help the programmer build safe and efficient concurrent programs [Klabnik and Nichols, 2019a]. Rust defines two types of code sections: safe and unsafe.

In a safe section, Rust will guarantee memory safety (access in read and write for data race prevention) by checking the code at compile-time. By default, the language is conservative, and it prefers rejecting a correct piece of code rather than accepting an incorrect one. Memory checking provides some guarantee on concurrent computation.

In an unsafe section, it is possible to (1) dereference a raw pointer, (2) call an unsafe function or method, (3) access or modify a mutable static variable, (4) implement an unsafe trait, and (5) access fields of unions [Klabnik and Nichols, 2019b].

It is recommended to wrap an unsafe section in a safe section. The idea is to add guarantees or checks around unsafe code to contain the unsafe portions of code. This way, Rust can provide guarantees over this wrapper like over all safe parts of the code.

This distinction is important in Rust as safe sections are guaranteed by the language to be free of data races [Qin et al., 2020].

1.9.3 Go

Go is also a recent multi-paradigm language that first appeared in 2009. It is meant to be a simple language that provides a way to do concurrency considered essential from the beginning of the language. It provides two main primitives to express concurrent programs: (1) the goroutines (go-coroutines) and (2) the channels.

A goroutine is a concurrent function [Doxey, 2016]. They choose to provide this construct instead of threads to hide low-level details to the developer. The runtime is in charge of executing these goroutines in a thread or in many threads depending on the architecture. If one thread blocks, the system is also in charge of moving all other goroutines to different threads to avoid all of them to wait.

The channels are the communication and synchronization mechanism. It allows sending messages between goroutines and waiting for them. Lower level synchronization techniques also exist in the language such as mutexes. In the language philosophy, these primitives are used for simple tasks while the channels are structural high-level patterns. They have a proverb about it: "Do not communicate by sharing memory. Instead, share memory by communicating". One way to proceed is to structure the program around the idea that only one goroutine is responsible / holds a piece of data, and the property is passed by communicating.

The motto is a guide used to avoid data races. In addition the language possesses a data race detector. The language can also detect a deadlock when the whole program is frozen. In paper [Tu et al., 2019], the authors studied the concurrency safety of the language and pointed some limits.

1.9.4 Matrix multiplication with a parallel language

In Code 1.9, we present a full implementation of the matrix multiplication algorithm with the Chapel language. Usually, a Chapel developer would have directly used the *LinearAlgebra* module using BLAS routines behind.

The code could be understood as a sequential code as it is only composed of a for-loop construct. Chapel is a parallel language and the for-loop, `coforall`, is a parallel loop that will compute, in parallel, blocks of matrix C. `D2` is a 2D block dimension of the matrix that allows traversal of the matrix in blocks.

Source Code 1.9: Matrix multiplication with Chapel

```
use Random;

config const size = 10000;
config const alpha = 10;
config const beta = 5;
var A : [1..size, 1..size] real;
var B : [1..size, 1..size] real;
var C : [1..size, 1..size] real;
const D2 = {1..size, 1..size};

fillRandom(A);
fillRandom(B);

coforall (i,j) in D2 do {
  for k in 1..size do { C[i,j] += alpha * A[i,k] * B[k,j]; }
  C[i,j] += beta;
}
```

1.10 Conclusion

In this chapter, we have presented approaches to exploit parallelism on various hardware platforms. Parallelism is complicated and there is no perfect solution that fits all problems and users. However, we have found interesting properties that could be used to express parallelism and to suit a large spectrum of users.

An interesting property is the accessibility of the approach. The model needs to be well defined and explicit. What is presented to the programmer should be reused during the execution to help him/her understand how the computation is conducted. Moreover, it should not act as a black box, but it should rather allow the user to tune the execution to improve the overall performances. Also, since we are designing a parallel approach, the parallelism should be intrinsic to the model, like what we have described with the parallel languages.

A good base model with this property is the data-flow graph model. This model is well defined with its vertices describing the computational kernels and its edges indicating the flow of data. One can consider using the graph without any adjustment from the library. Provided that the tasks run on different threads, the model proposes an intrinsic parallelism. However, the model is commonly used with an execution model where there is a scheduler along with a pool of threads; the HTGS model proposes an interesting alternative with tasks statically bound to threads and an execution that only relies on the OS scheduler. This novel approach reinforces the idea of explicitness, as it is easier to think about the algorithm representation without any interface doing NP-complex mapping.

Another major asset of such an approach is the idea of open computation. Even if the structure is well defined, the data-flow graph model is totally compatible with other approaches to express kernels in the graph's nodes. It has also no specific domain target, as long as the algorithm is expressed under the form of such graphs, it is possible to express linear algebra, image processing, or other types of applications.

In addition, if a data-flow graph can be considered as a node itself, it can be embedded in another graph, sharing the same composability property as the algorithmic skeletons. This is a major property as it helps with abstracting parts of an algorithm and therefore helps a user to reason and to design the whole algorithm's structure. It also helps with sharing codes as it suffices to only share a graph to share a full algorithm for reuse by others. We can also imagine building libraries of graphs for a given set of applications, like what OpenBLAS did for linear algebra by implementing BLAS routines.

Chapter 2

Metaprogramming techniques in C++

The parallel run-time systems presented in the previous chapter usually require the developer to design the parallel process using abstractions (the programming model) that may have an impact on the execution time (an execution plan is produced from the programming model). Moreover, it seems important to bring as much checking as possible to the programming model before starting the execution.

As the programming model is usually known before execution (*e.g.*, a data-flow graph), it can be considered analyzing the graph during the preparation phase of the program, in order to avoid some checking at run-time that could lead to overhead and to detect as soon as possible potential incoherence. Metaprogramming is usually used to manipulate models (representing programs), analyze them, produce new models and ultimately produce the final executable code. There exist many metaprogramming techniques that differ on what can be done and how to achieve it, as shown by Damaševičius and Štuikys [Damaševičius and Štuikys, 2008] that have identified and classified 35 concepts around metaprogramming presented in Table 2.1. Among the possible techniques, the C++ language offers possibilities of metaprogramming during the compilation steps, which enables designing libraries that perform some computations at compile-time, such libraries are called *active libraries* [Veldhuizen and Gannon, 1998].

With the objective of providing a library-based solution that is portable, we chose to focus our study on what is available with C++. The latest C++ norms bring new facilities that enhance a programmer's capabilities to design complex algorithms at compile-time. Especially the "generalized constant expression" concept (mainly known through the *constexpr* keyword) has gained a lot of capabilities that help in the design of active libraries.

In this chapter, we first present in Section 2.1 different taxonomies that try in their respective times to classify metaprogramming techniques. We follow this presentation by a discussion about different types of metaprogramming techniques enabled by the C++ language. It includes trivial metaprogramming techniques in Section 2.2, template metaprogramming techniques in Section 2.3 and constant expressions metaprogramming techniques in Section 2.4.

2.1 Metaprogramming taxonomies

Metaprogramming is a programming technique to manipulate programs, either for analyzing or generating programs. A metaprogram can be defined as a program that manipulates other programs as its data [Czarnecki and Eisenecker, 2000]. Thus, a

metaprogram can take programs either as input or output, or both.

Concept class	Concept
Transformation	Manipulation
	Transformation
	Modification
	Adaptation
	Translation
	Preprocessing
Generation	Code generation
	Instantiation
	Weaving
Metaprogram	Template
	Generic component
	Macro
	Metaprogram
	Metaspecification
Levels of abstraction	Representation
	Abstraction
	Encapsulation

Concept class	Concept
Generalization	Construction
	Generalization
	Parametrization
Separation of concerns	Analysis
	Concern separation
Reflection	Reflection
	Introspection
Metadata	Metadata
	Parameters
Other concepts	Metaobject protocol
	Traits
	Theorem proving
	Partial evaluation
	Inspection
	Specialization
	Runtime execution
	Optimization
	Interpretation

Table 2.1: Concepts related to metaprogramming from Damaševičius and Štuikys' analysis

In the last decades, Pasalic [Pasalic, 2004], Sheard [Sheard, 2001], Damaševičius and Štuikys [Damaševičius and Štuikys, 2008], and Lilis and Savidis [Lilis and Savidis, 2019] proposed multiple taxonomies to classify metaprogramming techniques, each focusing on different aspects, which means that none of these taxonomies is considered more accepted than the others.

In order to discuss these classifications, we first introduce some terms. We call *metalanguage* the language used to express a metaprogram, *object-program* a program manipulated by a metaprogram, and *object-language* the language that the metaprogram manipulates, i.e. the language used to express the object-program(s) treated by the metaprogram.

2.1.1 Pasalic's taxonomy

In 1995, Emir Pasalic proposed to classify metaprograms according to three axes: generator vs. analyzer, homogeneous vs. heterogeneous and open vs. closed.

A *generator* creates its output, an object-program, based on its inputs (a compiler could be considered as a generator [Aho et al., 1986]). An *analyzer* analyzes an object-program and computes some results (a proof program is a good example [Hoare, 1971, Binkley, 2007]). It is possible to have metaprograms that enable creating both generator and analyzer, while others like C++ template metaprogramming (cf. section 2.3) are considered by the authors to do just one, they assume that it is only possible to write generators at that time.

The difference between *homogeneous* and *heterogeneous* metaprograms lies in the comparison between the metalanguage and the object-language. If the metalanguage and the object-language are the same, the metaprogram is homogeneous; if they are different the metaprogram is heterogeneous. In a way, every language using

strings can be heterogeneous or homogeneous, because the strings can represent metaprograms. C++ template metaprogramming can not be clearly categorized in one of these two categories. Template metaprogramming follows a syntax defined by the language, so template metaprogramming *is* part of C++. However, it is undeniably different to code a "usual" program in C++ than using template metaprogramming. We do consider template metaprogramming heterogeneous, while, by contrast, C++ generalized constant expressions are homogeneous as they allow using the same syntax as usual code (cf. Section 2.4).

Finally, whether the metalanguage is *open* or *closed* is the metalanguage designer's choice. A closed metalanguage can only manipulate an object-language known in advance. An open metalanguage offers features that can encode and manipulate multiple object-languages chosen by the designer of the metalanguage. In the cases of template metaprogramming and generalized constant expressions, they are both closed, as they are meant to manipulate C++ code only.

2.1.2 Sheard's taxonomy

In 2001, Tim Sheard proposes its taxonomy [Sheard, 2001]. He shares with Pasalic the generators vs. analyzers and homogeneous vs. heterogeneous differences. The taxonomy adds a distinction between static/run-time generator and manually/automatically annotated.

A static generator generates a program that is "written to disk" and later exploited by a compiler. A run-time generator generates the object-program and executes it right after. For the author, if the generated code is a generator itself we have a form of multistage programming [Taha, 1999].

C++ template metaprogramming is mostly a static generator, the code issued from the instantiation of a metaprogram (i.e. a template, cf. section 2.3) is then compiled.

He calls *staging annotations*, notations that separate static and dynamic sections of code (in the metalanguage and in the object-language respectively). These notations can be placed *manually* by a developer (manually staged system) or *automatically* by a process (automatically staged system). For the author, the partial evaluation [Jones, 1996] technique is the pioneer of the automatic staging annotation.

2.1.3 Damaševičius and Štuikys' taxonomy

In 2008, Robertas Damaševičius and Vistautas Štuikys propose a new taxonomy [Damaševičius and Štuikys, 2008] to classify metaprogramming techniques (summarized by Figure 2.1) based on their analysis (Table 2.1). The taxonomy focuses on two classes: *structural* and *processing* classes. The different concepts are sorted in these two concept classes presented in Figure 2.1.

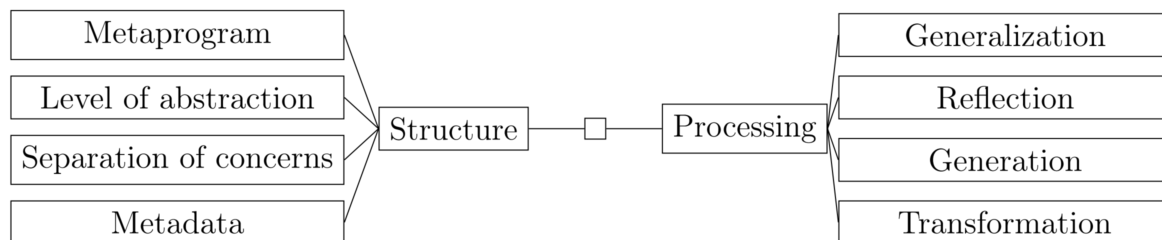


Figure 2.1: Damaševičius and Štuikys' taxonomy

The Structural concept class presents the metalanguage abstractions. It is *static* and *tool-dependent* because it depends on the metalanguage designer and the capacity of the language. It is used *during the construction* of the metaprogram. This concept class is composed of different concepts: metaprogram, level of abstraction, separation of concerns and metadata. A metaprogram is a generic component representing similar component instances and containing different functionalities. The layers of abstraction model a semantic system that represents different aspects of design. The separation of concerns is a process to break a problem into orthogonal distinct tasks implemented separately. Metadata are annotations to describe properties on a specific layer of abstraction.

Processing concepts present the metalanguage operations. They are *dynamic* because they describe some processing rather than tools and *domain-dependent* because they can be implemented using different meta and domain structures. The transformation process allows the program to change form to another one. The generation process allows to create a target system from high-level specifications. The reflection is the ability to observe and change its structure and behavior. The generalization is the transformation of a domain specific component to a generic one, making it more widely usable.

The authors remark that the structural and processing concepts are inter-dependent. They sum up these relations in Figure 2.2, for example generation processing depends on metaparameters described as metadata.

2.1.4 Lilis and Savidis' taxonomy

In 2019, Lilis and Savidis propose their own taxonomy of metaprogramming [Lilis and Savidis, 2019] presented in Figure 2.3.

This taxonomy presents four dimensions to classify metaprogramming languages: (1) the relation between the metalanguage and the object-language, (2) the source location, (3) when the metaprogram is evaluated, and (4) the metaprogramming model.

The taxonomy studies, like the previous ones, the relation between the metalanguage and the object-language. What was called homogeneous and heterogeneous is respectively called indistinguishable and different here. Lilis and Savidis add a third classification called *extension*, where the metalanguage reuses the base language while adding new syntax to express the code of the metaprogram. The classification of template metaprogramming and constant expression in C++ is not obvious. We can make a separation between the "usual" C++ language and the language specific to the templates, making template metaprogramming heterogeneous. However, the

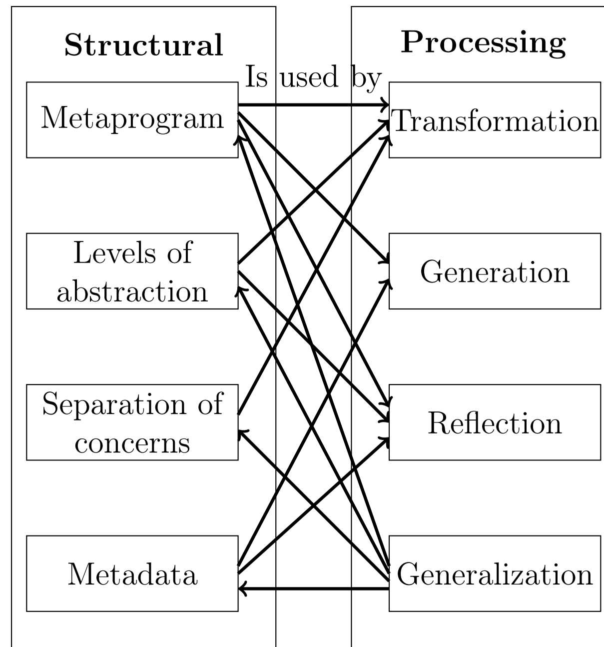


Figure 2.2: Relations between structural and processing concepts

"template language" is part of the C++ norm, making it more an extension of the "main" language. Constant expression can be seen homogeneous with the "main" C++ language, although the entire language is not available inside constant expressions (cf. Section 2.4).

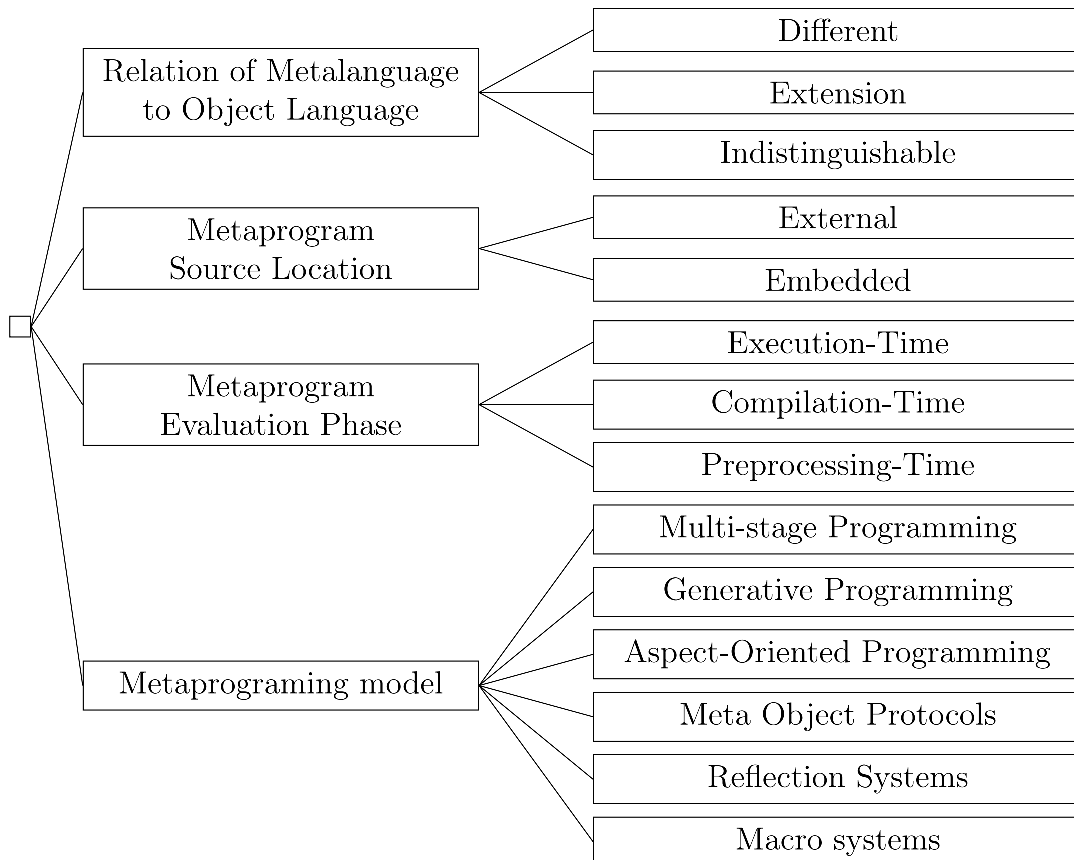


Figure 2.3: Lilis and Savidis' taxonomy

They also classify a metaprogram depending on its source location, it can be located either (1) inside the code it modifies or (2) it can be independent. In C++, template metaprogramming and constant expressions are inside the code they modify.

Another metric used to classify metaprogramming techniques is when the code is evaluated. They have distinguished three periods: (1) during preprocessing, (2) during compilation, and (3) during execution. For example, C++ macros are executed during preprocessing, C++ template metaprogramming or constant expressions are evaluated during compilation, and Java reflection is made during execution.

Finally, metaprograms can be classified depending on the model they follow. They distinguish six categories of models:

1. macro systems (cf. Section 2.2.2) available in C/C++.
2. reflection systems: possibility for a program to observe and modify its own structure like Java.
3. metaobject protocols: possibility for a system to manipulate the original object system behavior and implementation. The authors cite Smalltalk and the Common Lisp Object System (CLOS) as example of such systems.
4. aspect-oriented programming: addition of *advices* (additional behaviors) to be executed at *join points*, AspectJ is an example of extension of the Java language for aspect programming.
5. generative programming: transformation and creation of a program based on algorithm or program representation, C++ template metaprogramming is an example of such programming (cf. Section 2.3).
6. multistage programming: making available to the developer levels of evaluation [Glück and Jørgensen, 1996] of the program computations with a special syntax called *staging annotations*. Languages supporting multistage programming typically support an unbound number of stages and are called multistage languages. An example of language that allows multistage programming is MetaML.

2.1.5 Conclusion

As we have written, there is no taxonomy that dominates in the literature compared to the others. We have seen that these taxonomies can follow orthogonal approaches to classify metaprogramming methods and should be considered as complementary. Pasalic and Sheard's taxonomies base their classification on the type of computation done whether it is generation or analysis. Damaševičius and Štuikys' taxonomy adds structural and processing concepts notions. Lilis and Savidis' taxonomy, the most recent one, is closer to the latter one while adding the model of metaprogramming used. It seems the most wide and precise compared to the previous ones.

In the next sections, we cover metaprogramming techniques in C++. We have chosen the C++ language because in addition to the performance available from the language at runtime, it allows one to do more complex computation at compile-time with the introduction of new facilities that result from the need of developers to run complex programs at compile-time. The language presents different forms of metaprogramming techniques that have evolved following the different norms. C++

template metaprogramming is a form of functional programming (so not the style of programming that is common with C++) that allows one to manipulate mainly types and integers at compile-time. Constant expressions use the "classical" C++ syntax to write programs and allow one to manipulate (and generate) any "data structure" (values of primitive types or POD, instances of classes) that are known at compile-time. They allow designing complex programs more easily.

2.2 Trivial metaprogramming in C++

Luc Touraille in his PhD thesis [Touraille, 2012] proposed a survey of metaprogramming techniques with C++. We recap the different methods, from the trivial ones like string manipulation and macros in this section, to metaprogramming techniques like template metaprogramming and constant expressions in the next sections. The first technique that we cover is the usage of strings to represent programs, presented in Section 2.2.1. Then, we do a quick overview of macro mechanisms in Section 2.2.2. Next, we will present different ways of doing template metaprogramming in Section 2.3. Finally, in Section 2.4, we elaborate on the usage of the *constexpr* keyword.

2.2.1 Metaprogramming with strings

This is the most immediate technique as every language with string capabilities can achieve it. We present the following Code 2.1 that will generate a new code file 2.2 which, upon compilation, will print what has been given as parameter to the program of Code 2.1.

Source Code 2.1: Metaprogram based on strings

```
1  #include <fstream>
2  
3  int main(int argc, char ** argv) {
4    std::ofstream ofs ("meta.cpp", std::ofstream::out);
5  
6    ofs << "#include <iostream>\n\n";
7    ofs << "int main() {\n";
8    ofs << "  std::cout << \"" << argv[1] << "\";\n";
9    ofs << "  return 0;\n";
10   ofs << "}\n";
11 
12   ofs.close();
13   return 0;
14 }
```

Source Code 2.2: Output of the previous code with "blablibliou" given as argument

```
#include <iostream>

int main() {
  std::cout << "blablibliou";
  return 0;
}
```

When developing with this technique, there is no advanced syntactic checking and we do not have access to help given by integrated development environment, because we do not manipulate the actual language. Transforming an existing source code is possible but difficult, because it relies only on text operations.

2.2.2 Metaprogramming with macros

Macros in C++ are constructs (based on C constructs [Ritchie et al., 1988]) that are executed by the preprocessor during the 4th phase of the translation process. Theses constructs are defined with directives. Directives start with the # character, and represent instructions, mainly `#if` for conditional statement and `#define` that allows defining a "code pattern" (possibly with parameters) that will then be "called" to be applied/expanded anywhere in the code. The `#define` directive is used to define macros. Their purpose is to substitute text but without syntactic checks.

For example, in the following Code 2.3[1], the pattern `CREATE_OPERATOR` with the parameter `OP` is defined (lines 1 to 3). It defines the pattern of the code to write the overload of the unary operator `OP` for the `Vec` class. The pattern `CREATE_OPERATOR` is then expanded for the operators +=, -=, *=, and /=, by giving respectively the argument +, -, * and / (lines 5 to 8). It avoids to repeat by hand the writing of this structure, which avoids errors and lowers maintenance costs.

Source Code 2.3: Metaprogram based on macro

```
1 #define CREATE_OPERATOR(OP) \
2   Vec& Vec::operator OP##= (const double x) \
3   { return apply([x](double y) { return x OP y; }); }
4
5 CREATE_OPERATOR(+)
6 CREATE_OPERATOR(-)
7 CREATE_OPERATOR(*)
8 CREATE_OPERATOR(/)
```

Operator overloading is a feature from the C++ language that is used to customized operations, in our case, +=, -=, *= and /= for objects of class `Vec`. Overloading is a type of polymorphism allowing functions (and thus operators) to have several implementations that can be distinguished mainly by the types of their arguments. The type deduction mechanism of C++ finds the right implementation to call based on the types of the arguments when calling a function. In our example, we do not have to write the overloading for each operator by hand. The code produced with the macro expansion is equivalent to the Fragment 2.4.

Source Code 2.4: Code with the macro expanded

```
1 Vec& Vec::operator+= (const double x) {
2   return apply([x](double y) { return x + y; });
3 }
4 Vec& Vec::operator-= (const double x) {
5   return apply([x](double y) { return x - y; });
6 }
7 Vec& Vec::operator*= (const double x) {
8   return apply([x](double y) { return x * y; });
```

[1] Taken from Stack Overflow website: `https://stackoverflow.com/questions/34090385/c-macro-metaprogramming`

```
9  }
10 Vec& Vec::operator/= (const double x) {
11   return apply([x](double y) { return x / y; });
12 }
```

The macro in this case does direct code replacement while using the argument provided, which means that there is no type-safety check and side effects are possible, cf. Code 2.5.

Source Code 2.5: Metaprogram based on macro with side effects

```
1 #define POWER(x) ((x) * (x))
2
3 int main() {
4   int val = 41.0;
5   int result = POWER(++val);
6 }
```

In this example, the macro acts like a basic power function. We are expecting in the variable `result` the value 1764 (because $42 * 42 = 1764$). The generated code is `result = (++val) * (++val)` and the result is 1936, because the variable `val` is incremented twice.

This shows with a simple example that macros could do some metaprogramming, as we have generated some code in our first example, but could be difficult to use especially due to the side effects and their limitation. However, the Boost library proposes a library using macros for complex metaprogramming operations on data structures notably, called Preprocessor [2].

2.3 Template metaprogramming

Template metaprogramming in C++ is based on templates that are the way generic programming [Musser and Stepanov, 1989] is offered in C++. Programs are defined with undefined parameter types (and/or values, mainly integers), and later at compile-time, from these generic definitions, concrete code will be built by revealing the unknown parameter values and types, in a process called instantiation. The original idea is to remove source code redundancy by defining generic function, class, or method that will be applicable to multiple types. In this section, we present the basic principles of generic programming with templates, and their specificity of instantiation and specialization needed for template metaprogramming. The specificities of templates, mainly the way they are instantiated (creating the concrete element from its generic form that is usually costless at runtime), and their ability to be specialized, allow template metaprogramming and make it relevant to use in some situations.

2.3.1 Fundamental template principles

In order to describe the template mechanism, we need to introduce how templates can be declared and used, and give some details on the fundamental instantiation and specialization principles [Vandevoorde et al., 2017].

[2] https://www.boost.org/doc/libs/1_72_0/libs/preprocessor/doc/index.html

In Code 2.6, we declare a template function called `equal` with one template parameter `T` that represents an unknown type at this point. At line 1, we declare the template parameter (there can be many, the list of parameters is delimited by $<$ and $>$), and at line 2, the function is defined the classical way, the only difference being the use of parameter `T` instead of an actual type. Thus, at this point, the type of arguments `lhs` and `rhs` is unknown.

Instantiation

The *instantiation* is the process of binding values to the parameters of a template in order to create a concrete entity; this entity could be a function, a class, a variable, or a type alias.

Code 2.6 presents an example of explicit and implicit instantiation. At lines 1-2, we define a method `equal` with a template parameter `T`. At line 8, we instantiate the `equal` function by binding the `int` type to parameter `T`. To get a concrete function from the template function, a "value" for the template parameter is provided in order for the compiler to produce a dedicated version of `equal`. This instantiation generates the `equal` function from the template `equal` function, with the `int` type replacing the unknown type `T`, so the compiler generates a function with the signature: `bool equal (int const & lhs, int const & rhs)`. At line 9, we instantiate the same way the `equal` function with the type `float`, which induces the creation of another concrete `equal` function with the signature: `bool equal (float const & lhs, float const & rhs)`. These instantiations are explicit, as we provide clearly the value for `T` when using the `equal` functions: `equal<int>` and `equal<float>`.

Source Code 2.6: Template definition and instantiation

```
template <class T>
bool equal(T const & lhs, T const & rhs) { return lhs == rhs; }

int main (){
  int a = 4, b = 2;
  float c = 42, d = 42;

  equal<int>(a, b);
  equal<float>(c, d);

  equal(a, b);
  equal(c, d);
}
```

However, it is possible to let the compiler deduce the type bound to parameter `T` as shown at lines 11-12, this is called implicit instantiation. The compiler knows the type of `a`, `b`, `c`, and `d`, and because arguments of type `T` are expected, it will deduce that `int` should be bound to parameter `T` at line 11 and `float` at line 12 to get appropriate instantiations of `equal`. This deduction is not always possible and follows some rules [3].

In the final code, the `equal` function exists in two flavors, one for each type, which results in a larger code base, so a larger executable. The counterpart of the increased size is a gain in performance because it exists as dedicated code of the `equal` function for each type bound to `T` (cf. Section 2.3.4).

[3] https://en.cppreference.com/w/cpp/language/template_argument_deduction

The same principle can apply for classes and variables; they can be parameterized with the same kind of construct to build generic classes (cf. lines 2-4 in Code 2.7) and variables (cf. lines 7-8 in Code 2.7).

Source Code 2.7: Class and variable templates

```
// Class template
template <class T> class GenericClass {
  T attribute_;
};

// Variable template, available since C++14
template<class T>
constexpr T pi = T(3.1415926535897932385L);
```

Specialization

The *specialization* of a template is the declaration or definition of a dedicated version of a template for specific values bound to at least one parameter of the template. If all template parameters are bound to a value in a specialization, we talk about a *full specialization*; alternatively, if at least one template parameter remains undefined, we talk about *partial specialization*.

Code 2.8 shows examples of specializations. First, the template structure `MyStruct` is defined at lines 2-4, we call it the "primary" template version. The structure `MyStruct` is defined with two template parameters `T` and `N`. At lines 6-8 and 10-12, we have two partial template instantiations, one binding `int` type to `T` and the other `float` type. In both cases, the value of template parameter `N` is still undefined. In the last specialization, at lines 15-17, both parameters `T` and `N` are bound to values, so it is a full specialization.

Source Code 2.8: Partial and full template specializations

```
// Primary template
template <class T, int N> struct MyStruct {
  MyStruct() { std::cout << "MyStruct<?,?>\n"; }
};
// #0: partial specialization where T is int
template <int N> struct MyStruct<int, N> {
  MyStruct() { std::cout << "MyStruct<int,?> init\n"; }
};
// #1: partial specialization where T is float
template <int N> struct MyStruct<float, N> {
  MyStruct() { std::cout << "MyStruct<float,?> init\n"; }
};

// #2: full specialization where T is float and N equals 1
template <> struct MyStruct<float, 1> {
  MyStruct() { std::cout << "MyStruct<float,1> init\n"; }
};

int main() {
  MyStruct<double, 10> class1; // Print "MyStruct<?,?> init"
  MyStruct<int, 10>    class2; // Print "MyStruct<int,?> init"
  MyStruct<float, 10>  class3; // Print "MyStruct<float,?> init"
  MyStruct<float, 1>   class4; // Print "MyStruct<float,1> init"
}
```

In Code 2.8, `N` is a non-type template parameter (NTTP). An NTTP however presents some limitations: it needs to have a *structural type* [4]. Until C++ 20, theses types were mostly reduced to integral types and pointer types. For example, a template parameter of integral type can be used to define at compile-time the size of a `std::array` as shown in code 2.9. In this example, we design a house full of humans, the number is statically set with the NTTP `NumberOfHumans` that is used to parameterize the `std::array` of `Human`.

Source Code 2.9: NTTP usage

```
#include <array>
#include <cstddef>

class Human {};

// Usage of NTTP
template <size_t NumberOfHumans> class House {
  std::array<Human, NumberOfHumans> humans_;
};
```

2.3.2 Variadic templates

A variadic template is a template that can accept variable/unbound template arguments as shown in Code 2.10. In this case, the variadic template function `sumOfSquares` is defined with a set of parameters (undefined types and quantity) identified as a unique identifier `Args` called a *template parameter pack* (cf. line 6). This pack represents here a set of undefined types and is used to declare the parameters of the function; it is also represented by a unique identifier `args` called a *function parameter pack*. The parameter pack consists of an undefined number of parameters, with unknown types (represented by the template parameter pack) introduced using an ellipsis (...). Contrary to classical variadic functions that use only an ellipsis without information on the types, this approach as a variadic template allows more type control at compile-time.

Source Code 2.10: Variadic template and pack expansion

```
// Generic square function
template<class Arg>
Arg square(Arg const & arg) { return arg * arg; }

// Variadic template function
template<class ...Args>
auto sumOfSquares(Args const & ... args) {
  // Right argument unfolding with a + operator
  return (square(args) + ...);
}

int main (){
  int a = 4;
  float b = 2;
  double c = 6;
  return sumOfSquares(a, b, c);
}
```

As a parameter pack is a list of parameters, it cannot be used as is, it must be "expanded", in order to get the list of the parameters used in a given context. Pack

[4] https://en.cppreference.com/w/cpp/language/template_parameters

expansion appears when at least a pack is used in an expression pattern followed by an ellipsis. Pack expansion can appear at many places, when building, for instance, a template argument list to instantiate another template, a function argument list, a series of operations on each element of a function parameter pack... More specifically, the last kind of pattern implying a binary operator is called *fold-expression*, and the expansion of fold-expressions follows some rules presented in Table 2.2 [Vandevoorde et al., 2017].

Fold Expression	**Evaluation**
(... op pack)	(((pack1 op pack2) op pack3) ... op packN)
(pack op ...)	(pack1 op (... (packN-1 op packN)))
(init op ... op pack)	(((init op pack1) op pack2) ... op packN)
(pack op ... op init)	(pack1 op (... (packN op init)))

Table 2.2: Fold expression evaluation

In Code 2.10, `a`, `b`, and `c` are `int`, `float`, and `double` values respectively. When calling the method `sumOfSquare(a, b, c)`, the compiler deduces these types and instantiates the function with those types, meaning the set of types <`int,float,double`> is bound to the parameter pack `Args` (line 6). Then, the argument list at line 7 is expanded into `int const & args1, float const & args2, float const & args3`. Finally, at line 9, we sum the square of each number of the function parameter pack `args`: the fold-expression `(square(args) + ...)` is expanded into `square(args1) + square(args2) + square(args3)`.

The template instantiation looks like the output presented in Source Code 2.11. It is a modified output (to ease understanding) from the source code compiled with C++ and some plugins (clang++ -std=c++17 -Xclang -ast-print -fsyntax-only). We can see in this output the instantiations of the square function for each of the types (`int`, `float`, and `double`), and also the instantiation of `sumOfSquare` function with a pack.

Source Code 2.11: Output example after template instantiation

```
1  template <class Arg> Arg square(const Arg &arg) {
2    return arg * arg;
3  }
4  template<> double square<double>(const double &arg) {
5    return arg * arg;
6  }
7  template<> float square<float>(const float &arg) {
8    return arg * arg;
9  }
10 template<> int square<int>(const int &arg) {
11   return arg * arg;
12 }
13 template <class ...Args> auto sumOfSquare(const Args &...args) {
14   return (square(args) + ...);
15 }
16 template<> double sumOfSquare<<int, float, double>>(const int
   ↪  &args1, const float &args2, const double &args3) {
17   return square(args1) + square(args2) + square(args3);
18 }
```

Elements in a template parameter pack can be manipulated without the expansion

mechanism. It is possible to extract recursively the types from a template parameter pack as shown in Code 2.12.

Source Code 2.12: Recursive manipulation of pack elements

```
1  #include <iostream>
2
3  template <class Front, class ...Types>
4  void foo(Front front, Types... remaining) {
5    std::cout << __PRETTY_FUNCTION__ << " front: " << front << "\n";
6    if constexpr (sizeof...(Types) > 0) foo<Types...>(remaining...);
7  }
8
9  int main () {
10   foo<int, float, double>(1, 2., 42.);
11 }
12
13 // Execution output
14 // void foo(Front, Types...) [Front = int, Types = <float, double>]
   ↪  front: 1
15 // void foo(Front, Types...) [Front = float, Types = <double>]
   ↪  front: 2
16 // void foo(Front, Types...) [Front = double, Types = <>] front: 42
```

If one or more arguments are passed to `foo`, identifying the first argument separately from the remaining ones allows doing a computation on this first one before calling recursively the same function on the remaining ones. The operator `sizeof...` allows to get the size of the template parameter pack and to stop the computation when no type in `Types...` and no argument in `remaining` remain. The output of the execution shows how the function is recursively called and how the types are extracted from the parameter pack. Note here that this code is only valid if at least one argument is given to the function, otherwise the function is not defined, because `front` is not provided.

2.3.3 Type deduction

The C++ language offers two keywords, `auto` and `decltype`, to explicitly use its type deduction mechanisms. They are useful in template metaprogramming, first to avoid writing explicitly complex type expressions, but more important, to allow powerful expressiveness. For instance, without such feature, writing a simple generic addition function is tedious (if not impossible): how to express the type that `auto` represents at line 24 in Code 2.13 ?

Source Code 2.13: Difference of deduction between `auto` and `decltype`

```
1  #include <string_view>
2  #include <iostream>
3
4  // For illustration purpose, works only with GCC
5  template <typename T> constexpr auto type_name() noexcept {
6    std::string_view name = __PRETTY_FUNCTION__;
7    std::string_view prefix = "constexpr auto type_name() [with T = ";
8    std::string_view suffix = "]";
9    name.remove_prefix(prefix.size());
10   name.remove_suffix(suffix.size());
11   return name;
12 }
13
14 // Class declaration for testing
```

```
15 class A {
16   int i_;
17  public:
18   A(int const i) : i_(i) {}
19   int const & i() { return i_; }
20 };
21
22 // Generic addition function
23 template <class T, class U>
24 auto add(T const & lhs, U const & rhs) { return lhs + rhs; }
25
26 // Generic multiplication function
27 auto mul(auto const & lhs, auto const & rhs){ return lhs + rhs; }
28
29 int main(void) {
30   A a(12);
31
32   auto i = a.i();
33   decltype(a.i()) j = a.i();
34
35   std::cout << type_name<decltype(i)>(); // => int
36   std::cout << type_name<decltype(j)>(); // => const int&
37
38   std::cout << type_name<decltype(add(i, 12.5))>(); // => double
39   std::cout << type_name<decltype(mul(i, 12.5))>(); // => double
40 }
```

We will not describe all the differences between these two deduction features, but we illustrate some of them with Code 2.13. In this code, we use a template function (lines 5-12, only for GCC) to get precisely what is the type bound to its template parameter `T`. Class `A` is used to test which return type is deduced by `auto` and `decltype` for the accessor `i()` at lines 32 and 33.

The first keyword, `auto` [Järvi et al., 2006], is used as a placeholder for a type that will automatically be deduced by the compiler based on the expression in which `auto` is used. The deduction follows the same logic as template type deduction [5]. For instance, expression at line 32 will result in the deduction that `auto` is of type `int`. Note that here, the deduced type comes with no type qualifier (e.g., `const`) and not as a reference (e.g., one could have expected here `const int &`). To get a reference for instance, one would write: `const auto & i = a.i()`.

`decltype` [Niebler et al., 2011] is a specifier that deduces the type of an expression. For instance, expression at line 33 in Code 2.13 will deduce type `const int &`. Note that the expression is analyzed to find out its type, but it will not lead to runtime code (no binary code is generated).

Keyword `auto` is also used in two functions in our code. First, at lines 23-24, `add` function uses `auto` as a placeholder for the return type of the function. Second, at line 27, the `mul` function also uses `auto` as placeholders for the types of its two arguments, illustrating that type deduction with `auto` behaves as with template parameters.

2.3.4 C++ templates vs. Java generics

One thing that makes template metaprogramming interesting, and that is one specificity of C++, is its almost no overhead at runtime, which is due to the instantiation mechanism of C++. Indeed, some languages like Java use another mechanism for instantiation that is based on *type erasure*, that may imply runtime overhead.

[5] https://en.cppreference.com/w/cpp/language/template_argument_deduction

Java 5 brought generic programming [Parnin et al., 2013] with a construct that allows a class, an interface or a method to be parameterized with unknown types, in a way similar to C++. Instantiating such a generic element is apparently very similar to C++, but the instantiation process actually generates very different code as explained in this section.

Class-based type parameters

First, we need to recall that every object in Java inherits from the `Object` class, which means there exists a valid cast from any class to `Object`. Thus, one way to abstract any class-based type is by using the base class `Object`. This is illustrated in Code 2.14 at lines 3-5, where method `foo` receives any possible object as argument. This way of abstracting class-based types in Java is used in generics: when a generic parameter is declared, it represents a class-based type, which means that non class-based types cannot be bound directly to a parameter type.

By contrast, in C++, any type (either class-based or not) can be abstracted by a template parameter. If we consider for instance the `ArrayList<T>` generic standard class of Java, instantiating `ArrayList<int>` to get a resizable array of integers is not possible directly. One needs to instantiate `ArrayList<Integer>`, where `Integer` is a class that represents "integer" objects (i.e., objects wrapping an integer value). Storing integers with this generic class leads to an array of integer objects that is an inefficient data structure compared to an array of integer values that would have been produced with the equivalent template in C++.

The transformation of an integer value (of a non class-based type) into an integer object is called "boxing", and is implicitly performed every time an object is expected and a non-class based value is provided (cf. line 13 of Code 2.14). The opposite operation (cf. line 16 of Code 2.14) is called "unboxing".

Source Code 2.14: Boxing in Java

```
1  public class ObjectTest {
2    // Method accepting any object
3    public static void foo(Object obj) {
4      System.out.println("obj = " + obj);
5    }
6
7    // Method accepting any object and returning it back
8    public static Object bar(Object obj) {
9      return obj;
10   }
11
12   public static void main(String []args){
13     foo(1); // Calls "foo" with an integer object (auto boxing)
14     foo("blabliblou"); // Call "foo" with a string object
15
16     int i = (Integer)bar(5);
17     // Call "bar" with an integer object (auto boxing)
18     // Mandatory downcast from "Object" to "Integer"
19     // And ultimately, auto unboxing to get integer value
20
21     System.out.println("i = " + i);
22   }
23 }
```

Type erasure

In Code 2.15, we introduce a generic class `Box` with a parameter type `T`, whose purpose is to wrap an object of type `T` inside an object `Box<T>`. Again, `T` can only be a class-based type, so many auto boxing and unboxing are induced in the code (lines 12, 15, and 18).

Source Code 2.15: Generics in Java

```
public class Box<T> {
  private T value;

  public Box(T v) { value = v; }

  public T getValue() { return value; }
  public void setValue(T v) { value = v; }

  @Override public String toString() { return value.toString(); }

  public static void main(String[] args) {
    Box<Integer> i = new Box<Integer>(5); // explicit + auto boxing
    Box<String>  s = new Box<String>("Hello !");

    int    j  = i.getValue(); // auto unboxing
    String t = s.getValue();

    i.setValue(2); // auto boxing
    s.setValue("Bonjour !");

    System.out.println("i = "+i);  System.out.println("s = "+s);
    System.out.println("j = "+j);  System.out.println("t = "+t);
  }
}
```

In this example, the generic class `Box` is instantiated with `T = Integer` and `T = String` at lines 12 and 13. This induces the generation of only one class in Java. What happens is called *type erasure*. Each generic parameter is replaced by class `Object` (or its bound class [Bracha, 2004]) in all the code of generic class `Box` to produce a concrete class `Box`, as shown by Code 2.16 that is the Java code resulting from the decompilation of the bytecode of Code 2.15. We can notice that the instantiation process also adds automatically type casts and generates bridges to guarantee type safety and preserve polymorphism in extended generic types [Ghosh, 2004]. We can also see that the final bytecode has no trace of generic parameters.

Source Code 2.16: Code produced by the instantiation process of Java

```
// Decompiled by Jad v1.5.8g.

import java.io.PrintStream;

public class Box{
  public Box(Object obj){ value = obj; }

  public Object getValue(){ return value; }
  public void setValue(Object obj){ value = obj; }
  public String toString(){ return value.toString(); }

  public static void main(String args[]){
    Box box = new Box(Integer.valueOf(5));
    Box box1 = new Box("Hello !");
    int i = ((Integer)box.getValue()).intValue();
    String s = (String)box1.getValue();
```

```
17      box.setValue(Integer.valueOf(2));
18      box1.setValue("Bonjour !");
19      System.out.println(
20      (new StringBuilder()).append("i = ").append(box).toString());
21      System.out.println(
22      (new StringBuilder()).append("s = ").append(box1).toString());
23      System.out.println(
24      (new StringBuilder()).append("j = ").append(i).toString());
25      System.out.println(
26      (new StringBuilder()).append("t = ").append(s).toString());
27    }
28    private Object value;
29  }
```

This type erasure mechanism, which produces additional code like casts, combined with the implicit boxing and unboxing, could lead to particularly inefficient code if not carefully crafted. Similar code in C++ would produce, for each template instantiation, a dedicated code that is somehow equivalent to the handwritten code issued by copying the generic code and replacing each occurrence of the template parameters by their value. It means that dedicated code is generated at compile-time for each different set of values bound to the parameters of a template, avoiding thus any overhead due to genericity, but creating larger executable as many instances of a template can possibly be produced.

2.3.5 Metafunction

Metafunctions in C++ are not part of the standard or directly supported by the language, but are a community-wide defined construct. A metafunction acts as a function at compile-time that takes as arguments static data (mainly types or numbers), and possibly returns static data. Metafunctions can be used to perform actual computation (cf. `factorial` in Section 2.3.9), to generate code (cf. `power` in Section 2.3.9), or to manipulate types for checking (cf. `is_same` in this section) or to produce types (cf. `remove_reference` in this section).

While there is the term "function" in their name, metafunctions are not actual functions but are built as template classes with a specific structure: the template parameters of the class represent the parameters of the metafunction and the return value(s) is/are member(s) exposed by the class.

Traditional functions can have zero to many parameters and can return a value or none, as shown at lines 1-3 in code 2.17 with a function `add` that takes two parameters `x` and `y` and returns the addition `x+y`. Such a function is designed to compute the addition at runtime (line 5).

Source Code 2.17: Example of addition function and metafunction

```
1  int add(int x, int y) {
2    return x + y;
3  }
4
5  assert(42 == add(13,29));
6
7  template <int X, int Y> struct Add {
8    static constexpr int value = X + Y;
9  }
10
11 static_assert(42 == Add<13,29>::value);
```

A similar metafunction `Add` can be designed to statically perform the addition. It can be represented by a class `Add` with two template parameters `X` and `Y`, and the return value is the class member `value` assigned with the result of the addition `X + Y` (lines 7-9). Such a function can be used at compile-time, as shown at line 11 with an assertion checked during compilation.

Type metafunctions are metafunctions that expose a type and *value metafunctions* are metafunctions that expose a value. By convention, a type metafunction returns a type by having an attribute *type*, and a value metafunction returns a value by having an attribute *value*. This allows the composition of metafunctions as we see in Code 2.19. To ease metafunction definitions, static integral numbers can notably be represented by the standard `integral_constant` template class, whose simplified definition is presented at lines 2-6 of Code 2.18. Among others, this class has the attributes `value` and `type`. Using this template, static integers 2 and 4 can be defined at lines 10 and 11.

Source Code 2.18: Example of value metafunction (`integral_constant`)

```
// Simplified definition of "integral_constant"
template<class T, T V> struct integral_constant {
  static constexpr T value = V;
  using value_type = T;
  using type = integral_constant<T,V>;
};

int main() {
  using two_t  = integral_constant<int,2>;
  using four_t = integral_constant<int,4>;

  static_assert(two_t::value*2 == four_t::value);
}
```

In Code 2.19, we first present at lines 2-4 the type metafunction `remove_reference` with one parameter `T`. Its goal is to return the actual type behind type `T` that can possibly be a reference. The primary version of the template (line 2) returns `T` by default, and specializations of the template (lines 3 and 4) for patterns of `T` that match `T = U&` and `T = U&&` both return `U` (lines 3 and 4).

Source Code 2.19: Example of type metafunction composition

```
// Metafunction "remove_reference"
template<class T> struct remove_reference      { using type = T; };
template<class U> struct remove_reference<U&>  { using type = U; };
template<class U> struct remove_reference<U&&> { using type = U; };

// Definition of "true_type" and "false_type"
using true_type = integral_constant<bool, true>;
using false_type = integral_constant<bool, false>;

// Metafunction "is_same" to test type equality
template<class T, class U> struct is_same : false_type {};
template<class T> struct is_same<T, T> : true_type {};

// Metafunction "is_same_v2" to test types without references
template<class T, class U> struct is_same_v2
  : is_same< typename remove_reference<T>::type,
      typename remove_reference<U>::type > {};

int main () {
  static_assert(is_same<int,int>::value);
  static_assert(!is_same<int,int&>::value);
```

```
22   static_assert(!is_same<int,int&&>::value);
23
24   static_assert(is_same_v2(int,int)::value);
25   static_assert(is_same_v2(int,int&)::value);
26   static_assert(is_same_v2(int,int&&)::value);
27 }
```

From the previous definition of `integral_constant`, `true_type` and `false_type` constants are built (lines 7-8), and we can define a metafunction `is_same<T,U>` to compare two types (lines 11 and 12). If the two types are the same, the specialization of `is_same` inheriting from `true_type` is selected (line 12); if the two types are different, the primary definition is selected inheriting from `false_type` (line 11). Notice that inheritance is used as the "return" construct in metafunctions, as the public members of the super-class will be accessible in the class representing the metafunction.

Metafunctions can be composed. The comparison of types is very strict with `is_same`, for instance, types `int` and `int&` are considered different. To loosen the test, one can first apply `remove_reference` on both `T` and `U` before comparison, as shown in metafunction `is_same_v2` (lines 15-17).

2.3.6 SFINAE

SFINAE for "Substitution Failure Is Not An Error", is a rule used in template metaprogramming. The rule says: "When substituting the explicitly specified or deduced type for the template parameter fails, the specialization is discarded from the overload set instead of causing a compile error." [Järvi et al., 2003].

Code 2.20 presents an example of a construct that exploits the SFINAE rule. The main idea is to test if a class has a `foo` method defined. At lines 6 and 7, we define two types, `yes` and `no`, that have different sizes, 1 and 2 respectively. At lines 9 and 11, we define two times the template method `fooExist` with parameter `U`. The first one returns `yes` if it possible to get the address of a method called `foo` from the type `U` (if method `U::foo` does not exist, the SFINAE applies); the second one, the fallback, can possibly be called in every situation, as it is a variadic function, and returns `no`. The size of the return of the call to `fooExist<T>` at lines 14-15 is then compared with the size of the `yes` type, and the result is stored into the exposed return of metafunction `testFoo` called "value".

Source Code 2.20: Usage of SFINAE to test if a member function exists

```
1  struct A { void foo(){} };
2  struct B {};
3
4  template <class T> class testFoo {
5   private:
6    using yes = char[1];
7    using no = char[2];
8
9    template <class U> static yes& fooExist(decltype(&U::foo));
10
11   template <class U> static no& fooExist(...);
12
13  public:
14   static const bool value =
15     sizeof(fooExist<T>(nullptr)) == sizeof(yes);
16 };
17
```

```
18 int main () {
19   static_assert(testFoo<A>::value);
20   static_assert(!testFoo<B>::value);
21 }
```

When instantiated with the type `A` (that has a method `foo`) at line 19, or with the type `B` (that has no method `foo`) at line 20, metafunction `testFoo` returns a value that implies the test of lines 14-15 with two possible overloads (line 9 and 11). When considering the first one, line 9, the compiler tries to deduce the type of the address of a method `foo` belonging to the type `A` or `B`. As `foo` exists for the type `A`, the address `&A::foo` exists and its type can be deduced with `decltype`, but for type `B`, method `foo` does not exist, which induces a substitution failure and ultimately the SFINAE applies. The second overload, line 11, is always valid as it is a variadic function. The compiler selects the more specific overload, which is the one returning `yes` for the instantiation `testFoo<A>`, and has no choice but to select the fallback `fooExist` that returns `no` for the instantiation `testFoo<B>`.

A common idiom exploiting the SFINAE rule is based on the standard `enable_if<B,T>` metafunction presented in Code 2.21 (lines 2-6). The principle of this metafunction is to expose a member `type` (equal to `T`) when the boolean parameter `B` is true (lines 5-6), and no member otherwise (lines 2-3). This way, using `enable_if` in the substitution of template parameters can activate the definition of a method depending on a test (similarly to the previous example).

For example, Code 2.21 defines two template functions `printInteger<T>` and `printDouble<T>`. The former function will be defined only if the type bound to parameter `T` is `int`, `int&`, or `int&&`, and the latter function will be defined only if the type bound to `T` is `double`, `double&`, or `double&&`. Lines 29 and 30 provide valid code, whereas lines 33 and 34 lead to compilation errors.

Source Code 2.21: `enable_if` metafunction

```
1  // Metafunction "enable_if"
2  template<bool B, class T = void>
3  struct enable_if {};
4
5  template<class T>
6  struct enable_if<true, T> { using type = T; };
7
8  // Function "printInteger"
9  template <class T,
10   class = typename enable_if<
11     is_same<int, typename remove_reference<T>::type>::value
12   >::type
13 >
14 void printInteger(T t) {
15   std::cout << "Integer: " << t << std::endl;
16 }
17
18 // Function "printDouble"
19 template <class T,
20   class = typename enable_if<
21     is_same<double, typename remove_reference<T>::type>::value
22   >::type
23 >
24 void printDouble(T t) {
25   std::cout << "Double: " << t << std::endl;
26 }
27
28 int main() {
29   printInteger(1);  // print "Integer: 1"
```

```
30    printDouble(1.1); // print "Double: 1.1"
31
32    // Compilation errors
33    // printInteger(1.1); // no match for "printInteger(double)"
34    // printDouble(1);    // no match for "printDouble(int)"
35    return 0;
36  }
```

2.3.7 Traits and helpers

Because template metaprogramming is mainly about type computation, there is a templatized interface to interact with type properties called *type traits* [Myers, 1996, Austern, 2005]. Their purpose is to give information and apply transformations on unknown types. They include type classification traits, type property inspection traits, and type transformations. The first ones propose a taxonomy of types and situate a type in this taxonomy; the second ones inspect types for their important characteristics; the third ones propose a type manipulation interface.

Standard type traits satisfy some requirements that have been organized in *named requirements* that should be formalized in C++ 20 with concepts (cf. Section 2.3.8). Mainly, three categories have been defined. A *UnaryTypeTrait* is a class template that takes one template argument and optional additional argument(s) to get the property of the template argument (e.g., `is_array` template). A *BinaryTypeTrait* is a class template that describes the relation between two types with optional additional argument(s) (e.g., `is_same` template). A *TransformationTrait* is a class template that takes one or more template arguments defining a transformation on the argument(s) (e.g., `remove_reference` template).

Traits are mainly metafunctions. However, some of them like `std::is_pod` (deprecated in C++ 20) or `std::is_union` are intrinsic, and they need assistance from the compiler.

A full list of type traits is available online [6]. Here is a list of the ones we have used:

- `is_constructible<T>`: Tests if a class `T` has a constructor with defined parameters,
- `is_default_constructible<T>`: Tests if a class `T` has a default constructor,
- `is_base_of<A,B>`: Tests if a class `A` inherits from a class `B`, or if `A` and `B` are the same class,
- `is_same<A,B>`: Tests if two types `A` and `B` are the same,
- `disjunction<B...>`: Logical OR operation to chain the traits in pack `B`.

Helper templates provide a generalized and easier way to access the result value or type of a trait or metafunction. The name of a standard helper has a suffix `_v` or `_t` to indicate the access to the return value or type respectively of the metafunction. The helper types were introduced first in C++ 14; for instance, in Code 2.21, to access the `type` attribute of `enable_if` metafunction, one need to use the `typename` keyword (lines 11 and 21). With the helper type `enable_if_t` (lines 2-3 of Code 2.22), the definition of function `printInteger` of Code 2.21 can be simplified (line 11 of

[6] `https://en.cppreference.com/w/cpp/header/type_traits`

Code 2.22). Similarly, the helper variable templates are a construct popularized in C++ 17; for instance, the helper variable `is_same_v` can be defined as at lines 5-6.

Source Code 2.22: Helper type and variable templates

```
// Helper type template
template<bool B, class T = void>
using enable_if_t = typename enable_if<B,T>::type;

// Helper variable template
template<class T, class U>
constexpr auto is_same_v = is_same<T,U>::value;

// Function "printInteger"
template <class T,
  class = enable_if_t< is_same_v< int, remove_reference_t<T> > >
>
void printInteger(T t) {
  std::cout << "Integer: " << t << std::endl;
}
```

2.3.8 Constraints and concepts

The *constraints* and *concepts* are an important addition in the C++ 20 norm [Stroustrup, 1997]. A first attempt to introduce concepts in the language has been studied [Gregor et al., 2007], but was abandoned for the benefit of a simpler approach [Sutton et al., 2013, Sutton, 2015]. Library-based implementations have been proposed, i.e. a Boost library for concepts checking [Siek and Lumsdaine, 2000], or a library for concept-based specialization of templates [Bachelet and Yon, 2017]. Ultimately, language-level implementations have been proposed (e.g. Clang compiler [Voufo et al., 2011]).

Constraints and *concepts* first purpose is to allow specifying constraints on unknown types (mainly template parameters and automatically deduced types like with the placeholder `auto`), either directly by a sequence of requirements (called *constraint*) or a named set of requirements (called *concept*). The second objective of constraints and concepts is to specify requirements on template parameters that will be used in the selection of the most appropriate template specialization or function overload, replacing the `enable_if` technique.

Constraints

A *constraint* is a sequence of logical operations and operands that specifies requirements on one or several template parameters. A constraint can be used directly on a class template, function template, or a member of a class template using the `requires` keyword to force template parameters to be bound to types that satisfy the constraint. As explained later, a constraint can also be used to define a concept. There are exists three types of constraints: conjunction of constraints, disjunction of constraints, and atomic constraint.

For example, Code 2.23 defines a function template `add<T,U>` to add two numbers of types `T` and `U` respectively. We want the use of this function to be limited to integer and floating point types for `T` and `U`.

Source Code 2.23: Conjunction and disjunction of constraints

```
#include <concepts>

class House {};

template <class T, class U>
requires (std::integral<T> || std::floating_point<T>) &&
↪  (std::integral<U> || std::floating_point<U>)
  auto add(T const & lhs, U const & rhs) { return lhs + rhs; }

int main () {
  int i = 1;
  float j = 2;
  add(i, j); // Compilation successful

  House h;
  add(i,h); // Compilation fails => constraint not satisfied for U
}
```

Such a constraint can be expressed as: *(T is integral OR T is floating point) AND (U is integral OR U is floating point)*. The logical *AND* defines a conjunction constraint (operator `&&`) and the logical *OR* defines a disjunction constraint (operator `||`). This constraint is imposed on the `add` function with the `requires` statement at line 6.

The `add` function is then instantiated at line 12 with `i` and `j` as arguments, because `T` and `U` are bound to `int` and `float` respectively that satisfy fully the constraint. The function is not instantiated at line 15 with `i` and `h` as arguments, because `U` is bound to `House` that is neither an integral nor a floating point, which fails satisfying the constraint.

Concepts

The standard `integral<T>` and `floating_point<T>` are concepts, and more precisely named atomic constraints from the standard traits `is_integral` and `is_floating_point`, as shown in Code 2.24 at line 3. An atomic constraint can be any expression on template parameters that can be statically evaluated as `true` and cannot be decomposed into a conjunction or a disjunction (if the expression is not valid, it is considered as evaluating to `false`). An atomic constraint can also be a *requires expression* (explained below); for instance in Code 2.24 at line 7, the constraint is a requirement that expression `a+b` is valid for variables `a` and `b` of type `T` (the expression is not evaluated, only language correctness is checked). Atomic constraints can be used directly as template constraints, see at lines 11 and 16.

Source Code 2.24: Concept definitions with a atomic constraint.

```
// Concept definition from a type trait
template <class T>
concept integral = std::is_integral<T>::value;

// Concept definition from a requires expression
template <class T>
concept Addable = requires (T a, T b) { a+b; };

// Atomic constraint from a type trait
template <class T>
requires (std::is_integral<T>::value)
auto f1(T const & lhs,T const & rhs) { return lhs+rhs; }

// Atomic constraint from a requires expression
template <class T>
```

```
16  requires (requires (T a, T b) { a+b; })
17  auto f2(T const & lhs,T const & rhs) { return lhs+rhs; }
```

It is possible to bundle constraints in a named *concept* to define a class of types. As such, we can think of functions or classes as accepting some type to bound a template parameter only if it satisfies a concept (i.e., satisfies all its requirements) as shown in Code 2.25: at line 3 with a requires statement and at line 7 directly in the declaration of a template parameter. It is also possible to use a concept to constrain a placeholder declaration with `auto` (line 11).

Source Code 2.25: Different usages of concepts

```
1   // Concept in requires statement
2   template <class T>
3   requires Addable<T>
4   auto f3(T const & lhs,T const & rhs) { return lhs+rhs; }
5
6   // Concept in template parameter declaration
7   template <Addable T>
8   auto f4(T const & lhs,T const & rhs) { return lhs+rhs; }
9
10  // Concept in placeholder declaration
11  auto f4(Addable auto const & lhs,Addable auto const & rhs)
12  { return lhs+rhs; }
```

To define requirements for one or more types, different kinds of *requires expressions* are available[7]. A requires expression follows the syntax *requires {requirement-sequence}* or *requires (parameter-list) {requirement-sequence}*. A requirement can be either a *simple* requirement, a *type* requirement, a *compound* requirement, or a *nested* requirement as shown in Code 2.26.

- A simple requirement asserts that an unevaluated expression statement is valid, only based on checking language correctness (line 4).
- A type requirement consists of the keyword *typename* followed by a type name. It can be used for testing (1) the existence of a nested type (line 7) or (2) that a template specialization names a type (line 8: the instantiation is not prevented by a constraint).
- A compound requirement has the form *{expression} [noexcept] [-> type constraint]* . The `noexcept` and the type constraint are optional. The requirement tests the validity of the expression: if it is throwing an exception or not, and the constraint on the return type of the expression is satisfied (line 11).
- Finally, a nested requirement is another *requires* statement that can be used on local parameters (like variable `a` at line 17).

Source Code 2.26: Types of requirements

```
1   template<typename T>
2   concept RequirementExample = requires (T a, T b) {
3     // Simple requirement
4     a + b; // Expression a+b is a valid expression that will compile
```

[7] https://en.cppreference.com/w/cpp/language/constraints

```
5
6      // Type requirements
7      typename T::inner; // Type T has a nested type named "inner"
8      typename A<T>; // Type A<T> is a valid instantiation
9
10     // Compound requirement
11     {a * 1} -> std::convertible_to<T>;
12       // Expression a*1 must be valid
13       // And result must be convertible to T
14       // (i.e. must satisfy concept convertible_to<decltype(a*1),T>)
15
16     // Nested requirement
17     requires std::is_same<T*, decltype(&a)>::value;
18   };
```

Replacing `enable_if`

As an example (Code 2.27), we design a concept that defines what a cat is. After a long observation of test subjects, in its natural environment, a cat is default constructible, it meows, purrs, and sleeps. This is defined on lines 3 to 9 by concept `CatConcept<T>`. We use a standard trait to require the type `T` to be default constructible and add *simple* requirements to type `T`. Then, we define two types `DefinitelyACat` (lines 11-15) and `NotACat` (lines 17-21) that respectively meet and not meet the concept requirements. Finally, we define a function `doCatThings` that will accept anything that satisfies the concept of cat defined earlier. When we call `doCatThings` with an instance of `DefinitelyACat` the code compiles. The code does not compile with an instance of `NotACat`.

To show the improvement from traditional template metaprogramming techniques enabled by the use of concepts, we propose Code 2.28 that is equivalent to Code 2.27 to model the same concept of cat, using the SFINAE technique both to detect the existence of methods (like Code 2.20) and to constrain the instantiation of function template `doCatThings` using metafunction `enable_if` (like Code 2.21).

Source Code 2.27: Concept example

```
1   #include <type_traits>
2
3   template <class T>
4   concept CatConcept = requires(T aCat) { // Define what a cat is
5     requires std::is_default_constructible_v<T>;
6     aCat.meow();
7     aCat.purr();
8     aCat.sleep();
9   };
10
11  struct DefinitelyACat {
12    void meow(){};
13    void purr(){};
14    void sleep(){};
15  };
16
17  struct NotACat {
18    void fly(){};
19    void peck(){};
20    void sleep(){};
21  };
22
23  void doCatThings(CatConcept auto & cat) {
24    cat.meow();
25    cat.purr();
```

```
}

int main() {
  DefinitelyACat aCat;
  doCatThings(aCat);

  // NotACat whatCouldItBe;
  // doCatThings(whatCouldItBe); // Compilation failure
}
```

Source Code 2.28: SFINAE alternative to concept

```
template <class T> class IsCatSFINAE {
 private:
  using yes = char[1];
  using no = char[2];

  template <class U>
  static yes& testCat(decltype(&U::meow), decltype(&U::purr),
   ↪  decltype(&U::sleep));

  template <class U> static no& testCat(...);

 public:
  static const bool value =
    std::is_default_constructible_v<T> &&
    (sizeof(testCat<T>(nullptr, nullptr, nullptr)) == sizeof(yes));
};

template <class T,
  class = std::enable_if_t<IsCatSFINAE<T>::value>
>
void doCatThings(T & cat) {
  cat.meow();
  cat.purr();
}
```

2.3.9 Computation with template metaprogramming

C++ templates allow computations on types and to a certain extent on values using template metaprogramming techniques. The first use of such techniques has been described in [Unruh, 1994]. The template system of C++ is Turing complete [Veldhuizen, 2003]. As such, it is possible to theoretically solve any computational problem with template metaprogramming that is doable by a Turing machine, without any guarantees about runtime or memory footprints at compilation. More practically it is possible to compute simple values at compile-time or to generate a more specialized code, as we will show in the following examples.

Code 2.29 presents the well-known example of factorial computation with template metaprogramming.

Source Code 2.29: Factorial computation with template metaprogramming

```
template <int N>
struct Factorial {
  static const unsigned long long value = N * Factorial<N-1>::value;
};

template <>
struct Factorial<0> {
  static const unsigned long long value = 1;
};
```

```
10
11 int main() {
12   auto facto = Factorial<20>::value;
13 }
```

Metafunction `Factorial<N>` is first defined with a primary template that expresses the recursive computation $N! = N \times (N-1)!$ (lines 1-4). The result of the computation is stored in attribute `value`. The terminal case of recursion, here when $N = 0$, is expressed as a specialization of `Factorial` (lines 6-9). At compile-time, the `Factorial` template is recursively instantiated (line 12, starting with $N = 20$), and the calculus $20 \times 19 ... \times 1$ is progressively built and is computed by the compiler (as all values are known), and the final value is directly set in variable `facto` as shown in the assembly Code 2.30 resulting from the compilation of Code 2.29: the value 2432902008176640000 (20!) is stored on register `rax`.

Source Code 2.30: Assembly code of compile-time factorial

```
1 main:
2     push   rbp
3     mov    rbp, rsp
4     movabs  rax, 2432902008176640000
5     mov    QWORD PTR [rbp-8], rax
6     mov    eax, 0
7     pop    rbp
8     ret
```

To illustrate template metaprogramming for code generation, we present an example in Code 2.31 of three different implementations of the power function: `powerDyn(v,n)` that computes v^n, with `v` and `n` being dynamic values; `power<N>(v)` that computes v^N, with `v` being a dynamic value and `N` a static one; `power4(v)` that computes v^4, with `v` being a dynamic value. We study how GCC and Clang compile these implementations with different variations: (1) if the functions are inline or not (i.e., the keyword `inline` precedes the function definition or not), (2) the nature of the arguments passed to the functions (dynamic values - coming from arguments - or static values - literals), and (3) the compiler optimization flag (*-O0* - almost no optimization, or *-O1*/*-O2* - first levels of optimization for code size and execution time).

Source Code 2.31: Different power function implementations

```
1  // Function "power" with a partial static evaluation
2  template <unsigned N>
3  inline double power(double v) { return v*power<N-1>(v); }
4
5  template <>
6  inline double power<0>(double) { return 1; }
7
8  // Function "powerDyn" with a fully dynamic evaluation
9  inline double powerDyn(double v,unsigned p) {
10   double r = 1; while (p-- > 0) r*=v; return r;
11 }
12
13 // Function "power4" that is dedicated to n = 4
14 inline double power4(double v) { return v*v*v*v; }
15
16 // Test with "v" and "n" unknown at compile-time
17 double test_1(double v, unsigned n) { return powerDyn(v,n); }
18
```

```
19 // Tests with "v" unknown and "n" known at compile-time
20 double test_2(double v) { return powerDyn(v,4); }
21 double test_3(double v) { return power<4>(v); }
22 double test_4(double v) { return power4(v); }
23
24 // Tests with "v" and "n" known at compile-time
25 double test_5() { return powerDyn(12,4); }
26 double test_6() { return power<4>(12); }
27 double test_7() { return power4(12); }
```

We have observed different behaviors: (1) the function is created (symbol and body are present in assembly code) (F); (2) the function is invoked (a call to the function is present in assembly code) (C); (3) the function body is moved to the test function (the function is actually inlined and its code is present in the assembly code of the test function) (I); (4) the loop is unrolled (U); (4) the result value is computed at compile-time (the result value is present in the assembly code and directly used) (V). These behaviors are recapitulated in Table 2.3.

Table 2.3: Evaluation of the power function implementations for Clang 12.0.1 and GCC 11.1 (cf. Code 2.31)

	Power				
	No optimisation	**GCC Optimized -O1**		**Clang Optimized -O2**	
		Inlined	**Not Inlined**	**Inlined**	**Not Inlined**
test_1	F & C	I	F & I	I	F & I
test_2	F & C	I & U	F & I & U	I & U	F & I & U
test_3	F & C	I & U	F & I & U	I & U	F & I & U
test_4	F & C	I	F & I	I	F & I
test_5	F & C	V	F & V	V	F & V
test_6	F & C	V	F & V	V	F & V
test_7	F & C	V	F & V	V	F & V

F = [F]unction created / C = function is [C]alled / I = effective [I]nline / V = [V]alue computed at compile-time / U = loop [U]nrolled

We notice that when optimization is activated, the compilers decide to both inline and execute at compile-time, when possible, the code of function `powerDyn`, which leads to code similar to the one produced with template metaprogramming. Thus, we propose a second test that, in a similar way, implements the calculus of sine based on Taylor series. We recall that $sin(x)$ can be approximated by $\sum_{k=0}^{K} -1^k \frac{x^{2k+1}}{(2k+1)!}$, which can be implemented with template metaprogramming as in Code 2.32.

Source Code 2.32: Different sine function implementations

```
1  // Function "sine" with a partial static evaluation
2  template <int K>
3  inline double sine(double x) {
4    return power<K>(-1) * power<2*K+1>(x) / Factorial<2*K+1>::value
5      + sine<K-1>(x);
6  }
7
8  template <>
9  inline double sine<0>(double x) { return x; }
10
11 // Function "sineDyn" with a fully dynamic evaluation
```

```
12 inline double sineDyn(double x,unsigned k) {
13   double r = x;
14
15   while (k>0) {
16     r += powerDyn(-1,k) * powerDyn(x,2*k+1) / factorialDyn(2*k+1);
17     k--;
18   }
19
20   return r;
21 }
22
23 // Test with "x" and "k" unknown at compile-time
24 double test_1(double x,int k) { return sineDyn(x,k); }
25
26 // Test with "x" unknown and "k" known at compile-time
27 double test_2(double x) { return sineDyn(x,5); }
28 double test_3(double x) { return sine<5>(x); }
29
30 // Test with "x" and "k" known at compile-time
31 double test_4() { return sineDyn(std::numbers::pi/4,5); }
32 double test_5() { return sine<5>(std::numbers::pi/4); }
```

It appears this time that the compiler cannot always optimize code (Clang 12.0.1 does, but not GCC 11.1) to get the value of sine directly computed when both x and k are known at compile-time, as shown in Table 2.4.

Table 2.4: Evaluation of the sine function implementations for Clang 12.0.1 and GCC 11.1 (cf. Code 2.32)

	Sine				
	No optimisation	**GCC (-O1)**		**Clang (-O2)**	
		Inlined	**Not Inlined**	**Inlined**	**Not Inlined**
test_1	F & C	I	F & C	I	F & I
test_2		I		I & U	F & I & U
test_3		I & U		I & U	F & I & U
test_4		I		V	F & V
test_5		V		V	F & V

F = [F]unction created / C = function is [C]alled / I = effective [I]nline / V = [V]alue computed at compile-time / U = loop [U]nrolled

When used regularly, these possibilities are limited as writing elaborated algorithms becomes rapidly laborious, and the main interest of template metaprogramming is its ability of computation on types as seen in Section 2.3.5. Notably, to manipulate sets of types, the *typelist* and *tuple* [Vandevoorde et al., 2017] structures are commonly used. As we have seen before, it is possible to use traits to define properties of types, SFINAE constructs to test them, and the newest addition, the concepts, allows one to create classes of types.

Lilis and Savidis (Section 2.1.4) classified template metaprogramming as different from the object language, thereby requiring "an entirely different programming style and involving custom coding practices that deviate from common programming styles" [Lilis and Savidis, 2019].

For example, Code 2.33 shows the common way to define a simple *if* statement in template metaprogramming (lines 1-5), with a use case, metafunction `Absolute<N>` that computes the absolute value of `N` (`N` is a class with a `value` attribute) at lines 7-12.

Source Code 2.33: If construct with template metaprogramming

```
template <class TEST, class IF, class ELSE, bool = TEST::value>
struct If : ELSE {};

template <class TEST, class IF, class ELSE>
struct If<TEST, IF, ELSE, true> : IF {};

template <typename N>
struct Absolute : If<
  isNeg<N>,
  Number<typename N::type, -N::value>,
  Number<typename N::type, N::value>
> {};
```

The `if<TEST,IF,ELSE>` construct is defined with a primary template (lines 1-2) that returns `ELSE` whatever the value of `TEST`. The specialization when `TEST` (more precisely its `value` attribute) is `true` returns `IF`. This way, the public members of `IF`, or `ELSE` depending on the value of `TEST`, become the members of `IF<TEST,IF,ELSE>`.

Now, if we consider for instance the OR statement as designed in Code 2.34, the first implementation at lines 2-3 has the inconvenience that `T1` and `T2` need to be evaluated any time `or<T1,T2>` is instantiated, which can possibly lead to long compilation time, notably if this phenomenon occurs recursively in the evaluation of `T1` and `T2`. The second implementation at lines 6-14 will evaluate `T2` only if `T1` is `false`; for this purpose, the two branches of the *if* statement are split in two parts as shown at lines 11-12 and 14.

Source Code 2.34: OR statement with template metaprogramming

```
// Easily readable version
template <class T1, class T2>
struct or : if<T1,std::true_type,T2> {};

// Improved for performance version
template <class T1, class T2, bool TEST> struct _or_if_;

template <class T1, class T2>
struct or : _or_if_<T1,T2,T1::value> {};

template <class T1, class T2>
struct _or_if_<T1,T2,true> : std::true_type {};

template <class T1, class T2> struct _or_if_<T1,T2,false> : T2 {};
```

2.4 Constant expressions

A constant expression is an expression that can be executed at compile-time [Dos Reis et al., 2007]. There are different types of constant expressions; we focus on the ones that are tagged with the `constexpr` specifier. `constexpr` was introduced in C++ 11. It states that an expression *can* be evaluated at compile-time. The `constexpr` follows a long set of rules to be actually applied. We have mainly used it on variables, lambda expressions, functions, constructors, and *if* statements.

The following properties are the most important for our needs and are not meant to be exhaustive:

- A `constexpr` variable must be immediately initialized and `constexpr` implies `const`.

- A `constexpr` function only accepts and returns *literal types*; it does not contain any expression that can produce undefined behavior in the context of the call. An undefined behavior is a situation where the result of an expression is not defined by the C++ norm. For example, in Code 2.39, we try to dereference a pointer to a freed memory (call to destructor at line 4). The content of the pointed memory is thus in an unknown state, therefore dereferencing the now invalid pointer that produces an undetermined result, thus leading to an undefined behavior in the following code.

- A literal type is either a scalar, an array of literal type, a reference, a lambda, or a class type with a trivial `constexpr` destructor, a `constexpr` constructor, or literal type data members.

- A defined `constexpr` constructor should initialize all of its base class and non-static data members (including POD - Plain Old Data - type).

- When calling a `constexpr` function, it can be evaluated at compile-time only if its arguments can be evaluated at compile-time, and each of its inside instructions can be evaluated at compile-time.

If these constraints are met, the variable or function is allowed to be evaluated at compile-time, but it is not mandatory. Indeed, each compiler has its own compile-time execution policy and even if the previous conditions are all satisfied, the compiler can decide not to execute the function at compile-time, based on its own criteria.

For example, Code 2.35 presents two implementations, one recursive and one iterative, of the factorial computation using the `constexpr` specifier. This use case shows that the compilers can handle the computation differently, and that even when a `constexpr` function is called with arguments set at compile-time (this makes the compile-time execution of the function totally valid), there is no guarantee that the function will actually be executed at compile-time. GCC does the computation at compile-time, so the result (2432902008176640000) is directly stored in the variable in the final assembly code. Clang compiles the function which will be called at runtime.

Source Code 2.35: Factorial with `constexpr` specifier (using C++ 11 norm)

```
// C++11 constexpr function using recursion
constexpr unsigned long long factorial(unsigned long long n) {
  return n <= 1 ? 1 : (n * factorial(n - 1));
}

// C++11 constexpr function using iteration
constexpr unsigned long long factorialLoop(unsigned long long n) {
  unsigned long long result = 1;

  for(unsigned long long loop = 2; loop <= n; ++loop)
    result *= loop;

  return result;
}

int main() {
  auto facto = factorial(20);
  auto factoLoop = factorialLoop(20);
}
```

We will not present all the functionalities of the specifier `constexpr`; they are available in reference documentation [8]. However, there have been a lot of changes following the evolution of C++ standards 14, 17 and 20. The specifier gains in options and complexity. For example, in C++ 11, it is not possible to have local variables and loops in `constexpr` functions, they have been introduced in C++ 14. C++ 17 added the `constexpr if` statement and the `constexpr` lambda expressions. With these tools, it is now possible to have most of the basic mechanism to develop natural C++ code at compile-time. C++ 17, `constexpr` enables compile-time evaluation which gathers compile-time function evaluation, constant folding, intra-function cycles and branching. These facilities are discussed by Yauhen Klimiankou [Klimiankou, 2019].

C++ 20 added new complementary features to the `constexpr` that enable the compile-time functionalities of Hedgehog. Allocation in C++ can be summarized for most user to the usage of the `new` (and `delete`) instruction for dynamic allocation. Although there is no `constexpr new` and a `constexpr delete`, instructions `new` and `delete` can be used in a `constexpr` context. In this situation, the allocation and deallocation should be made in the same `constexpr` context.

This has two direct consequences: it is not possible to pass a pointer from a `constexpr` context to a non-`constexpr` context, and no allocation is made in the final runtime code, because only the final result of the `constexpr` is computed. However, this induces a support for allocators, and added to algorithms and utilities, it also allows the usage of dynamic containers like the `constexpr std::vector`.

Source Code 2.36: "`constexpr`" `new` support, direct use

```
1  class ClassTest {};
2
3  int main() {
4    auto constexpr res = [] () {
5      auto classTest = new ClassTest();
6      delete classTest;
7      return 0;
8    }();
9  }
```

Codes 2.36 and 2.37 show two valid (i.e., compiling) codes in `constexpr` contexts. In both of them, the `constexpr` lambda expression allocates and deallocates an object `ClassTest` in the same `constexpr` context, even if the allocation and deallocation are made by separate functions (`allocate` and `deallocate`) called inside the lambda expression (lines 8 and 9, Code 2.37).

Source Code 2.37: "`constexpr`" `new` support, indirect use

```
1  class ClassTest {};
2
3  constexpr ClassTest* allocate() { return new ClassTest(); }
4
5  constexpr void deallocate(ClassTest* classTest)
6  { delete classTest; }
7
8  int main() {
9    auto constexpr res = [] () {
10     auto classTest = allocate();
11     deallocate(classTest);
12     return 0;
```

[8]https://en.cppreference.com/w/cpp/language/constexpr

```
  }();
}
```

Codes 2.38, 2.39, and 2.40, show three invalid (i.e., non-compiling) codes. Code 2.38 does not compile because the memory is not deallocated in the `constexpr` context.

Source Code 2.38: "`constexpr`" `new` support, deallocation missing

```
class ClassTest {};

int main() {
  auto constexpr res = [] () {
    auto classTest = new ClassTest();
    return 0;
  }();
}
```

Code 2.39 does not compile because we try to dereference a pointer that references memory that has been deallocated, which is an *undefined behavior*. The GCC compiler returns with the following error: *use of allocated storage after deallocation in a constant expression.*

Source Code 2.39: "`constexpr`" `new` support, undefined behavior

```
int main() {
  auto constexpr res = [] () {
    int *i = new int {42};
    delete i;
    auto value = *i;
    return 0;
  }();
}
```

Code 2.40 does not compile because the code returns a pointer referencing memory that was allocated in the `constexpr` context. GCC v.11.1 returns with the following error: *is not a constant expression because it refers to a result of 'operator new'.*

Source Code 2.40: "`constexpr`" `new` support, return of allocated memory

```
class ClassTest {};

int main() {
  auto constexpr res = [] () {
    auto res =  new ClassTest();
    return res;
  }();
}
```

The last major addition to C++ 20 that we heavily use is the support of `constexpr` virtual member function, which also comes with dynamic casting. It is however not possible to use virtual inheritance on a literal class. These additions allow the creation of much more complex codes. We showcase our usage of these recent tools in Section 2.

2.4.1 Constant expressions in Lilis and Savidis' taxonomy

We can situate metaprogramming with constant expressions in Lilis and Savidis' taxonomy. In contrast to template metaprogramming, which is a different language from the object language (C++), `constexpr` is just a specifier added to the core language. In this case, the metalanguage is indistinguishable from the core language, but only a subset of the language is available. The metalanguage source is embedded into the object language source, and is possibly evaluated at compilation-time. It proposes mainly generative programming. The main idea is to do complex computation at compile-time and produce a result that can be used during execution.

2.4.2 `constexpr` vs. `consteval`

There is a variant of the `constexpr` specifier called `consteval`. It forces the compiler to perform the function's computation at compile-time, as opposed to `constexpr` which only ensures that the function can possibly be executed at compile-time, as seen previously. Such functions are called *immediate* functions.

There are no tools embedded in the language that help debug compile-time code. Debugging has proven to be a challenge with common template metaprogramming; this problem remains when using constant expressions and is even magnified when aiming for the design of complex algorithms. If we had chosen to only use immediate functions, we would have found no help in common tools, while with `constexpr` functions we can. `constexpr` allows the compiler to execute the functions at compile-time. This means that is it possible to also compile them as regular code and execute them at runtime. Because it is common code, we can use common tools such as *GDB* or IDE visual debuggers to debug them.

Because of the possible virtuality added in C++ 20, the code we are proposing is not only a bundle of routines ready to be called, but also a library that end-users can extend. If the API was just exposing `consteval` methods, any derived method would also have been `consteval`. This means that the method would have been much more difficult to debug and test for an end-user. This is why we only use `constexpr` on our code, making it easier to debug and also that `constexpr` code is executed at compile-time.

2.4.3 Template metaprogramming and constant expressions

Coexistence

Constant expressions with `constexpr` are the newest addition to C++ to do metaprogramming. There is no question about replacement of traditional template metaprogramming by constant expressions. They are both targeting different types of computation. Template metaprogramming has a lot of possibilities to deal with types while `constexpr` expressions allow more traditional computations at compile-time. It is also possible to express some computation with both techniques. For example in Codes 2.41 and 2.42, we propose two functions, `uniqueConstexpr` and `uniqueTMP`, that use constant expressions and template metaprogramming techniques respectively to remove duplicate types from a tuple.

Source Code 2.41: Removing duplicates from tuples, using constant expressions

```
template<class Current, class ... Others>
static auto uniqueConstexpr() {
  if constexpr (sizeof...(Others) == 0) {
    return std::tuple<Current >{};
  } else if constexpr (std::disjunction_v<std::is_same<Current,
  ↪  Others>...>) {
    return uniqueConstexpr<Others...>();
  } else {
    return std::tuple_cat(std::tuple<Current>{},
    ↪  uniqueConstexpr<Others...>());
  }
}
```

Source Code 2.42: Removing duplicates from tuples, using template metaprogramming

```
template <typename T, typename... Ts>
struct uniqueTMP : std::type_identity<T> {};

template <typename... Ts, typename U, typename... Us>
struct uniqueTMP<std::tuple<Ts...>, U, Us...>
  : std::conditional_t<(std::is_same_v<U, Ts> || ...),
    uniqueTMP<std::tuple<Ts...>, Us...>,
    uniqueTMP<std::tuple<Ts..., U>, Us...>> {};

template <typename... Ts>
using unique_tuple =
  typename uniquetemplate metaprogramming<std::tuple<>,
  ↪  Ts...>::type;
```

In this example, the `constexpr` function `uniqueConstexpr` manipulates values (from which types can be deduced), and template metaprogramming manipulates types directly. Another difference is that `void` cannot be manipulated by `uniqueConstexpr` directly, because it can not be instantiated, while `uniqueTMP` has no such problem. In Hedgehog, we use the `uniqueConstexpr` version with pointers only, which avoids this problem.

Template metaprogramming criticism

Template metaprogramming is the most common way to do computation at compile-time in C++. The technique is not exempt of criticisms.

Google through its C++ style guide [9] formulates the following ones.

- Template metaprogramming is often obscure to non-experts.
- It is often unreadable, hard to debug and maintain.
- It has poor compile-time error messages because implementation details become visible.
- It makes it hard large code base refactoring.
- It is easy to go "too far" and be "overly clever".

[9] https://google.github.io/styleguide/cppguide.html#Template_metaprogramming

However, they recognize that it allows to create cleaner and easier-to-use interfaces and they suggest to minimize and isolate template metaprogramming code and add comments to help end-users debug their code in case of error.

They cover until the C++ 17 norm, which means they do not include “concepts” in their guide. Concepts are the main improvement of template metaprogramming in C++ 20. They definitely help for debugging and maintenance related issues with clearer error messages. If the program fails when trying to check the requirements for a type, the compiler will point to the concept itself and the corresponding failing requirement(s).

Source Code 2.43: Output from Code 2.27, with a faulty case

```
<source>: In function 'int main()':
<source>:33:14: error: no matching function for call to
↪ 'doCatThings(NotACat&)'
   33 |     doCatThings(whatCouldItBe); // Compilation failure
      |     ~~~~~~~~~~~^~~~~~~~~~~~~~~
<source>:23:6: note: candidate: 'template<class auto:1>  requires
↪ CatConcept<auto:1> void doCatThings(auto:1&)'
   23 | void doCatThings(CatConcept auto & cat){
      |      ^~~~~~~~~~~
<source>:23:6: note:   template argument deduction/substitution
↪ failed:
<source>:23:6: note: constraints not satisfied
<source>: In substitution of 'template<class auto:1>  requires
↪ CatConcept<auto:1> void doCatThings(auto:1&) [with auto:1 =
↪ NotACat]':
<source>:33:14:   required from here
<source>:4:9:   required for the satisfaction of
↪ 'CatConcept<auto:1>' [with auto:1 = NotACat]
<source>:4:22:   in requirements with 'T aCat' [with T = NotACat]
<source>:6:12: note: the required expression 'aCat.meow()' is
↪ invalid
    6 |     aCat.meow();
      |     ~~~~~~~~~^~
<source>:7:12: note: the required expression 'aCat.purr()' is
↪ invalid
    7 |     aCat.purr();
      |     ~~~~~~~~~^~
```

In the case of Output 2.43 which is the compiler output from the faulty case commented at line 33 of Code 2.27, the compiler indicates clearly that when instantiating function `doCatThings(NotACat&)`, the constraints are not satisfied: the expressions `aCat.meow()` and `aCat.purr()` are not valid for the `NotACat` type.

Source Code 2.44: Output from Code 2.28, with a faulty case

```
<source>: In function 'int main()':
<source>:41:30: error: no matching function for call to
↪ 'doCatThings(NotACat&)'
   41 |     doCatThings(whatCouldItBe); // Compilation failure
      |                              ^
<source>:34:6: note: candidate: 'template<class T, class> void
↪ doCatThings(T&)'
   34 | void doCatThings(T & cat){
      |      ^~~~~~~~~~~
<source>:34:6: note:   template argument deduction/substitution
↪ failed:
In file included from <source>:1:
```

```
10  /opt/compiler-explorer/gcc-10.2.0/include/c++/10.2.0/type_traits: In
    substitution of 'template<bool _Cond, class _Tp> using
 ↪  enable_if_t = typename std::enable_if::type [with bool _Cond =
 ↪  false; _Tp = void]':
11  <source>:32:9:   required from here
12  /opt/compiler-explorer/gcc-10.2.0/include/c++/10.2.0/type_traits:
 ↪  2554:11: error: no type named 'type' in 'struct
 ↪  std::enable_if<false, void>'
13   2554 |       using enable_if_t = typename enable_if<_Cond,
  ↪  _Tp>::type;
14        |                                         ^~~~~~~~~~~
15  Compiler returned: 1
```

In the case of compiler's Output 2.44 using the SFINAE equivalent Code 2.28 of Code 2.27 based on concepts, no clear information about what is causing the problem is exposed, only that the `enable_if` of `doCatThings` was not able to produce a type. Therefore, its condition fails.

However, the concepts, as they are for now, do not change the template language itself, so if custom metafunctions need to be developed, they will be usually of no help.

`constexpr` criticism

The `constexpr` specifier had tremendous improvements in C++ 20 as presented in Section 2.4. Because it is so recent, there is little published research on its use or a library fully exploiting the novelties added.

The receptions of this specifier are mitigated, some like Ben Dean and Jason Turner suggest to "`constexpr` all the things" in their proposal for C++ [Deane and Turner, 2017]; others like Yauhen Klimiankou consider it as a "great good but wrong idea" [Klimiankou, 2019]. Yauhen Klimiankou in his paper expresses 7 critiques over `constexpr`.

- First, `constexpr` is only a hint for the compiler. The difference between equivalent functions that can be evaluated at compile-time depends on the presence of the keyword and not on the ability to evaluate the functions. According to the author, the compiler should be able to do it by itself and take care of these low-level optimizations in the case of compile-time variable initialization.

- Second, `constexpr` brings to C++, like template metaprogramming, a new language that the author refers to as *Generalized Constant Expression C++ (GCEC++)*. For him, this metalanguage is different from the object language. There are two reasons behind considering this a new language: the first is to guarantee "cross-compiler portability" and the second is because we can only use a subset of the C++ language.

- Third, the author says this tool is only a "tool for the psychological satisfaction of the programmer", giving him/her a "feeling of power and mastership" while the compiler could do it better.

- Fourth, in the history of C++, `constexpr` is close to what has been the `register` specifier removed in C++ 17, and close to the `inline` or `volatile` specifiers.

They all are designs that do not add new semantics and are optimization dedicated solely for optimization enforcement.

- Fifth, the C++ standard does not specify resources limits used by compile-time function evaluation. This means that a code can be compiled by a compiler while reflected by another. As such, cross-compiler compatibility cannot be guaranteed.

- Sixth, a modification of a function from being compile-time to runtime or conversely, implies a cascade of changes to preserve language semantics. For instance, to declare a function as `constexpr`, one needs to declare also all called functions as `constexpr`.

- Seventh, the author claims having better results with an automatic post-compilation tool rather than using `constexpr`.

Notice that we have observed some behavior of the compilers during our study of the various implementations of `power` and `sine` functions (cf. Section 2.3.9) that suggests that compilers decide, to some extent, to evaluate at compile-time functions that are not marked as `constexpr`. From our experience, we consider that the `constexpr` facility does not imply using a different language than the common C++. Even if it only uses a subset of the possibilities of the language, it exploits the same language with the same syntax and the same grammar as C++ and that is one of the main differences with template metaprogramming. Even it were possible to express some computation with template metaprogramming, it is much more complex and different than the common C++ language compared to `constexpr` expressions. Furthermore, we disagree when the author says that this addition is just a hint. It allows with the same language to express compile-time execution. If we let the compiler choose automatically whether to process some code or not at compile-time, how do we debug code processed at compile-time with common tools? In our opinion, it is always better to have an explicit keyword that gives the possibility to the programmer to choose when to do the processing. In C++ 20, it is possible to use `consteval` to force a computation at compile-time, and `constexpr` is the best of both worlds because it allows one to do computation at compile-time and to do debugging with common tools with the exact same code.

2.4.4 Usage of metaprogramming

We found many usages of metaprogramming techniques in different types of libraries.

The C++ standard library uses `constexpr` to extend the variety of objects and tools available at compile-time.

Kokkos [Edwards et al., 2014b], a library used for manycore parallelism on clusters with MPI, targets HPC applications. It also comes with a co-library, kokkos-kernel, specialized for linear algebra. They claim to use template metaprogramming in order to optimize code for some architectures [Edwards and Sunderland, 2012].

We found libraries like Eve [Penuchot et al., 2018] or Spy [Falcou, 2019] that use the `constexpr` specifier. Eve is a C++ 20 wrapper implementation around SIMD extensions sets for different architectures. Spy is a C++ 17 library that gathers information on the architecture used and made it available in `constexpr` context.

MetaPASS [Hollman et al., 2016], for "Metaprogramming-enabled Parallelism from Apparently Sequential Semantics", is a proof-of-concept library that analyzes a sequential code and deduces a parallel code using a directed acyclic graph (DAG) representation. It relies on a dependency analysis, using template metaprogramming, of read/write variable access to derive a DAG of the execution to be performed. The C++ type system (reference, `const`) is analyzed with metafunctions and SFINAE constructs, to derive the usage of the variables.

2.5 Conclusion

In this chapter, we presented different techniques to do computations at compile-time. We mainly focused on different taxonomies to sort those approaches and on what is nowadays available with the C++ language. The two main approaches for static metaprogramming in C++ are template metaprogramming and metaprogramming based on constant expressions. Both approaches are complementary because they bring different kinds of compile-time capabilities to the language.

Template metaprogramming is particularly fitted to do computation on types. This technique is widely used in the standard library, notably to design traits and metafunctions. It can also be combined with the SFINAE rule, using the `enable_if` construct, to remove functions from the overload resolution. The latest additions to this field are concepts and constraints. They allow one to define sets of named constraints to filter types used in class or function template parameters, and are a true alternative to `enable_if`.

Constant expressions allow one to do complex computations on values (either of primitive types or objects). However, they only propose a subset of the C++ language that can be executed at compile-time. The possibilities offered by constant expressions are growing with the evolution of C++, and yet, it is possible to create extensible libraries through inheritance.

We utilize both approaches in the design of our library Hedgehog, as explained in Chapter 5. Our proposal is based on the manipulation of a data-flow graph, known at compile-time, where some checking and analysis is necessary. Template metaprogramming is used to secure our API and add some constraints on types to check the coherence of the data-flow graph designed by a user of the library. And constant expressions are used to create a fully extensible compile-time library to perform advanced analysis on the data-flow graph before execution.

Part II

Design

Chapter 3

Hedgehog framework

In Chapter 1, we presented different ways to achieve parallelism. We articulated that a good approach should be accessible, explicit, present intrinsic parallelism, and present composability properties to help design complex algorithms and enable code sharing.

Our solution is a library called Hedgehog. It is a revised version of the Hybrid Task Graph Scheduler (HTGS) made by Timothy Blattner and Walid Keyrouz at NIST. It is mainly composed of two parts, the runtime system that is presented first in this chapter and a compile-time system that is presented in Chapter 5.

To introduce the runtime system, we first overview the original system HTGS (briefly introduced in Section 1.7.2), with a few of its limitations, in Section 3.2. Then, we expose the early decisions that we made to develop Hedgehog in Section 3.3. With all the context described, we then present the Hedgehog model in Section 3.4, with a special focus on a core design aspect, the separation of concerns, in Section 3.5. After, in Section 3.6, we dive into the library architecture.

The base model presented in this chapter and our results in Chapter 4 were published by IEEE [Bardakoff et al., 2020a], presented at the Nvidia *GPU Technology Conference* held in 2020 [Bardakoff et al., 2020b], and at the HiHAT[1] online workgroup.

3.1 HTGS

HTGS [Blattner et al., 2017, Blattner, 2016] is a library developed to help conceiving parallel applications on a single heterogeneous compute node. It is available on GitHub[2] and is used for internal applications at NIST or by external users.

3.1.1 Model and types of nodes

HTGS bases its representation of the computation of a program on an explicit data-flow graph, with a well-defined type of graph nodes. The data-flow graph execution consists of statically associated threads to each node from the graph's construction to its destruction. It aims at managing coarse-grained parallelism to amortize the cost of sending data from one node to another.

Parallelism granularity is a qualitative measurement of the amount of work done for a task over the communication time. It is opposed to fine-grained parallelism

[1] `https://hihat-wiki.modelado.org/`
[2] `https://pages.nist.gov/HTGS/`

which defines the type of computation composed of small tasks. For example, a parallel addition over all elements of an array with vectorization instruction is considered fine-grained parallelism, as a task, i.e. the computation for an element, is on the order of nanoseconds. We designed HTGS and Hedgehog to target coarse-grained parallelism, their tasks ingest big pieces of data and therefore have long computation times. They present reasonable communication times (what we call latency, cf. Section 4.2.1) from $1\,\mu s$ to $10\,\mu s$.

The graph nodes of HTGS have one input type and one output type, meaning that a node can only receive one type of data and produce one type. Each node possesses a queue to store incoming data corresponding to the node's input type. Each node can only have one output edge and there is only one connection possible between two nodes.

When a node receives a piece of data, it wakes up its associated thread(s), pops the piece of data out of the input queue, executes the kernel with this data, and if the end-user chooses so, sends output data to the next node. If the queue is not empty, the node will automatically pop another element and repeat this process.

For usual graphs, these nodes can be (1) computational tasks or (2) bookkeepers as shown in Figure 3.1. More complex graphs can present nodes that are graphs, they wrap them into specialized tasks or into structures called execution pipelines.

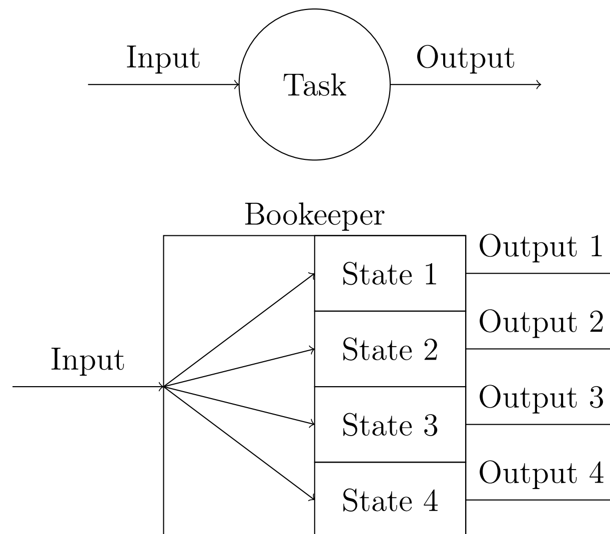


Figure 3.1: Task and bookkeeper in HTGS

The computational tasks, or simply tasks, are the nodes where the computation kernels are deployed. The graph tasks can be duplicated by the runtime system and associated with multiple threads.

The bookkeepers are a type of nodes for managing the computations. They are single-threaded and associated with a set of rules to represent the different options the bookkeepers will traverse. A bookkeeper and its rules present the same input types. The bookkeeper by itself does not have an output type *per se* but its rules may have diverse output types and be linked to different nodes. If we consider the bookkeeper and its rules, this gathering produces multiple output types and can broadcast data to multiple other nodes. When a bookkeeper receives a piece of data, it will send it sequentially to the different rules, and depending on the computation, a rule sends output data on its output edge.

Two more categories of nodes are available to an end-user: (1) the graph by itself, and (2) the execution pipeline. The role of a graph is to gather a set of nodes to present a black-box abstraction to the user. A graph can be transformed into a special type of node, a `TGTask` for "Task-Graph Task". This `TGTask` can be then embedded in another graph as a normal task.

In order to tackle heterogeneous computing, a specialized task abstraction for Nvidia GPUs, the `CUDATask`, was created. Such a task works like a normal task; it is just associated with a specific GPU at initialization and allows one to execute the computation on this GPU. A design decision constrains all `CUDATask`s in a graph to be bound to the same device. To achieve multi-GPU computation, the execution pipeline needs to be used. The execution pipeline will duplicate a graph, and its inner nodes recursively, and binds each of the copies to different GPU. We can also choose in which graph of an execution pipeline to send the data to.

3.1.2 Memory manager

One of the major problems when dealing with GPU computation is memory management. The memory available in a GPU is limited compared to the memory available in a CPU node. For example, on an Nvidia DGX A100, there are 8 Nvidia A100 with 80 GB of memory each (640 GB total) compared to the 2 TB of memory available to CPU. A base principle to get performance on a GPU is to feed the GPU while staying within its capacity limit while avoiding allocation during the execution to avoid synchronization.

In HTGS, the memory manager was designed to help end-users deal with these problems. The memory manager has a pool of memory allocatable on a device that lives in its own thread. The memory manager is then associated with tasks that can pull data from it. With this construct, it is possible to throttle the computation by limiting the quantity of memory available and staying below the threshold available in the device while improving the locality. There are different types of memory managers, static or dynamic depending on when the allocation happens, before the execution of the memory pool creation (to avoid GPU synchronization for example) or during the execution, respectively.

3.1.3 Graphical feedback

HTGS's explicit model allows the end-user to understand the computation. But sometimes the computation does not run as expected, deadlocks, heavyweight tasks, or other problems can occur. In order to get a view of how the execution behaves, a graphical feedback mechanism was put in place. The idea is to gather metrics at runtime, at the node level, to understand the execution. A color code allows us to understand where the critical path is; also it detects where a deadlock occurs by analyzing how much data are left to be processed on nodes.

3.2 HTGS limitations

HTGS as it is presented has some limitations.

3.2.1 `IData` interface

All data that go through an HTGS graph need to inherit from the `IData` class. This means that even an `int` needs to be embedded into a class, `MyInt` for example, that must inherit from `IData` to be used in a graph.

3.2.2 Graph manipulation

In order to construct and use a graph, an end-user needs to:

1. Create an empty `TaskGraphConf`.
2. Create the graph structure with the `TaskGraphConf` by adding edges between the different graph's nodes.
3. Transform a `TaskGraphConf` into a `TaskGraphRuntime`.
4. Execute the `TaskGraphRuntime`.
5. Push data to the `TaskGraphConf` with the `produceData` method.
6. Indicate to `TaskGraphConf` that no more data will be sent to the graph.
7. Get result out of the `TaskGraphConf` with the `consumeData` method.
8. Wait for the runtime, the `TaskGraphRuntime`, to terminate.
9. If needed, the graphical output can be produced from the `TaskGraphConf`.

This process is complex and requires handling multiple objects like `TaskGraphConf` and `TaskGraphRuntime`. It may be possible to simplify this API for end-users.

3.2.3 Graph construction

To construct the graph and, in particular, define edges between two nodes, several methods in the API exist:

- `addEdge`: creates an arc between two tasks or bookkeepers.
- `addRuleEdge`: creates a link between a bookkeeper and a rule.
- `addMemoryManagerEdge`: creates a link between a memory manager and a task for "normal" data.
- `addCudaMemoryManagerEdge`: creates a link between a memory manager and a `CUDATask`.
- `setGraphConsumerTask`: sets a node as the graph's input.
- `setGraphProducerTask`: sets a node as the graph's output.

Internally, the different edge objects are connected to nodes through interfaces named *connectors*. They are used to manage the input and output of `IData` between tasks.

Two questions arise: is it possible to simplify the creation of the edges? Is it necessary to use the connectors?

3.2.4 Bookkeeper broadcast

The bookkeeper is an HTGS node that helps to manage the computation. It is combined with different rules to conditionally apply a set of computations. Because each rule can send data to a node, the bookkeeper and the rules can send the same data to multiple nodes and therefore redirect the flow of data.

It may be a good option to generalize the possibility to send a piece of data to multiple nodes for any node. We could also study the idea for a node to accept or produce multiple types of data.

3.2.5 Internals

Looking at the code, a lot of internal methods that should be hidden are available to the end-user through the API. For example, the *TaskGraphConf* initialization is directly callable; it is also possible to access the underlying connectors and set them etc.[3] From the `ITask` objects, we can get all information about the graphical feedbacks, the internal identifiers of the connectors, etc.[4] A new architecture to separate the internals or limit their access from the end-user could be implemented to ease the usability.

Furthermore, manager classes, like the different types of `TaskManager` used to "encapsulates an `ITask` to interact with an `ITask`'s functionality", or the `RuleManager` used to "connects a `Bookkeeper` to another `ITask` using one or more `IRule`(s)" seem redundant. An `ITask` should not need a wrapper to access its own functionality and the `Bookkeeper` should only use its `IRule`(s) to be connected to other `ITask`.

3.2.6 Memory manager

A memory manager is associated with a thread. However, it is usually used with a task that also has its own thread. The overall model can use fewer resources by removing these extra threads and handle a memory manager as an extra tool used by active tasks.

3.2.7 Conclusion

These limitations give us opportunities to design Hedgehog. The new architecture should offer a clear separation between the internals and the API for end-users. About the internals, the interfaces between the different nodes can be reworked to offer more flexibility and allow the graph to accept any type, even primitives types. The API can also be revised to offer simpler objects with simpler interactions to build the graph.

3.3 Early decisions

Our goal when designing and developing Hedgehog is to propose a library succeeding HTGS. It aims at the same purpose, helping developers create parallel libraries with a graph-based model. It should at least provide the same features as HTGS.

[3]https://pages.nist.gov/HTGS/doxygen/classhtgs_1_1_task_graph_conf.html
[4]https://pages.nist.gov/HTGS/doxygen/classhtgs_1_1_i_task.html

All limitations that we have presented in Section 3.2 need to be tackled and are opportunities for extra features. In terms of usability, we target an API close to HTGS while simplifying the manipulation of the model. In terms of performance, Hedgehog needs to be at least as fast as HTGS. In this section, we present what are the early thoughts that we had to design Hedgehog.

3.3.1 Multiple inputs and output broadcast

Limitation description

This section is made to express the idea and examines the repercussion of changing a task's behavior from having one input and one output to multiple inputs and outputs. HTGS uses one interface for all execution kernels, the template class `ITask`. An `ITask` (that will be specialized to define specific tasks) has two template parameters, the first is its input type, the second is its output type. As such, a task of class `T` inheriting from `ITask<I,O>`, will accept an object `I` as input and will produce an object of type `O`. Another HTGS class that is affected by the proposed modification is the bookkeeper. A bookkeeper is a special task that will have different outputs depending on the rules it follows. These rules specify a scheduling over input data, and each rule executes on that data synchronously. Therefore, a bookkeeper `BK<I>`, will accept an object `I` as input and can produce object `O`, `P`, `Q`... depending on the rules.

Multiple inputs

The third HTGS tutorial (matrix multiplication [5]) was chosen to illustrate the discussion. Figure 3.2 presents the matrix multiplication task graph.

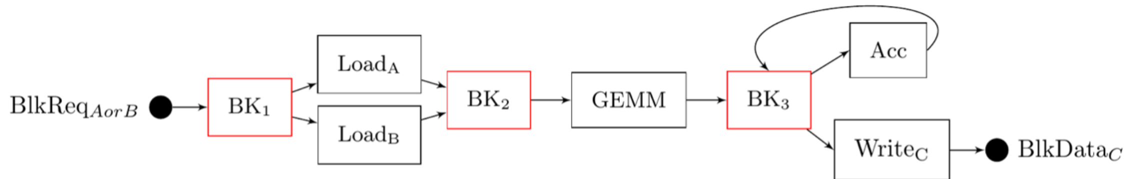


Figure 3.2: Graph representation of the matrix multiplication algorithm in HTGS

This graph presents four major steps: (1) the loading of matrices A and B blocks into memory ($Load_A$ and $Load_B$), (2) the block matrix multiplication ($GEMM$), (3) the accumulation (Acc) of resulting blocks, and (4) the output of C matrix blocks ($Write_C$). Three bookkeepers (BK_1, BK_2, and BK_3) are used to control the flow of data.

Focusing on the first bookkeeper $BK1$, it just takes the same objects (`MatrixRequestData`), either it receives a block of matrix A or matrix B, with a flag in the data indicating to which matrix it belongs. Then depending on this flag, the data are sent to task $Load_A$ or $Load_B$. To define the load task, only one method `executeTask` needs to be overridden from class `ITask` as shown in Code 3.1.

[5] `https://github.com/usnistgov/HTGS-Tutorials`

Source Code 3.1: `LoadMatrixTask` basic structure

```
class LoadMatrixTask
: public htgs::ITask<MatrixRequestData, MatrixBlockData<double *>> {
  //...
  void executeTask(std::shared_ptr<MatrixRequestData> data) override
  {/*...*/}
  //...
};
```

Instead of having one bookkeeper just checking for a flag in each input data and then producing the data to the right task, we propose to have multiple inputs. The task `LoadMatrixTask` would be rewritten as shown in Code 3.2.

Source Code 3.2: `LoadMatrixTask` rewritten

```
class LoadMatrixTask :
  public htgs::ITask<
    MatrixBlockData<double *>,
    MatrixRequestDataA, MatrixRequestDataB
  > {/*...*/};
```

The first template parameter of class `ITask` would be the output type, and the last ones the multiple input types. This means that the task will accept a piece of data of type `MatrixRequestDataA` *or* `MatrixRequestDataB` and then have different functionality for each type as shown in code 3.3. Methods `executeTask` will still be overridden from `ITask` to provide an overload for each kind of input data.

Source Code 3.3: `LoadMatrixTask executeTask` rewritten

```
virtual void executeTask(std::shared_ptr<MatrixRequestDataA> data)
{/*...*/}

virtual void executeTask(std::shared_ptr<MatrixRequestDataB> data)
{/*...*/}
```

The idea is to reduce the complexity of the graph building code (avoiding unnecessary bookkeepers), by adding more input data types that represent the current execution. A task will also be able to accept different types of objects which have common processing.

For the next point, let us take as an example the following task: `class Task : public ITask<O, I1, I2>`. This task will accept `I1` or `I2` as input and `O` as output. By accepting `I1` or `I2`, we mean that the task has a computation kernel implemented for each type. A problem arises if a kernel needs to use both of them. A first solution would be to store in the instance of the task pieces of data from `I1` or `I2` when they arrive and use them when a piece of data from the other type arrives. Another solution would be to rewrite the task as `ITask<O, I1, I2, I3>`. `I3` in this case would be a union of `I1` and `I2`. This latter scenario, the task proposes three kernels: one for `I1`, one for `I2`, and one for `I3` that represents the execution done for `I1` and `I2` together. This second solution requires a bookkeeper in front of the task to produce `I3` from `I1` and `I2`.

Output broadcast

Each task can only have one output edge sending a single type of objects. The task's produced output will become part of another task's input. The exception is the bookkeeper that will be able to send different kinds of objects to different nodes depending on its rules. A task can only be implied in one call of the `addEdge` method as source node when representing the links between tasks in the data-flow graph.

We propose that a task should be able to broadcast its output to multiple tasks that share the same object type. So, a task *T1* will be able to send its output to *T2* and *T3* for instance, as represented in Code 3.4.

Source Code 3.4: Broadcast writing proposal

```
1 graph->addEdge(T1, T2);
2 graph->addEdge(T1, T3);
```

Conclusion

The new interface for the `ITask` will allow to have multiple input types and one output type that can be broadcast to multiple tasks. The task representation brings then more flexibility and the graph gains in expressiveness.

The multiple inputs need to be expressed in some way with C++. The language feature that we need to use is variadic templates (cf. Section 2.3.2). We choose to express the output first, instead of the inputs, because of the way the variadic templates work. If the template parameters were written `<I1, I2, I3, O>` by the end-user, the abstraction would be the parameter list `<class... Types>`. It is possible to extract the last type of the list with a simple metafunction, but in our design, we need to have direct access to the type of output. As the parameter list `<class... Inputs, class Output>` is not a valid syntax (the pack representing variable parameters in a template is always the last one in the list of parameters), our choice as a design is to have the output declared first and then the list of input types: `<class Output, class... Inputs>`. In this case, the output type `Output` is directly accessible, and the inputs are accessible as a pack `Inputs`, or they can be unfolded.

For the same reason we do not, as a first version of the library, allow a node to send different types of output. The main concerns are to express two lists of types in the same template declaration and be able to capture the correct types (the ones for the input types and the ones for the output types) to express some form of inheritance. If we express this with only one pack, `<class... Types>`, we can not differentiate the input types from the output types, and `<class... Inputs, class... Outputs>` is not a valid syntax.

3.3.2 Major API terminology changes

From the bookkeeper to the state manager

As explained in the previous section, all Hedgehog's nodes can send their output to multiple successors, something that was only possible in some way by the bookkeeper with HTGS. The other role of the bookkeeper is to manage the flow of data within a task graph.

The bookkeeper does the management of the flow of data for a task graph between a set of nodes. It can then be seen as a local computation doing state management. This is where the *state manager* terminology comes from. A state manager manages a *state*. We have renamed the bookkeeper's rules the *state* and this is why we have transformed the HTGS bookkeeper node and rules to tightly couple state and state manager.

Graph terminology

The graph model in HTGS is designed as a task graph. The choice was made to separate configuration from usage during the execution of the graph representation, the `TaskGraphConf` and the `TaskGraphRuntime`. In Hedgehog we merged both into the `Graph` object. It is used to create, run and interact with the computation.

To connect different nodes in the HTGS graph, the library used different types of edges depending on what is connected: an *edge* (producer / consumer edge) for connecting tasks or bookkeepers, a *rule edge* to link a rule to its bookkeepers, or a *memory manager edge* or a *CUDA memory manager edge* to connect a memory manager to a task depending on the type of memory managed. In Hedgehog we decided to give less importance to the edge and to have only one type of edge for everything in our API.

3.3.3 Key decisions from HTGS

Hedgehog being the successor of HTGS, it follows the same philosophy as the original library. The separation of concerns put in place to separate computation clearly nodes (tasks), and management nodes (bookkeepers) is a clear asset of the library. The threading model without a scheduler allows having a clear execution policy based solely on the graph representation. The representation under the form of a graph allows wrapping nicely a set of coherent tasks. It allows also sharing a computation via the graph abstraction. Heterogeneity is dealt with at a graph level, a graph is assigned to a device. To achieve multi-GPU computation, the GPU graph is duplicated *via* an execution pipeline to the different GPUs. The graphical representation is also an asset of the library to understand how the computation went on the current hardware.

3.3.4 Conclusion

In this section, we presented the technical and conceptual origins of Hedgehog. HTGS is a task graph library with an explicit model focusing strongly on a strict separation of concerns concerning the objects available through the API. HTGS presents some limitations in its API, and in the way, the internals are exposed and used. These limitations are opportunities for designing new functionalities and improvement for Hedgehog. While maintaining the philosophy of the model of HTGS, Hedgehog proposes a more flexible library with a simpler interface. This simplicity for the end-user requires adding more control from the library at compile-time to avoid design errors, as presented in Section 5.1. A detailed description of the architecture is presented in Section 3.6. Hedgehog has also refined the execution model used as presented in the following section.

3.4 Hedgehog model

Hedgehog like HTGS has an explicit runtime system. It is explicit because the objects defined by the end-user are the ones used during the execution with no transformations like many parallel runtime systems would do as seen in Section 1.7. What is designed by the end-user is exactly what is executed by the system, which allows the user to understand what happens during execution and get exploitable feedback from profiling as seen in Section 4.1. HTGS refers to its model as a "task graph". We have refined in Hedgehog the model denomination. We first present this representation in Section 3.4.1 and then its specificities in Sections 3.4.2, 3.4.3, and 3.4.4.

3.4.1 Hedgehog data-flow graph

Hedgehog bases its representation on a data-flow graph.

For instance, the data-flow graph in Figure 3.3 is composed of data streams represented by the edges and of tasks represented by the vertices.

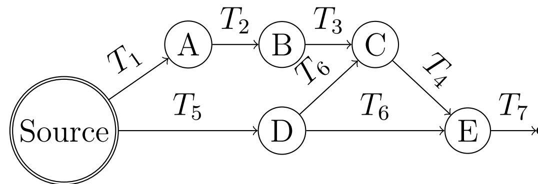


Figure 3.3: Hedgehog data-flow graph model

The term "task" refers here to any type of computation. A node is therefore an instance of an execution kernel that will take an input on one of its input edge, do its computation, and send its results on its output edges. In the example: A, B, C, D, E are the nodes. A node in Hedgehog can accept multiple input types, for example, node C accepts two types as inputs T_3 and T_6. A node is defined by its input types and its output type, among other properties.

The arcs are just the data stream support. They are explicitly created by the end-user but have no object representation. They are directional and support only one type. To be able to draw an edge between two nodes, the output type of the sender should correspond to one of the input types of the receiver, that is our compatibility rule (checked at compile-time). There is no restriction on creating an edge between two nodes if they are compatible; it is then possible to draw directed cycles in the graph (such cycles can potentially lead to deadlock, and are thus detected at compile-time to emit a warning).

The graph has only one entry point, the source node (double circle in the figure), and one exit point, the sink node (the dot in the figure). The source and sink are parts of the graph. When a user pushes data into the graph, it interacts with the source that will transfer the data to its successor nodes (i.e. the graph input nodes). These input nodes are specifically tagged by the developer. By symmetry, the output nodes are the nodes that create the final output data, transfer them to the sink to make them available to the client code. In the example A and D are the input nodes and E is the output node. We can see that in this example, the graph accepts T_1 and T_5 types of data and produces T_7 type of data.

In the graph API that we present later, the input and output types are defined explicitly when defining the graph object. To set a node as input of the graph, it should share at least one input type in common with the graph. To set a node as an output of the graph, it should share the same output type as the graph. In addition to these two rules there are other requirements to fully build a Hedgehog graph that have to be checked at compile-time.

3.4.2 Data pipelining

The data pipelining is a direct consequence of our execution model. For a sequence of linked tasks that run independently from each other, it is possible to execute concurrently many tasks on a set of data.

In the example presented in Figure 3.4, there are three connected tasks: *Read*, *Compute*, and *Write*. *Read*'s output is *Compute*'s input, and *Compute*'s output is *Write*'s input. The execution mechanism for each task is simple: at the moment a new piece of data is available for a task, the task grabs it, does its processing, and sends the output data to the successor task.

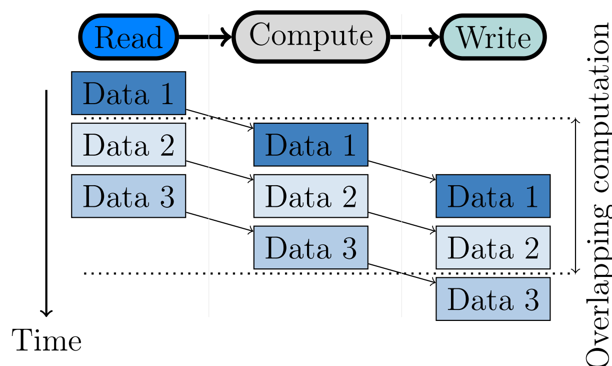


Figure 3.4: Data pipelining

At the beginning, "Data 1" is sent to the pipeline and only the *Read* task becomes active. When *Read* has done its processing, it sends its output to *Compute* (that becomes active) and gets a new data "Data 2". At this moment, two tasks are executing concurrently on a unique set of data in a streaming fashion.

This mechanism is important because it keeps the hardware busy and overlaps memory transfer and computation. A way to do heterogeneous computing is to have in sequence for instance: a CPU task, a copy to GPU task, a GPU task, a copy to CPU task, and a CPU task. In this sequence, the copies happen as soon as a piece of data is ready to be copied, the copies in and out of the device are overlapped with computations, and the computations from CPU and GPU are overlapped.

One crucial parameter in such systems is latency: the time to transfer a piece of data between tasks. The latency will determine how fine-grained the data need to be streamed into the graph and the execution time of the tasks. The higher the latency, the more this time needs to be amortized by an important computation, which means having a huge amount of data for a task and/or a lot of computation in a task. On the contrary, if the latency is small, the system is able to achieve more fine-grained parallelism.

In Hedgehog, we have designed a visual feedback to help the developer understand how a computation went, and therefore customize the size of the data chunk that

traverses the graph and the grain of the tasks in order to improve the performance. A study on the latency is given in Section 4.2.1.

3.4.3 Output streaming

As seen in the previous sections, the runtime system relies on a graph representation and on data pipelining to get performance. An important consideration is the data streaming aspect. In order to keep the system busy, the data need to be streamed into the graph.

A direct consequence of the streaming aspect is that it enables one to have a partial final result much sooner than the complete result. If we chain multiple graphs, it is possible to start the computation of the next graph in the sequence much earlier than waiting for the full result. Even if the graph is not linked to another graph, but the end-user gathers in some other way the partial output data, it is possible for him or her to exploit these results without waiting the full computation.

It is based on this behavior that we have started a study on a linear algebra library [Kroiz et al., 2021] to compare Hedgehog implementation and BLAS/LAPACK implementation of matrix multiplication and LU decomposition with partial pivoting. We present our results in Section 4.2.

3.4.4 Scheduler-free execution

The fundamental difference between Hedgehog and other graph-based libraries (Cf. Section 1.7.2) is thread management.

In Hedgehog, some nodes conduct an execution and live on a thread. We associate the threads with these nodes at graph construction and until graph destruction. There is no thread management, there is no scheduler nor additional algorithm to manage the association between the threads and the nodes. We only rely on the OS scheduler to manage the threads. In our experiments, we did not need to customize the priority (nice values) of the threads or the processor affinity.

In terms of performance, a thread is not active all the time. If the node associated with a thread has no input data and this thread is not processing data, it will be in a waiting state. We will cover the node operation with the different calls and the different thread status in Section 3.5.

3.4.5 Conclusion

Hedgehog relies on a simple and explicit model. Computation needs to be represented with a data-flow graph where the nodes are parts of the targeted algorithm. It relies on data pipelining to maximize the overall usage of hardware and overlap communication and different kinds of computations.

The threading model is one-of-a-kind, where each node that does a computation is associated to a thread from the beginning of the graph's execution, up to the graph's termination. Other than that, the threads are managed by the OS scheduler, without any hint provided (nor only priority or affinity).

3.5 Separations of concerns

One cornerstone design decision Hedgehog took from HTGS is the separation of concerns. The main idea is to have a specific object per usage. In this section, we cover the different objects available to the end-user to compose its computation.

3.5.1 Computational task

The main abstraction for doing computations is represented by the `AbstractTask` class (cf. Section 3.6). For the sake of simplicity, we refer to it as a *task*. A task in Hedgehog is an abstraction where a computational kernel is implemented. We built it to represent a compute-intensive section of the algorithm with all the additions used by Hedgehog to manage and execute the kernel (i.e., input and output management, the copies in the case of multi-threaded task, memory management etc.).

Multi-threaded tasks

A task is by default attached to a thread. To improve its performance, it is possible to copy it and attach each copy to a new thread. This copy mechanism is customized by overloading the `copy` method. It is invoked by the Hedgehog runtime system automatically when constructing the graph and instantiating its nodes. By default the method return a `nullptr`. This default value is used to detect if the end-user has overloaded the method or not, when the library needs it, and also allows us to not force the user to overload the method when he or she is using tasks with only one thread.

Figure 3.5 shows how the input queues are handled with multiple copies of a task. A task with multiple inputs has one queue per input type shown in the upper sub-figure. The bottom sub-figure shows the configuration for a multi-thread task with multiple inputs. In this example, a task is copied $n-1$ times, for a total of n instances, and they all share the same queues.

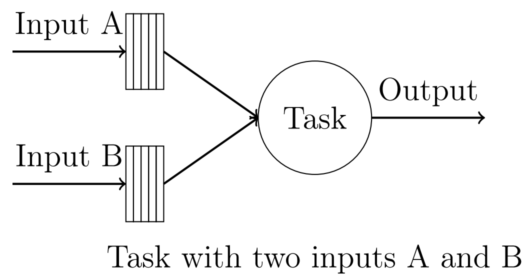
The connection between the queues and the copies is managed by Hedgehog. The queues are thread-safe; they are secured with mutexes that will lock all inputs for a task when one tries to access them. The number of inputs and copies of a task have an impact on the latency as shown in Section 4.2.1

Task runtime

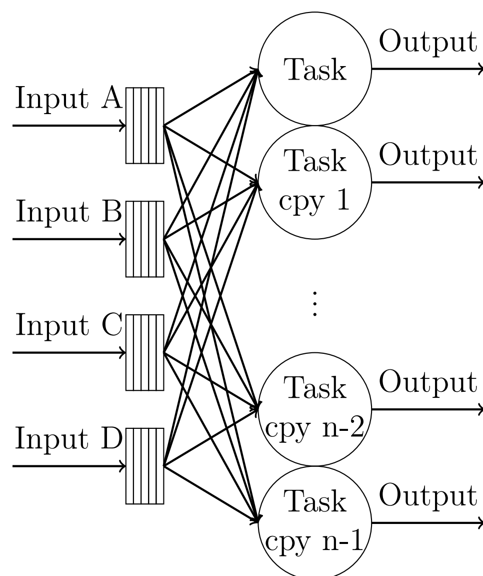
The task execution is a multi-step process (cf. Figure 3.6), in which some steps can be customized by the end-user to represent the computation needed. In this diagram, the blue shapes are all the steps that are customizable by the end-user.

The first step is an *initialization* step. It is done once when the thread is associated with the task, before it starts running. It is customizable by overloading the `preRun` method. The task runtime follows then a cycle that repeats executing the kernel on every piece of data that is received.

This cycle starts by determining if the task must end. By default, a task ends if there are no more predecessor nodes connected to it and if there is no more data available in the input queues. If the task does not end, it enters in a waiting state unless there is data available in one of its input queues. If no data is available,



Task with two inputs A and B



Multi-threaded task with four inputs A, B, C, and D

Figure 3.5: Hedgehog Multi-threaded task

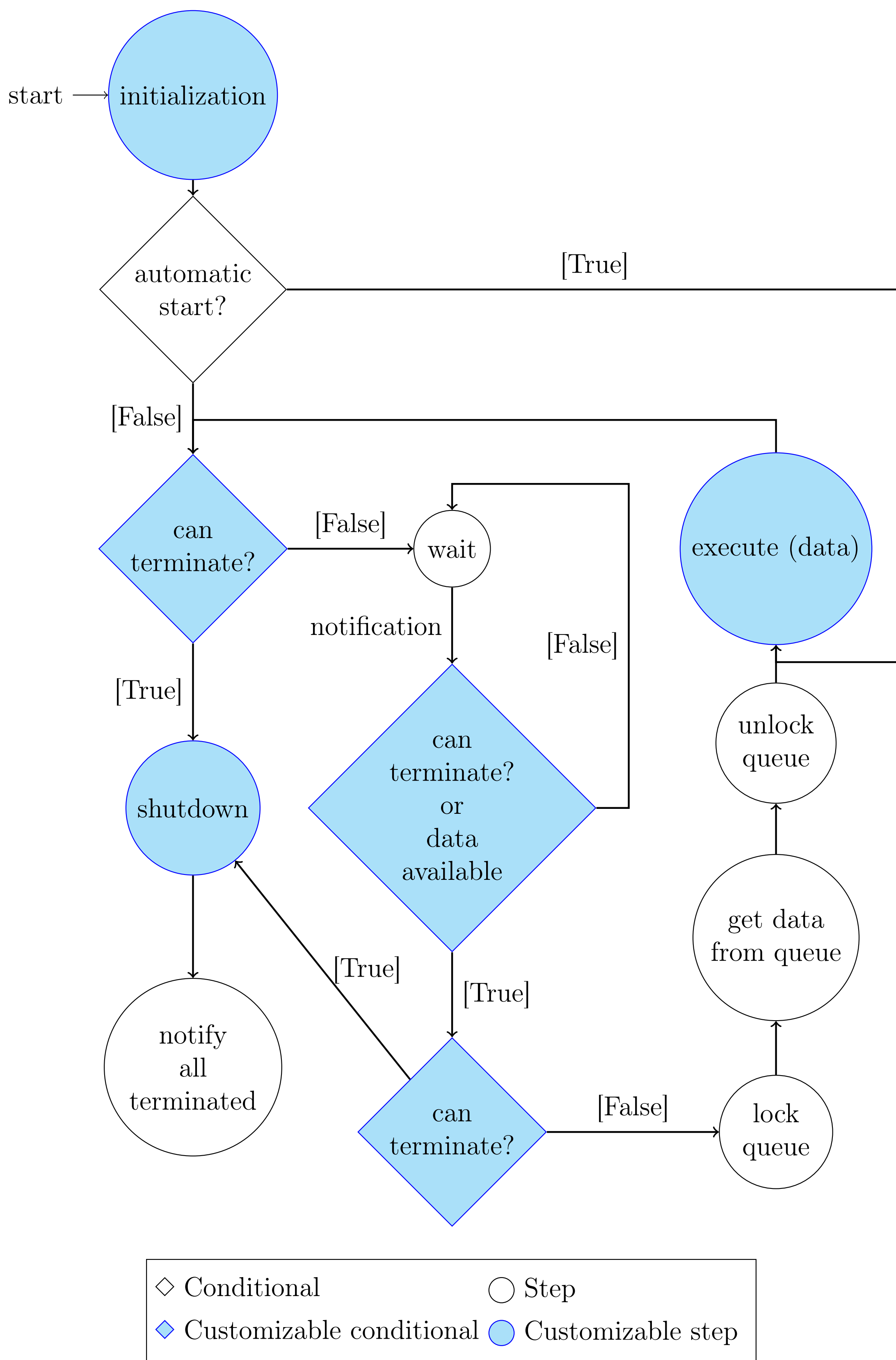


Figure 3.6: Hedgehog task activity diagram

the thread is blocked with an `std::condition_variable` [6] until an antecedent node sends data to the task and notifies it. When data arrive, the task wakes up, checks again if it needs to terminate, and if not, it will process its different inputs. Sequentially for all its input types, the task will lock a queue, get an element if the queue is not empty, unlock the queue, and run the `execute` method with the data popped from the queue. When the execution returns, the task tests again if the task must end, finishing the task runtime cycle.

The default condition to stop a task works well for graphs without directed cycles. The termination condition for a task is therefore customizable to avoid deadlocks. For instance, one simple case of directed cycle is with two interconnected tasks, A and B, having an edge from A to B and an arc from B to A. In this case, without any additional mechanism, both A and B will never match the condition presented above to stop, as A is an antecedent of B and B is an antecedent of A. This interlocking situation prevents the whole graph from ending. In such a situation, the `canTerminate` method can be overloaded to customize the termination condition of a task and therefore give the end-user the possibility to break the cycle with information from the task.

If the task must end, the task executes once a shutdown method called `postRun`. When a task ends, it notifies its successors and its copies, enabling to end in cascade these tasks if the conditions presented previously are met.

At the task creation, it is possible to set the task as automatic, by setting a flag at construction. An automatic task is a task that will start automatically without getting a piece of data from its input queues when the graph is first executed, but with a `nullptr` sent to its `execute` method. After this first execution, the task continues its usual runtime cycle.

Nvidia GPU tasks

Tasks are meant by default to achieve CPU computations. We designed a task for GPU computations on Nvidia devices represented by the `AbstractCUDATask` class (cf. Section 3.6).

In term of C++ objects, an `AbstractCUDATask` inherits from `AbstractTask`, so they mainly behave the same way. The only differences are in the initialization phase, where we bind the task to a device, enable peer access if needed, and create a stream to the device. In the shutdown phase, we destroy the stream to the device.

The only constraint put in place is that all the tasks in a given graph are bound to the same device. It is possible to do multi-GPU computation by duplicating a graph *via* an execution pipeline (cf. Section 3.5.4).

If the developer needs to add steps in the initialization phase, he/she has at his/her disposal a virtual `initializeCuda` method that is called after the initialization that we described earlier and a virtual `shutdownCuda` method that is called before destroying the stream to the device.

This design is replicable by developers for other device as developers just need to create a specialized task for another device provided the specialized task has the same behavior as a normal task. If the internal behavior changes, it is possible to create other types of nodes. We describe this possibility with our internal architecture in Section 3.6.

[6] `https://en.cppreference.com/w/cpp/thread/condition_variable`

Memory manager

One of the key problems with GPU computation is dealing with memory. To be efficient, we should avoid allocating memory in a device to avoid synchronization, we should stay within the limit of memory available on the device and we should reuse memory already allocated on the device.

These goals are achievable with a *memory manager*. The memory manager in Hedgehog is a tool attachable to any task (`AbstractTask` and any subclass). It is a limited pool of memory that will be accessible by the task attached to it.

When the task is initializing, the attached memory manager's pool is filled with its data. By default, the data are initialized with their default constructor. It is possible to customize the memory manager to initialize the data with a special constructor. The special constructor can create pieces of data suitable for GPU computation (previous HTGS static memory manager). We can use the default constructor as is or with allocation later when used by the task (previous HTGS dynamic memory manager). For both cases, the data can be reinitialized when recycled to the memory manager.

When the task wants to get data out of the memory manager, a piece of data is taken from the pool. If there is no more data available, the task blocks until a piece of data is given back to the memory manager and made available.

A managed memory should know by which manager it is supervised. That is why any memory that is handled by a memory manager should inherit from our `MemoryData` interface. This specific interface is designed following the *curiously recurring template pattern* (CRTP). If we want a class `A` to be a managed memory, we will define it as follows: `class A : public hh::MemoryData<A>`, where `MemoryData<A>` provides the functionalities necessary to be supervised by a memory manager. That way, in any other node or even outside the graph a piece of data can be returned to its memory manager, cleaned, and made available again to the task.

The memory manager helps with the data locality as the pool is filled with data allocated at a specific location: main memory or specific device. We can stay under the memory limit of devices as we can throttle the computation by limiting the number of data in the pool. Device synchronization can also be avoided by reusing the same piece of data, doing so no re-allocation.

3.5.2 Management node

While tasks are meant to achieve computations, we added a special type of node for managing computations. This type of node is represented by the `StateManager` class. Technically a `StateManager` is a task limited to only one thread, demanding a `State` object at construction. A `State` is an object used to express the data synchronization strategy of a set of nodes handled by a state manager.

For example, we used blocks to parallelize the computation in the matrix multiplication algorithm (cf. Section 4.3.1). Matrices A and B are fed into the algorithm (we ignore matrix C for this demonstration). Input tasks will decompose them into blocks, in row fashion for the Matrix A and in column fashion for the Matrix B. The first step of the algorithm will be to multiply these blocks of matrices together. The problem is that we should not multiply all the blocks together, only compatible blocks should be multiplied following their positions in the matrix. A block of matrix A will need to be multiplied multiple times to blocks of B: for a given block of matrix

A at row i, it will be multiplied by all blocks of matrix B at column i. In order to manage the blocks coming from the decomposition tasks, forming the compatible pairs we use a `StateManager` and a `State`. The `State` holds a representation of the grid of blocks, retains them until a compatible pair is possible, forms then the pair of blocks and sends it to the block multiplication task (cf. Diagram 1.2). The code for a block of A is presented in Code 3.5. This code is symmetric for a block of B.

Source Code 3.5: Block management of matrix A for the matrix multiplication algorithm

```
void execute(std::shared_ptr<MatrixBlockData<Type, 'a', Ord>> ptr)
    override {
  // Stores the block of A
  matrixA(ptr);

  // For every compatible block of B
  for(size_t jB = 0; jB < gridWidthRight_; ++jB){

    // Checks if the block of B has already been received
    if(auto bB = matrixB(ptr->colIdx(), jB)){

      // Counts down the amount of time the block of A will be used
      ttlA_[ptr->rowIdx() * gridSharedDimension_ + ptr->colIdx()]
        = ttlA_[ptr->rowIdx() * gridSharedDimension_ +
            ptr->colIdx()] - 1;

      // Deletes the local representation of the block of A if it
          won't be used anymore
      if(ttlA_[ptr->rowIdx() * gridSharedDimension_ + ptr->colIdx()]
          == 0) {
        gridMatrixA_[ptr->rowIdx() * gridSharedDimension_ +
            ptr->colIdx()] = nullptr;
      }

      // Creates the pair of blocks A and B
      auto res = std::make_shared<
        std::pair<std::shared_ptr<MatrixBlockData<Type, 'a', Ord>>,
                  std::shared_ptr<MatrixBlockData<Type, 'b', Ord>> >
      >();

      res->first = ptr;
      res->second = bB;

      // Sends the pair to the next task
      this->push(res);
    }
  }
}
```

In terms of design, a state manager is a special task associated with one thread only, and constructed with a state object. Moreover, the state manager is a Hedgehog node while the state is not. In a graph, the state manager will be connected to another node and will be the only one to interact with the state as shown in Figure 3.7.

A detailed sequence diagram, Figure 3.8, shows the interactions between the state manager and the state. Upon receiving data (call to method `addResult`), a state manager will automatically execute its own `execute` method as presented in Figure 3.6 (a state manager is a task). Its `execute` method will lock the state with a mutex and call the state's `execute` method. Then, the user code is executed to manage the computation's state. In contrast with a normal task, the state will not directly send data to the output node, it will rather store them in an internal vector.

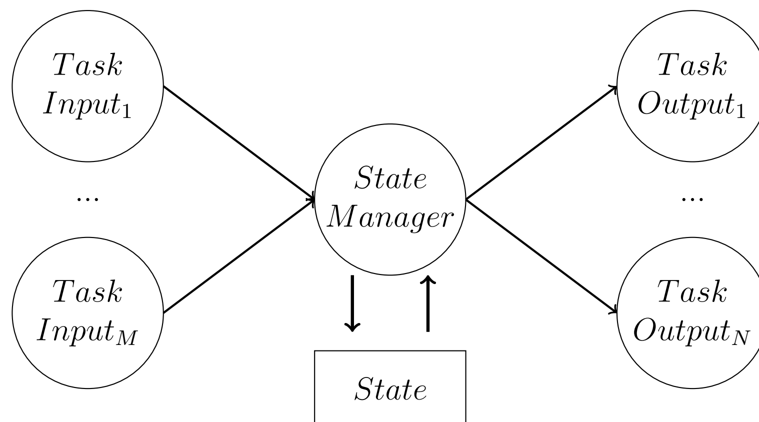


Figure 3.7: Relation between tasks and a state manager

When the execution is done and the `execute` method returns, the state manager will empty the queue and broadcast every piece of output data to the following node (call to method `addResult`). When this is done, it unlocks the state.

The state is thus thread-safe, and therefore can be shared among different state managers. These will share the same view of the computation because they share the same state, but they will have their own predecessor and successor nodes. This is particularly useful if, at different points of the graph, we need to have the same view of the data received, or if data need to be shared between different graphs. This second option is useful because a graph in Hedgehog can be bound to a GPU, and in order to achieve multi-GPU computations, a state can be shared and therefore data are shared.

The counterpart is that the state is locked by a state manager. The computation in a state should be as minimal as possible to avoid any bottlenecks, especially if the state is shared, contrary to a task that is made for heavy computations.

3.5.3 Graph

A graph in Hedgehog is a node that bundles a coherent and organized (tasks are linked together) set of tasks. It is meant to be the main interface that a user faces for building algorithms. Once the nodes are instantiated, they are placed into the graph. There are three main API methods to build the graph: `input`, `output`, and `addEdge`.

Method `input(x)` will set Node `x` as an input of the graph, which means that when a piece of data is sent to the graph it is transmitted to these input nodes. A node can only be set as an input of a graph if it shares at least one input type with the graph. Method `output(x)` will set Node `x` as an output of the graph, which means that when a piece of data is produced by an output node, it will be accessible as a result out of the graph. We can only set a node as an output of a graph if it has the same output as the graph. It is possible to set more than one node as input or output of the graph.

Method `addEdge(x,y)` will connect two nodes `x` and `y`. An antecedent node `x` can only be linked to a successor node `y` if the output type of Node `x` is one of the input types of Node `y`. When a piece of data is produced by a node, it will be automatically sent to its connected successor nodes; this occurs when the `addResult` method is invoked by the producing node. Internally, there is no explicit representation of

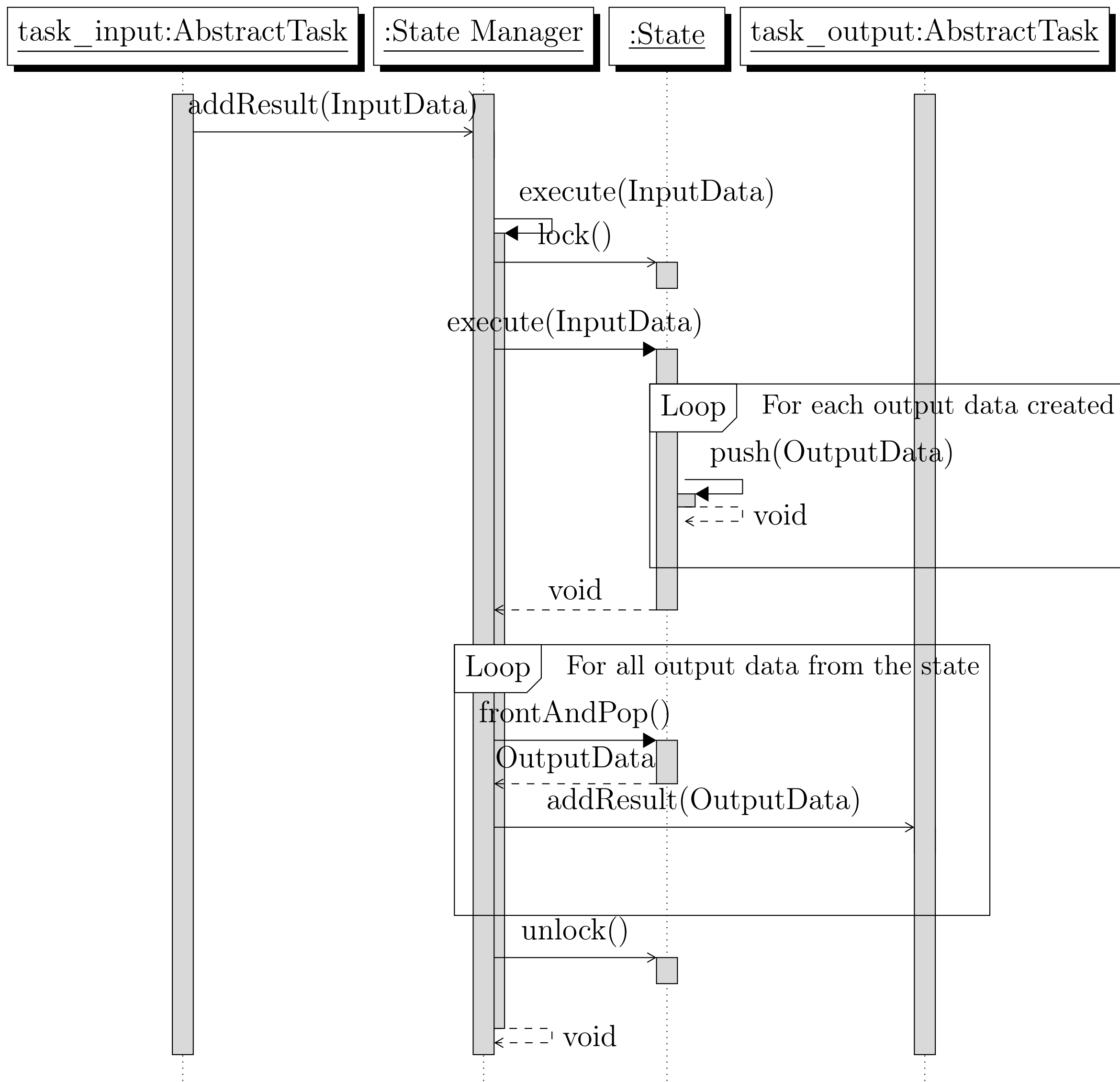


Figure 3.8: UML sequence diagram of interactions between a state manager and a state

an edge by an object, each node has a list of its antecedent nodes and a list of its successor nodes. The connection rules are checked for each edge at compile-time (cf. Section 5.1). In our design, all nodes bound to a GPU inside the same graph must be bound to the same GPU.

We have attached two important tools to the graph that an end-user can customize: (1) the printer and (2) the scheduler. The printer is a visitor (common design pattern [Gamma et al., 1994]) that traverses the graph and its nodes to gather metrics and render them.

The `DotPrinter` is a class producing a DOT file containing a visual graph representation (cf. Section 4.1.1).

The scheduler is the abstraction to manage the graph's threads. This is a major difference from other graph-based libraries, as the Hedgehog `DefaultScheduler` object is used to only create all the threads, using `std::thread` API, of the internal nodes recursively at graph creation. The graph joins all the threads when the graph's `waitForTermination()` method is called by the end-user. If a user wants to use another thread model or create a custom scheduler, he or she can do so by implementing a scheduler from our abstraction `AbstractScheduler` that will be provided to the graph at construction. This abstraction existed already in HTGS and has been amenable to further studies and published [Wu et al., 2021].

Graph composition

As discussed previously, a graph represents a complex computation made of a set of nodes, and because a graph is a node, it is possible to compose graphs together. It can therefore be connected to other nodes. When a user connects an antecedent node to a graph, this node is internally connected to all the graph's input nodes. When an internal graph is connected to a successor node, the graph's output nodes are connected to the successor node. At the moment a graph is connected inside another graph it can no longer be modified.

This feature has multiple advantages. First, the composition helps with the abstraction; it is possible to decompose a complex algorithm into a set of independent smaller graphs and create them independently. This methodology helps thinking about large computations by breaking them into smaller ones.

Second, the composition helps with code sharing. A graph can represent a full computation. Another developer can import this computation into his or her graph as a black box node. The only thing that matters when building the graph is the explicit output and input types to link it to other nodes. When visualizing the graph computation, the structure will be exposed showing any bottleneck.

Third, the composition helps with multiple GPU computations. It is then possible to have a computation per GPU by building a graph for each device. However, if all GPUs should execute the same piece of code on separate data, an execution pipeline can be used (cf. Section 3.5.4).

3.5.4 Execution pipeline

An execution pipeline is a node that is used to duplicate a graph for multi-GPU computations. It is constructed by specifying the following parameters: the graph, the number of times the graph needs to be duplicated, and the devices' IDs. To duplicate a graph, all internal graph nodes are recursively copied. Then, each graph

is associated with a device id, and all its `CUDATask` are bound to the GPU with this device id.

In order to fully define an execution pipeline, the user needs to implement a decomposition strategy. When data are sent to an execution pipeline, they are transferred through the decomposition strategy to the desired graph represented by its identifier (`graphId`). To define this, the end-user needs to overload the pure virtual method `bool sendToGraph(std::shared_ptr<GraphInput> &data, size_t const &graphId)`. The method must return `true` if the piece of data should be sent to the graph of identifier `graphId`.

3.6 Software architecture

In the previous section, we presented the different parts of the Hedgehog library with a focus on their usage by an end-user. In this section, we present a more technical point of view of the library with highlights on its architecture. The library is decomposed into two main parts, the API (elements used by the end-user) and the core (internal representation, mainly of nodes), that we describe here. This separation is based on two behavioral design patterns (*template method* and *strategy* [Gamma et al., 1994]).

In this section, we present UML class diagrams. They are a simplified version of the actual classes; we have removed non-useful methods or attributes for the sake of clarity while keeping a representation of the design choices. Moreover, we have suppressed mention of the standard library name space (`std::`) and have renamed the `shared_ptr` "sh" to limit the dimension of the diagrams.

3.6.1 API

The API presents all programming elements (classes, functions...) that a usual user will use to build and execute a Hedgehog graph.

We implemented all the API calls useful to end-users with a focus on simplicity. If the end-user uses an IDE, it should mainly show only the methods that are useful to define and initiate the computations.

Base classes are provided in the `hh::behavior` namespace, they hide implementation details that have been isolated in the core part of the library. They expose methods that the user may want to use or need to implement. Notably he/she can inherit from these classes to adapt their behavior to his/her needs and implement the kernel codes, by redefining some of their methods. The design to adapt the base classes relies on the *template method* design pattern.

The cores represent all the internals required by the library to manage the nodes from data I/O to memory management or computational kernel invocation. We define the relation between the API and the core in Section 3.6.2.

Graph

Figure 3.9 presents the API that the end-user must use to design a Hedgehog graph. After designing subclasses of `Node` (cf. next subsection) to define specific computational nodes, the user instantiates objects of those classes before adding them

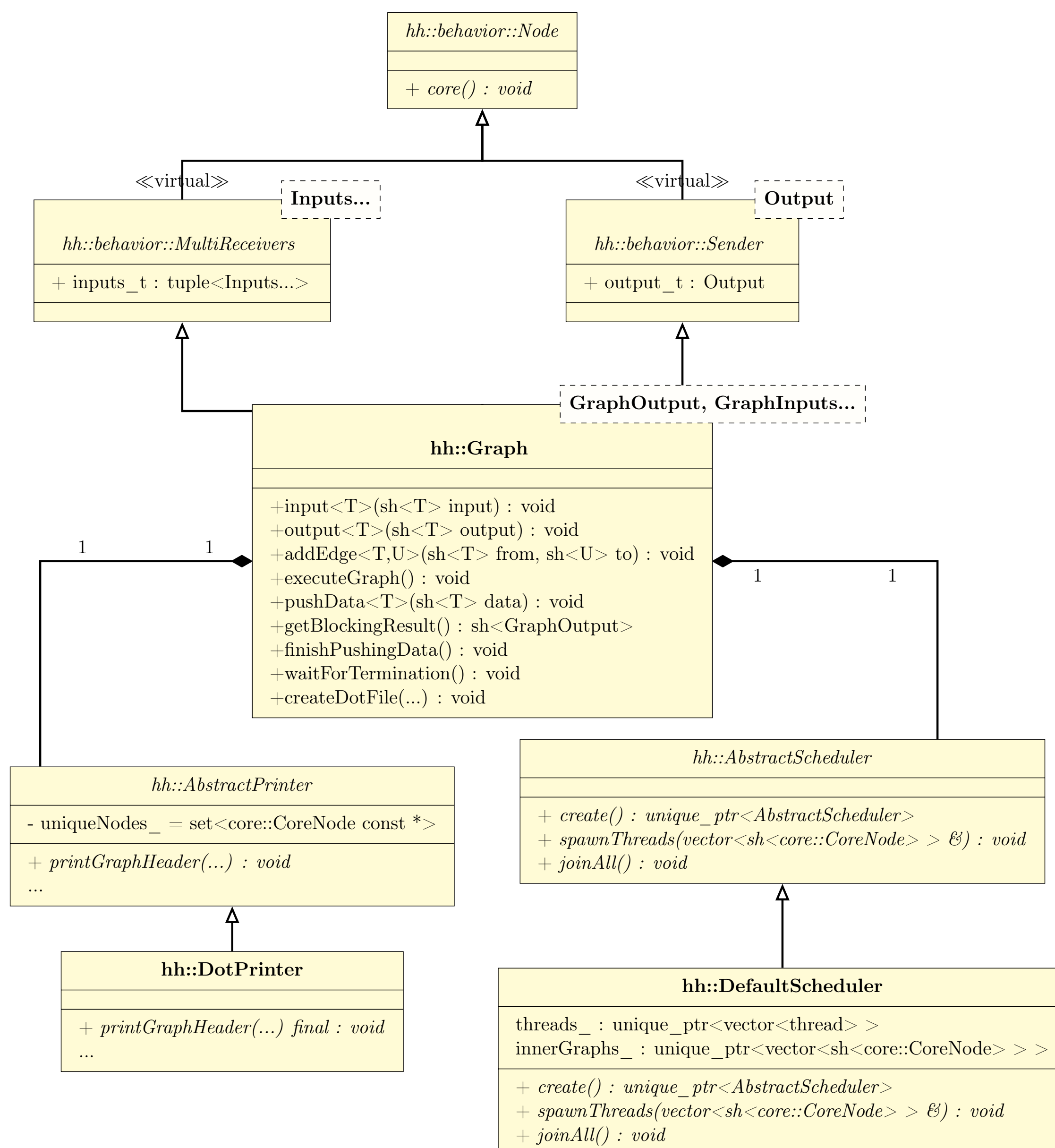


Figure 3.9: UML class diagram of the graph modeling [6]

to an object of class `Graph`, using methods `input`, `output` or `addEdge` (as presented in Section 3.5.3), thereby describing the whole desired computational process.

Classes `MultiReceivers` and `Sender`, inheriting from `Node`, have been defined to represent respectively a node that can receive inputs of multiple types and can send an output. By specialization, the graph is, (1) a node because it has a `core`, (2) multi-receivers because it can receive data of multiple types, and (3) a sender because it can send data of one type. This induces a diamond inheritance between these classes that is used for type cast in the different cores.

The figure presents the building methods `input`, `output`, and `addEdge` and the execution methods `executeGraph`, `pushData`, `getBlockingResult`, `finishPushingData`, `WaitForTermination`, and `createDotFile`. `executeGraph` does the task copies and attaches the threads. `pushData` allows sending data to the graph. `getBlockingResult`

[6] In UML, a class or method name in italic font means *abstract*.

allows getting data from the graph. finishPushingData signals to the graph that no more input data will be sent to it; this signal enables the shutdown in cascade of the graph's nodes when they have finished processing their input data (with the default behavior).

When connecting a node into the graph, type checking has to take place. Method input for example is only defined (compile-time checking that validates the overload of the method) if the argument object T is a Hedgehog node that can receive data (i.e., of type MultiReceivers) and is compatible with the graph (i.e., its set of input types is a subset of the graph's set of input types). In this method, we only know that T is a MultiReceivers, but do not know about its real type: a graph, a task, or another compatible class defined by the user. Moreover, we have the same situation (1) between the Sender class and the output method and (2) between the MultiReceivers and Sender classes and the addEdge method.

A graph uses an AbstractPrinter class to gather metrics from its nodes and visualizes them, via the DotPrinter methods available in the library (cf. Section 4.1.1). As presented in Section 3.5.3, the graph uses an AbstractScheduler class to manage its threads (the DefaultScheduler only creates the threads and attaches them to tasks).

Task

A task (of base abstract class AbstractTask), like a graph, is a node that can receive and send data, but it can also perform some computation, as shown by the UML class diagram of Figure 3.10. The task inherits from the *execute* interface for each of its input types to define a specific computation for each one, while the graph does not.

This computation is made available through the Execute abstract class, which class defines a method execute for an input type as argument. This means that because the AbstractTask class can accept multiple input types (it has TaskInputs as a template parameter pack), it inherits multiple times from class Execute<T> with different values for T (each one of template parameter pack TaskInputs).

For example, the instantiation AbstractTask<O, A, B, C> inherits three times from the Execute<T> class with T = A, T = B, and T = C. The pure virtual method execute will need to be defined three times, one per input, when the end-user will define its own task class inheriting from AbstractTask<O, A, B, C>:

- void execute(std::shared_ptr<A>),
- void execute(std::shared_ptr<B>),
- void execute(std::shared_ptr<C>).

This design also adds some type safety because it avoids constructing multiple inheritances of the same type, as without virtual inheritance, a class can not inherit multiple times from the same class. We therefore avoid this kind of construction: AbstractTask<O, A, A>. A check is also done at compile-time to give a proper error message and avoid compiler automatic messages.

The methods of AbstractTask define the steps of the execution cycle of a task (Section 3.5.1) : initialize, shutdown, canTerminate, and copy. The other methods help the end-user connect and use a memory manager.

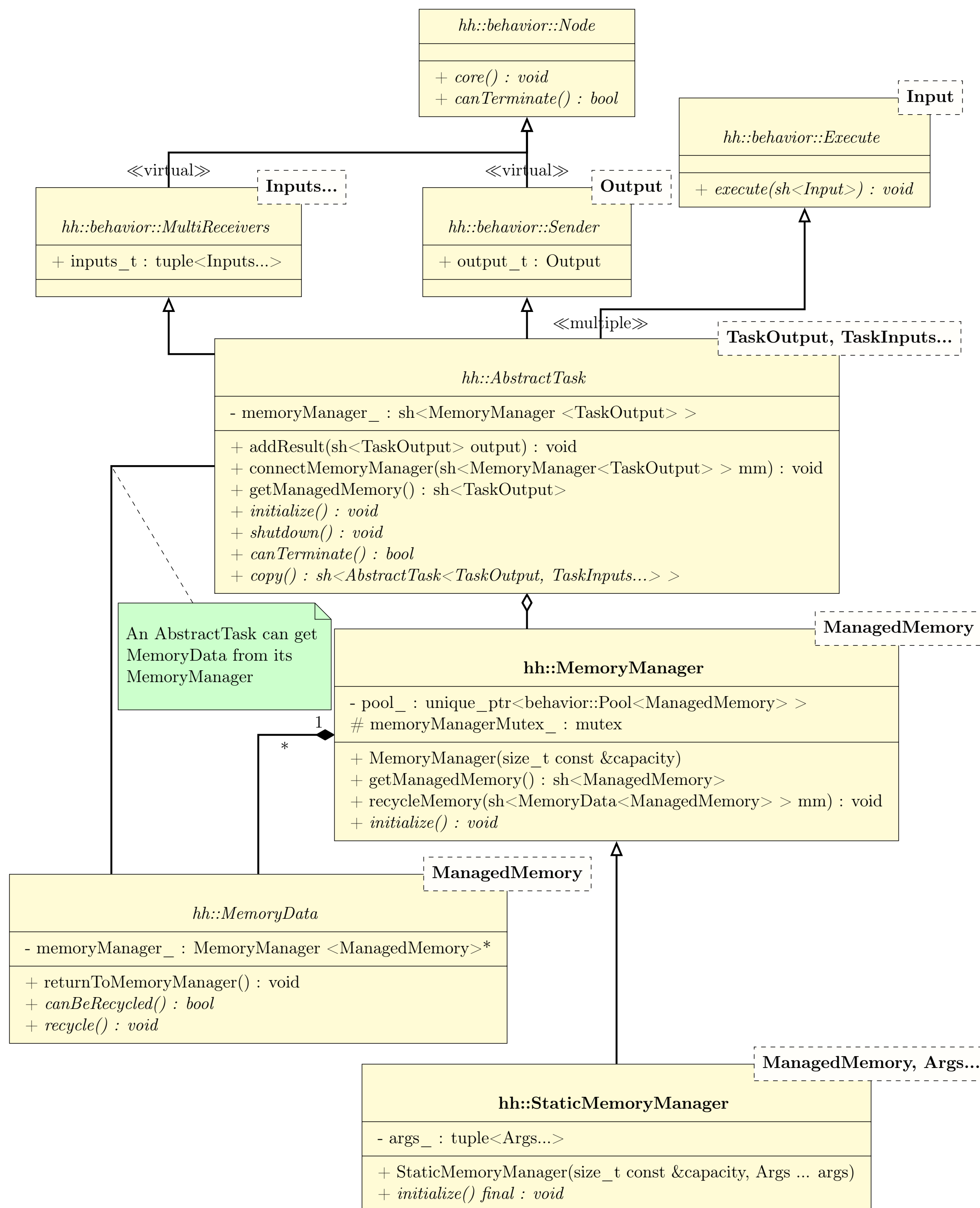


Figure 3.10: UML class diagram of the task modeling [6]

A memory manager can be attached to a task to throttle the memory produced by this task from this point in the graph (cf. Section 3.5.1). The task can then get memory out of the manager by calling the method `getManagedMemory`.

The base class `MemoryManager` populates its pool at construction with *default constructed* objects of type `MemoryData`. The specialization `StaticMemoryManager` populates its pool with `MemoryData` objects constructed with the arguments `args` received and transferred by the constructor of the manager. For example, a memory manager of type `MemoryManager<A>` creates a pool with `A` instances constructed with constructor `A()`. The memory manager `StaticMemoryManager<A, int, float, Z>` creates a pool with `A` instances constructed with constructor `A(int, float, Z)`.

Not all types can be managed by a memory manager; the type needs to implement the interface `MemoryData`. This is different from HTGS where all types flowing in the graph needed to implement the `IData` interface; in Hedgehog, only the types managed by the memory manager need to implement a specific interface. This is needed because memory data must know which memory manager so it can be produced and returned to it from anywhere in the graph or outside of the graph. The return to the memory manager by calling method `returnToMemoryManager` will start a recycle mechanism. The idea is to clean a piece of data in order to be reused afterwards. The data is cleaned with the method `recycle`. Because a managed data can be used by many nodes, it may need to be returned multiple times before being recycled. This is the role of the `canBeRecycled` method; it determines if the data needs to be recycled and made available to the pool now or in a later return.

Specialized tasks

Hedgehog defines other specialized tasks from the `AbstractTask` class as shown in Figure 3.11.

The `AbstractCUDATask` class defines a task specialized for CUDA computations on Nvidia devices. To achieve a computation on a GPU, we need to associate the task thread to the device. This is done by creating an Nvidia stream on this thread to the GPU during the task's initialization. During this phase, peer access is also enabled if needed; the stream is destroyed during the shutdown phase. Both initialization and shutdown phases for CUDA tasks follow a specific pattern defined in the final redefinition of methods `initialize` and `shutdown` inherited from `AbstractTask`. The customization of these two steps is still partially/locally possible for the end-user by redefining methods `initializeCuda` and `shutdownCuda`.

The state manager is another special type of task. However, it cannot directly inherit from `AbstractTask` because of the `execute` method that needs to be defined for each input type (from the `StateInputs` pack) to manage and perform the execution on the state as presented in Section 3.5.2, which is done in the `StateManagerExecuteDefinition` class for a given input type `StateInput`. Similarly, the `AbstractTask` inherits multiple times from the `Execute` class, the `StateManager` inherits multiple times from `StateManagerExecuteDefinition` for each type in `StateInputs` pack.

The `AbstractState` inherits from the `Execute` class (the same base class as `AbstractTask`), once per input type in `StateInputs` pack, to define its computation kernels. It also has a private list of output data and a mutex for its execution and management by multiple state managers.

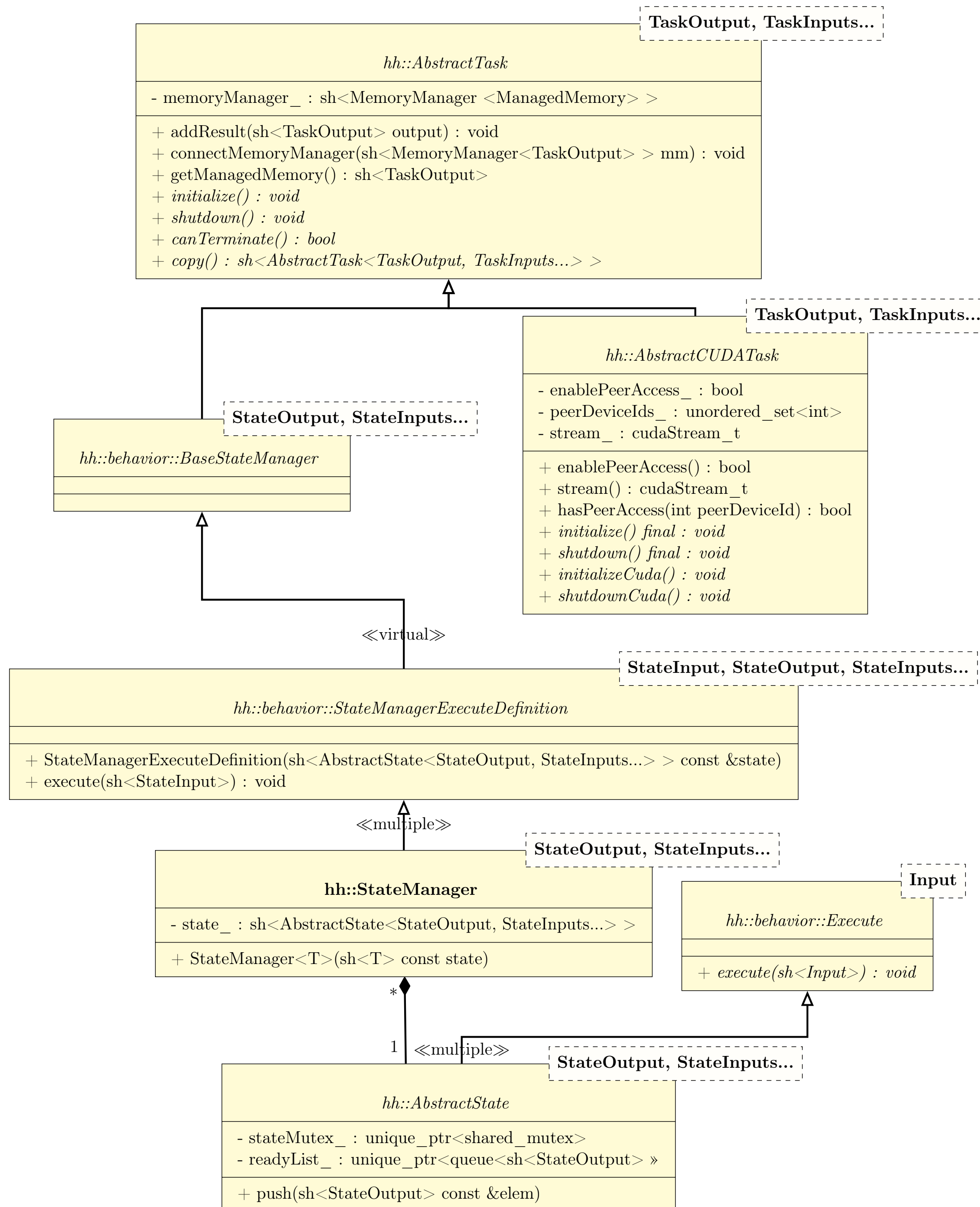


Figure 3.11: UML class diagram of specialized tasks modeling [6]

Execution pipeline

The execution pipeline is a special node type that acts as a wrapper around a graph to implement multi-GPU computation, as shown in Figure 3.12.

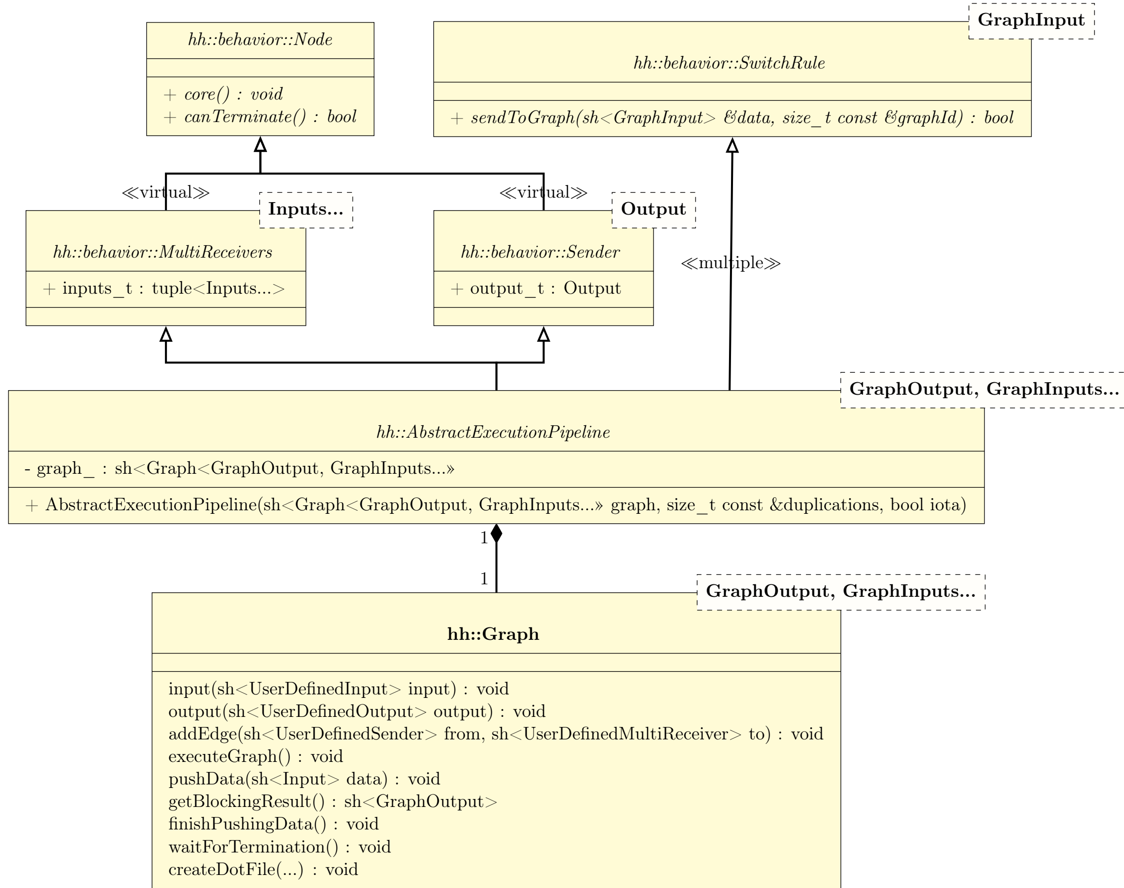


Figure 3.12: UML class diagram of the execution pipeline modeling [6]

Its goal is to clone a graph `duplications` -1 times and associate each graph (including the original one) to different GPUs. The execution pipeline accepts the same input types and output type as the graph it manages and can be connected to any other graph as a single node. The specificity of the execution pipeline is the multiple inheritance from `SwitchRule` for each input type to get an overload of method `sendToGraph` for each input type. Each time a piece of data is sent to the execution pipeline, the execution pipeline calls the corresponding `sendToGraph` method (type deduction selects automatically the suitable overload). This method returns `true` if the piece of data must be sent to the graph of identifier `graphId`, so it is tested for every graph ID until the good one is found.

The graph copy mechanism is based on the *visitor* pattern [Gamma et al., 1994] that will iterate over all the nodes of the graph to clone them. If the node is a graph, then all of its nodes will also be cloned recursively.

If the node is a task or a state manager, it will be cloned by calling the `copy` method. By default, a state manager will share the state it manages with its copies. In the case of multi-GPU computations, this behavior allows managing computations between GPUs, as it is possible to know in a thread-safe manner what piece of data is available in all GPUs. If peer access is enabled, all the GPUs have access to the memory of the other GPUs. If not, a piece of data can be sent to a task in order

to transfer it from a device to where it is needed, or use unified memory that will handle this automatically with a pre-fetching task before the data is needed to avoid page faults.

Smart pointers

In order to offer the simplest and safest API possible, we chose to use smart pointers to manage the data transferred between nodes of a graph. Smart pointers under the form of `shared_ptr` are an implementation of a *Resource acquisition is initialization* (RAII) technique [Stroustrup, 1997]. This is a paradigm change in the language for managing memory in C++. Common memory management is based on how the piece of memory is used, from when it is created with `new` to when it is destructed with `delete`. This new technique is not focused on a manually managed life cycle of the object but on the ownership of the data. The base mechanism behind a `shared_ptr` is that the object is automatically destructed when the last owner loses sight of the data.

This mechanism has three direct impacts on the usage of the library and the API. The first one is the security of the library. The graph only accepts `shared_ptr` of the nodes when it is constructed and keeps a copy of them. This means that when the graph is destructed and if the user has not kept a reference to the graph's nodes, all registered nodes are safely destructed without calling explicitly their destructors. If the user has kept references, they will be deallocated at program termination.

The second impact is about data transfer between nodes. Each piece of data is wrapped into a `shared_ptr`, this lowers the latency because we do not copy the data itself but only a [smart-]pointer. Indirectly, it also enables the broadcast mechanism, the same piece of data can be sent and used by different nodes at the same time.

The third impact is the additional management needed by the API level to check the acceptance of some nodes. For example, an input node of a graph needs to be a receiver, a node as output needs to be a sender, and when connecting an edge between two nodes, the first node needs to be a sender and the second needs to be a receiver. These conditions are not sufficient but are the first checking step that is needed at compile-time. We have implemented that through inheritance, these nodes inherit from two interfaces to ensure these connections: `Sender` and `MultiReceivers`. If we had used only raw pointers, we could have written for a graph instance `Graph<Output, Input1, Input2>` the method `output` as `void output(Sender<Output> * nodeOutput)`. The checking would thus be done automatically by the compiler, thanks to the inheritance principle: only the pointers to objects of class `Sender<Output>` or any subclass are accepted by method `output`.

But, this type checking mechanism does not handle smart pointers as simply as it handles raw pointers. If class `A` inherits from class `B`, a pointer of `A` (`A*`) can be stored into a pointer of `B` (`B*`). However, a smart pointer of `A` (`std::shared_ptr<A>`) can *not* be stored into a smart pointer of `B` (`std::shared_ptr<B>`).

To achieve the same behavior with smart pointers, we introduce two versions of the Hedgehog library, the first one bases this inheritance check on template metaprogramming techniques and is compatible with the C++ 17 norm (cf. Code 3.6); the second version uses concepts and is compatible with the C++ 20 norm (cf. Code 3.7).

Source Code 3.6: Smart pointer checking on Hedgehog API v.1 (C++ 17)

```
template<
  class UserDefinedSender,
  class IsSender = typename std::enable_if_t<
    std::is_base_of_v<
      behavior::Sender<GraphOutput>, UserDefinedSender
    >
  >
>
void output(std::shared_ptr<UserDefinedSender> output) {/*...*/}
```

Source Code 3.7: Smart pointer usage on Hedgehog API v.2 (C++ 20)

```
template<typename DynamicHedgehogNode>
concept HedgehogNode = std::is_base_of_v<
  hh::behavior::Node,
  DynamicHedgehogNode>;

template<typename DynamicHedgehogNode>
concept HedgehogMultiReceiver = HedgehogNode<DynamicHedgehogNode>
  && std::is_base_of_v<
    typename hh::helper::HelperMultiReceiversType<
      typename DynamicHedgehogNode::inputs_t>::type,
      DynamicHedgehogNode
    >;

template<typename DynamicHedgehogNode>
concept HedgehogSender = HedgehogNode<DynamicHedgehogNode>
  && std::is_base_of_v<
    hh::behavior::Sender<
      typename DynamicHedgehogNode::output_t
    >,
    DynamicHedgehogNode
  >;

template<typename DynamicHedgehogNode>
concept HedgehogConnectableNode = HedgehogNode<DynamicHedgehogNode>
  && HedgehogMultiReceiver<DynamicHedgehogNode>
  && HedgehogSender<DynamicHedgehogNode>;

template<HedgehogConnectableNode UserDefinedOutput>
  void output(std::shared_ptr<UserDefinedOutput> output){//...}
```

In the C++ 17 version, we need to first extract the type embedded into the smart pointer (here `UserDefinedSender`), and then check if this type has the good inheritance property. This is achieved with trait `std::is_base_of_v` that returns `true` if type ((behavior::Sender<GraphOutput>)) is the base class of type (`UserDefinedSender`). With this mechanism, we can filter which type is acceptable by keeping the method for this type in the overload resolution thanks to the `enable_if` construct 2.21 (cf. Section 2.3.6).

In C++ 20, we use concepts (cf. Section 2.3.8). We defined a base concept of what is a Hedgehog node (`HedgehogNode`, lines 1-4), and build from this concept what is a node that 1) can send data (concept `HedgehogSender`, lines 14-21), 2) can receive data (concept `HedgehogMultiReceiver`, lines 6-12), and 3) is *connectable* so it can receive and send data (`HedgehogConnectableNode`). `HedgehogMultiReceiver` uses a meta-function `HelperMultiReceiverType` that constructs a `MultiReceivers<Inputs...>` type from its input types (template parameter pack `Inputs`). We can then use these concept definitions to add checking in our API (lines 28-29). We could have use

`HedgehogSender` instead of `HedgehogConnectableNode` to be close of the previous API, but we have preferred specifying a more restrictive constraint.

We have only presented the checking of method `output`, but the API has been produced the same way for methods `input` and `addEdge`.

3.6.2 Core

Now that the structure of the API has been introduced (what is offered/accessible to the user), we present in this section the cores, the internal representation of the nodes.

Definition and relation with the API

Each of the API nodes is associated with a core, which defines the logic of a node. The nodes and the cores are interdependent: the nodes define the user code like the computation on the tasks (e.g., the block multiplication in the matrix multiplication); the cores define the logic of the nodes (e.g., the task runtime cycle presented in Figure 3.6) and all the internals needed to conduct the computation. Therefore, if multiple nodes behave in the same manner, they will share the same core type. In Table 3.1, we see that `AbstractTask`, `AbstractCudaTask`, and `StateManager`, shares the same type of core `CoreTask`.

Node	**Core**
Graph	CoreGraph
AbstractTask	CoreTask
AbstractCudaTask	CoreTask
StateManager	CoreTask
AbstractExecutionPipeline	CoreDefaultExecutionPipeline

Table 3.1: Relation between the nodes and their cores

Core architecture

The cores are defined with an architecture close to the API. The proximity of both architectures is meant to ease extensibility. However, they are different in numerous ways, the core hierarchy is more complex with more layers. The concept of *sender* and *receiver* that respectively sends and receives data is not precise enough. We need a more complex communication layer. This part of the architecture is presented in Figure 3.13. We decompose this communication into two parts: (1) a data transfer part and (2) a notification part.

For memory, a node in Hedgehog can accept multiple input types and can produce data from one output type only. To implement this, the data transfer part is decomposed into three main interfaces: a sender, a receiver, and multi-receiver. The sender is responsible for transferring data to a receiver, and because a node can accept multiple input data types, a core inherits once from `MultiReceivers` that inherits for each type from `Receiver`. The problem is that the data transfer is not identical for all nodes. Let us consider, for example, the graph and the task kinds of nodes. When a graph receives a piece of data, it needs to transfer it to all of its input nodes. When a task receives a piece of data, it stores it into the input queue corresponding

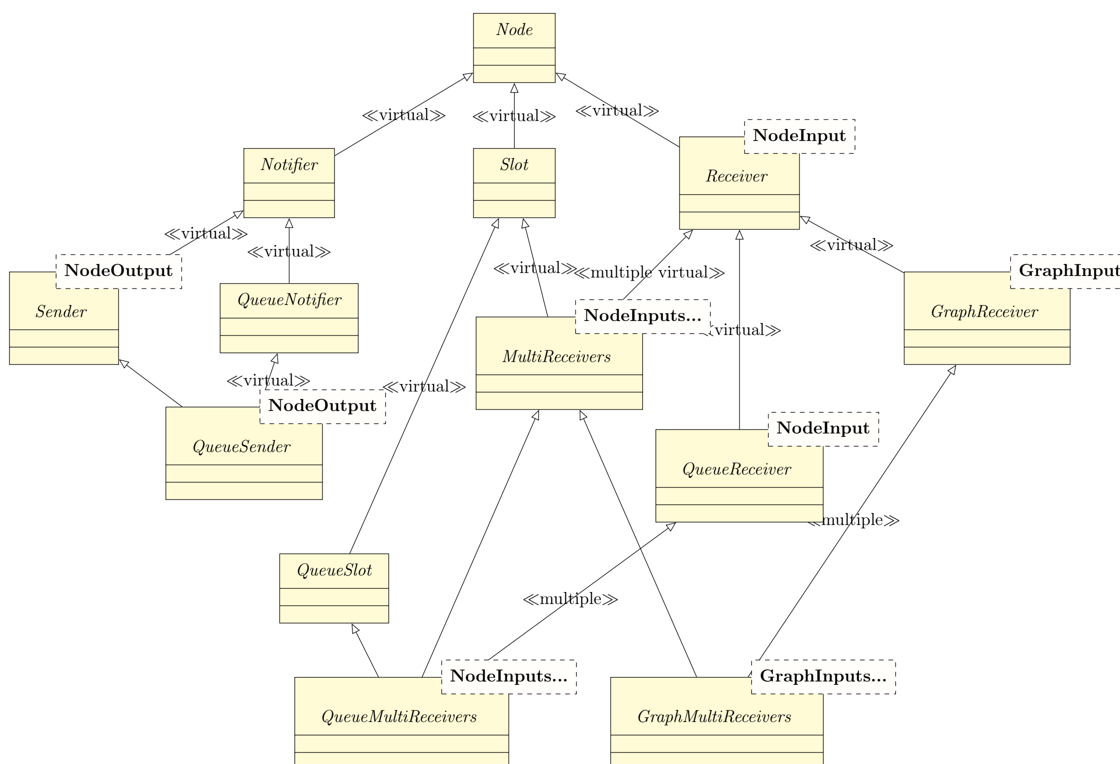


Figure 3.13: Simplified UML class diagram of the communication layer [6]

to the data type. The `QueueMultiReceivers` and the `GraphMultiReceivers` inherit both from `MultiReceivers`. These interfaces are sufficient to implement all data transfers in Hedgehog, but we need to add a notification mechanism to wake up the threads.

This notification mechanism follows a notification/slot architecture. It is composed of a *notifier* that will signal potentially multiple *slots* when an action is done, for example a piece of data has been sent. A notifier is attached to a sender and the slot is attached to a multi-receiver. Then depending on the core that receives the signal, the reaction is different.

We separate the data communication from the notification mechanisms to keep the library modular and because the notification is not only used when a data is sent. For example when a task terminates, it notifies its copies and its successor nodes of its termination to ensure that all the nodes in the graph terminate properly.

For a task, the thread is woken up if it is in a waiting state or does not take care of the notification if the task attached to the thread is already performing some computation as shown in Figure 3.6. When ready to treat new input data, the task locks its multi-receiver and therefore all of its receivers containing each a queue. We chose to have a locking mechanism at the multi-receiver level and not at the receiver level to avoid a heavy locking mechanism at the cost of a coarser grain control. The mechanism to test if a queue is not empty and to pull a piece of data is a minimal set of instructions in a critical section protected by a mutex, lowering the impact of the coarse grain control, and not impacting too much the system latency 4.2.1.

For a graph, the notification mechanism is a bit different, it does not have a thread attached to it so there is nothing to wake up.

We differentiate the graph that the end-user uses to conduct computations (the outer graph) from the possible inner graphs that compose the outer graph. When

an antecedent node is connected to an inner graph, it is in fact directly connected to the input nodes of the inner graph. By symmetry, the output nodes of an inner graph are connected to the successor nodes of the inner graph. The outer graph has a core that is composed of hidden cores for the input and the output, namely the `CoreGraphSource` and the `CoreGraphSink` of the graph.

We added these cores to ease the interactions with the graph in a transparent manner for the end-user, he or she only sees a graph. When a piece of data is sent to a graph, the `CoreGraph` transmits the data to its source, which knows all its input nodes and sends the piece of data to them. When the output tasks produce output data, they are all transferred to the sink that gathers them and waits for the user to pop them.

When a graph is used inside another graph as a node, the source and the sink are not used anymore to favor a direct connection to the inside nodes.

An execution pipeline uses two cores. The first, of class `CoreDefaultExecutionPipeline`, defines how the pipeline behaves such as the `CoreTask` does for the `AbstractTask`. It inherits from the `CoreTask` class, but redefines some behaviors to use a `CoreSwitch` instance. This second core is used to redirect the data flowing to the execution pipeline to the correct graph duplicate.

In Table 3.2, we sum up all the cores with the most precise interfaces they inherit from. The one in italic are the cores that does not have a direct counterpart in the API.

Core	CoreGraphMultiReceivers	CoreQueueMultiReceivers	CoreQueueSender	CoreExecute	CoreTask
CoreGraph	x				
CoreTask		x	x	x	
CoreDefaultExecutionPipeline					x
CoreSwitch			x		
CoreGraphSource			x		
CoreGraphSink		x			

Table 3.2: Inheritance table of the cores and their interfaces

The `CoreExecute` interface is made for calling the user implementation of the `execute` method. Figure 3.14 presents the relationship between the core and the API for an `AbstractTask`. All inheritance is presented, including the `CoreExecute` interface.

3.6.3 Library extensibility

We have already explained that the differentiation between the core and the API is to limit what the end-user sees and can interact with. However, the main reason behind this architecture is the extensibility.

If a developer wants to create a new type of node that behaves internally like a task, he/she can simply inherit from `AbstractTask`, which was done for the `AbstractCUDATask` and the `StateManager` classes. We can also define a new kind of core by inheriting from an existing one to change its behavior. Then if needed, we just have to create a new node that uses this new core.

Finally, if a new behavior needs to be created, for example, a non-locking communication, a developer can create a new sender and new receiver classes by inheriting from `Receiver`, `Sender`, and/or `MultiReceivers` to add this extra feature. The library uses solely the interface of these abstractions (i.e. the communication classes like `Sender`). As such, any subclass that integrates a new behavior by redefining some method should be compatible with the rest of the library.

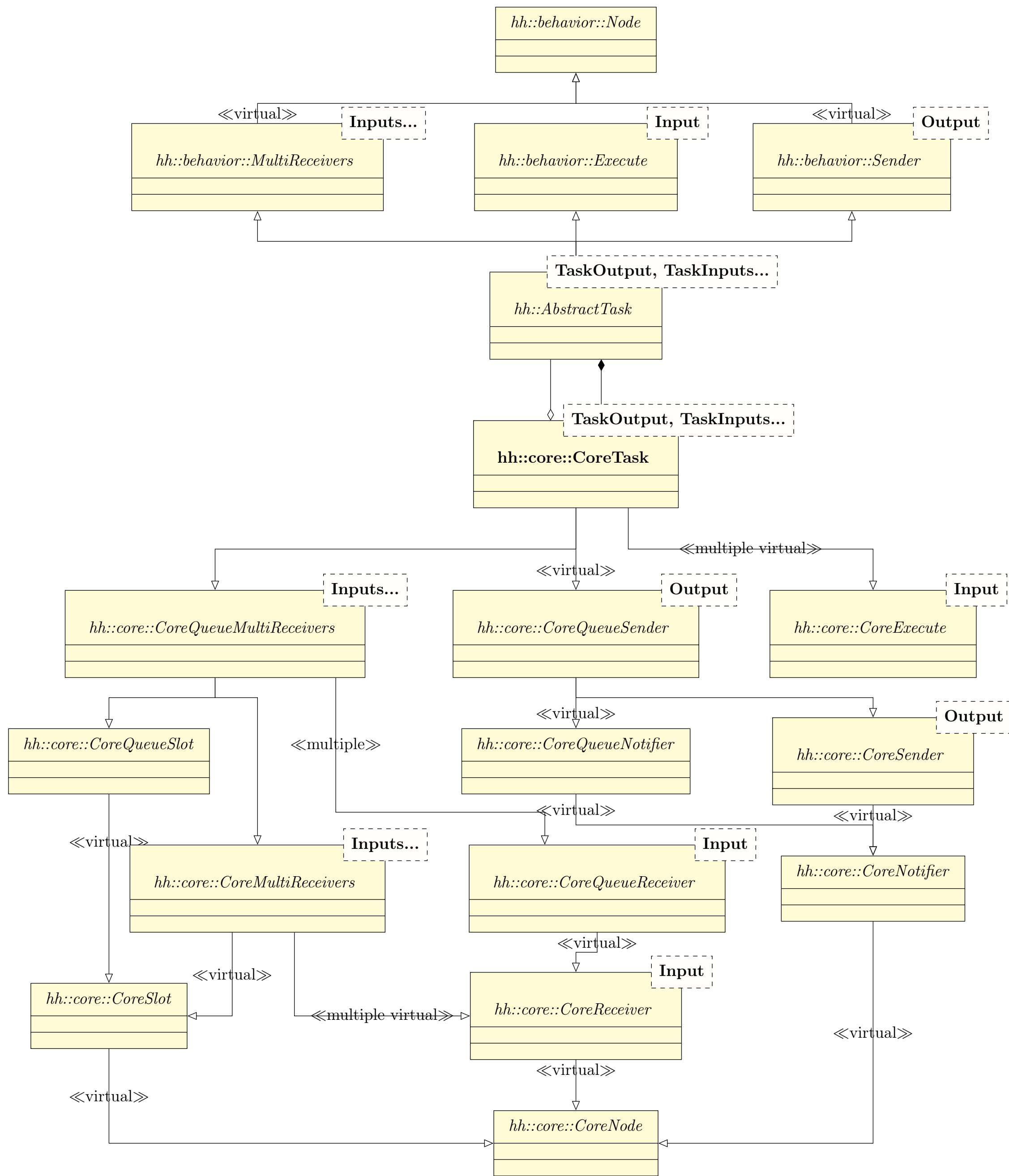


Figure 3.14: Simplified UML class diagram of the relation between the `AbstractTask` and `CoreTask`

3.7 Conclusion

In this chapter, we have presented our heterogeneous coarse-grained parallel runtime system, Hedgehog. Its execution model is based on a data-flow graph that works without any scheduler, which contrasts with many existing task-based libraries.

Hedgehog is based on HTGS, while adding new features. Mainly, the multiple input types and the output broadcast capabilities for any node of the data-flow graph enable new possibilities to create more expressive graphs. Notably, the *bookkeepers* from HTGS have been replaced by *state managers* to allow the generalization of the broadcast mechanism. However, we kept from HTGS its approach for composing graphs, its memory management for limiting the memory in the graph and staying within the limits of GPUs, its graph duplication mechanism in execution pipelines for multi-GPUs computation, and the logic of the API.

Hedgehog proposes a new architecture that separates what should be usable to an end-user, and what he/she can extend to implement domain-specific algorithms, from what is modifiable by advanced users who want to implement new task logic, all this allowing diverse degrees of extensibility. For example, a new node which acts like a classical task, requests only a specialization of a class from our API abstraction. By contrast, a new way to transmit data requires specializing a class from our core abstraction, while being fully compatible with the rest of the library.

The possibility for the user to keep the same model, from its early design to its implementation and execution with Hedgehog, allows him/her to get performance feedback that can be directly understandable. This enables what we call *experimentation for performance*, which we present in the next chapter.

Chapter 4

Experimentation for performance

In this chapter, we present how the approach that we call *experimentation for performance* can be achieved. The principle is to put the developer at the center of all the decisions to get performance. The Hedgehog model is explicit: what is designed by the end-user is what is actually executed. That is what we think to be the first step to get performance, a developer cannot really improve something that he or she does not fully understand.

However, this is not enough to get performance. Tools to give feedback on how the computation has been conducted on a specific hardware are necessary, first for debugging purpose, and then to identify possible ways of improvement of a parallel design. For this purpose, we have developed costless visual profiling tools presented in Section 4.1.

We explain in Section 4.3 how we conducted an experimentation for performance to reach performance of real-life applications on various architectures: CPU only, CPU with a GPU, CPU with multi-GPU. But before, Section 4.2 presents experiments that focus on determining the intrinsic performance of the library, mainly the internal data transfer latency and the cost of our profiling tools. The real-life applications show different implementations of a matrix multiplication algorithm for homogeneous and heterogeneous nodes, and a CPU-only LU decomposition with partial pivoting. Experiments showing the achieved performance are discussed.

We conclude this chapter by presenting in Section 4.4 two domain-specific libraries that we designed using Hedgehog. They aim at facilitating the design of linear algebra and image processing parallel algorithms.

4.1 Profiling and feedback

In Hedgehog, the developer is at the center of our approach to optimize performance. First, the library has been designed to be as easy as possible to design a parallel algorithm with an explicit model and a strong separation of concerns. Secondly, in order to help understand how the computation is performed on specific hardware, and help to optimize the performance of an implementation, we provide few profiling and feedback mechanisms.

There are three tools directly available: (1) the graphical representation (cf. Section 4.1.1), (2) the signal handler (cf. Section 4.1.2), and (3) the NVTX integration (cf. Section 4.1.3). We have made some tests to evaluate the cost of these profiling tools in Section 4.2.2.

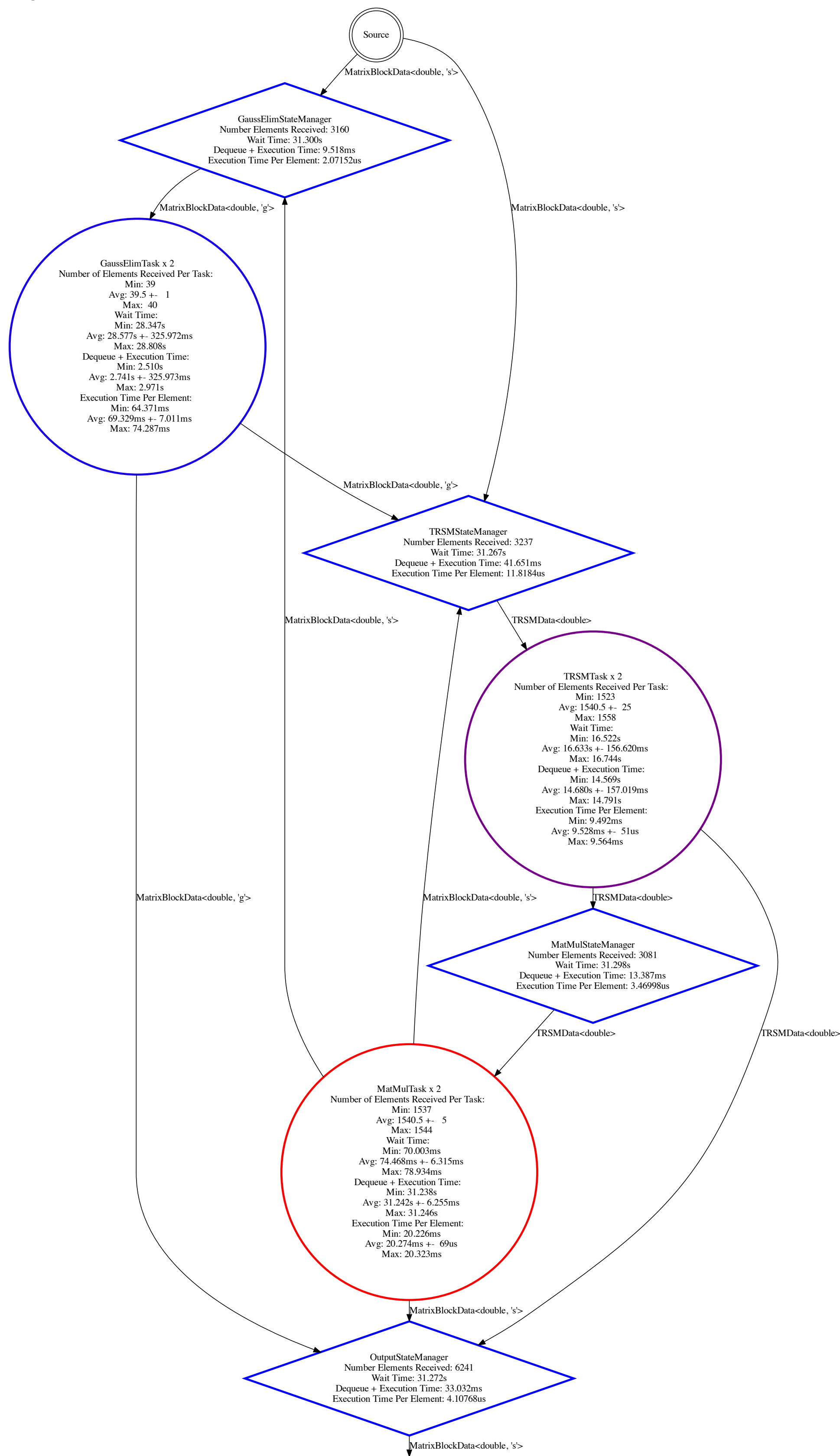


Figure 4.1: DOT representation of the graph achieving a LU decomposition algorithm

4.1.1 Graphical representation

The graphical representation, produced by an implementation of the `Printer` interface in Hedgehog, is a Graphviz [Ellson et al., 2001] DOT file representing the executed data-flow graph with measures collected during runtime. It is based on the *visitor* design pattern [Gamma et al., 1994] that visits all nodes in the graph, all sub-graphs recursively, and gathers metrics depending on the kind of visited node.

Figure 4.1 presents such an output of a graph achieving a LU decomposition algorithm. Shapes are significant: the double circle is the source of the graph, the dot at the bottom is the sink, the circles are the tasks, and the diamonds are the state managers. All the nodes present a customizable color outline depending on their accumulated execution or accumulated wait times. In this example, we chose the execution time to be represented: blue is the fastest and red is the slowest.

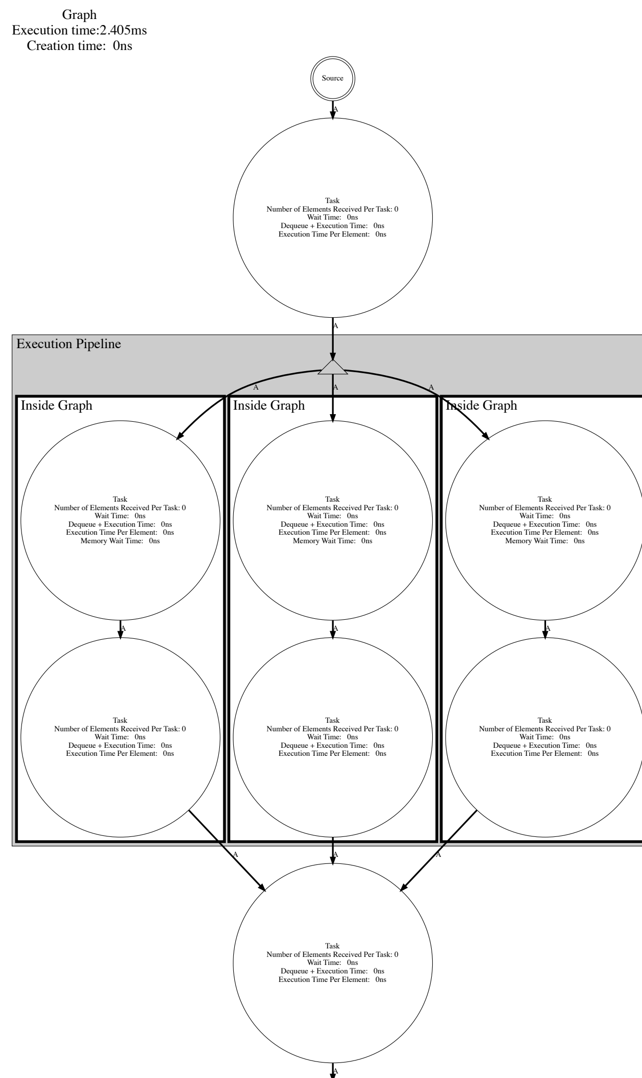


Figure 4.2: Graphical representation showing an execution pipeline

There are multiple options to represent the graph structure. As presented here, all the threads of a task are gathered into one shape and the metrics are presented with some statistics (average and standard deviation). It is possible to serve each

thread on a different shape to have the performance per thread. It is also possible to get some metrics about the queues: how many elements are currently in the queue and what is the maximum size of the queue during the execution.

The metrics gathered for the tasks and state managers are the number of elements received, the global node wait time (no data in queue available), the accumulated dequeue (access to a data in a queue) time added to the accumulated execution time, and the execution time per element. For the overall graph, we present the creation time and the end-to-end execution time.

In this example, there is one task at the bottom (`MatMulTask`) that seems to be the bottleneck because it has the highest execution time (it is in red). A way to improve the overall execution would be to increase the degree of parallelism in this task. This visual feedback used here provides an easy way to improve the computation at a glance.

Not presented in this example, managed memory data are presented with thicker arrows, and execution pipelines appear with the different graphs in a common bounding rectangle with the switch redirecting the data as shown in Figure 4.2. The switch is presented with a triangle in this figure, and the data of type `A` flowing in the graph are managed by a memory manager.

4.1.2 Signal handler

Our feedback mechanism is not only useful to gain performance but also to debug. A graph in Hedgehog can present some directed cycles and, if not handled properly can result in a deadlock situation. A kernel that has not been developed properly can also lead to a segmentation fault. In both cases, a signal can be sent to stop the execution, either manually to stop a deadlock or automatically by the system in the case of a segmentation fault.

We created a signal handler that will produce a DOT representation of the graph, at reception of a registered signal (e.g. `SIGTERM` or `SIGKILL`) before exiting the program. For example, Code 4.1 presents a graph with a directed cycle (lines 3-7) and the usage of the `GraphSignalHandler` class to handle signals (lines 9-14). Its template parameters are the output and input types of the graph. The directed cycle is not taken care of (no override of `canTerminate` for all tasks `t1`, `t2`, and `t3` of the graph), in order to willingly produce a deadlock. Figure 4.3 presents the output produced by the code execution after a `SIGTERM` signal is sent to the program.

We explicitly see that there is a directed cycle in the graph that needs to be handled. With a look at the edges, between `t2` and `t3` there are 100 elements in the queue (*QS:100*), indicating that there is a lot of stalling in this task compared to the maximum queue size (*MQS:100*).

This output helps with solving the current problem of deadlock because of the directed cycle that needs to be handled. Furthermore, there is a lot of waiting in front of the tasks indicating that either the data are too small and the system is too impacted by the inherent system latency and/or the tasks do not present enough parallelism to empty the input queues in time.

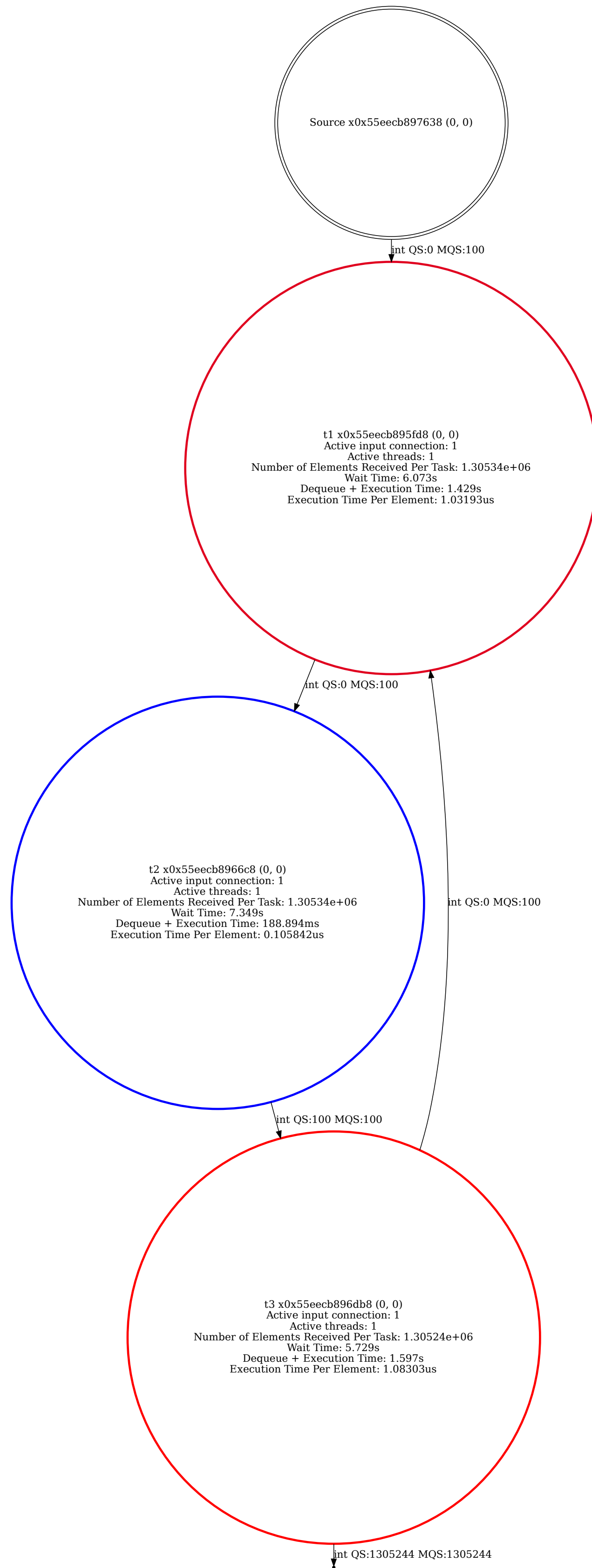


Figure 4.3: Graphical output produced by the signal handler

Source Code 4.1: Usage of the signal handler for a graph with deadlock

```
int main(){
  //...
  g.input(t1);
  g.addEdge(t1, t2);
  g.addEdge(t2, t3);
  g.addEdge(t3, t1);
  g.output(t3);

  using SigHandler = hh::GraphSignalHandler<int, int>;

  SigHandler::registerGraph(&g);
  SigHandler::registerSignal(SIGTERM);
  SigHandler::setColorScheme(hh::ColorScheme::EXECUTION);
  SigHandler::setStructureOptions(hh::StructureOptions::QUEUE);

  g.executeGraph();
  for (int i = 0; i < 100; ++i)
   ↪  g.pushData(std::make_shared<int>(i));
  g.finishPushingData();
  g.waitForTermination();
}
```

4.1.3 NVTX

The NVIDIA Tools Extension (NVTX)[1] is an annotation library from NVIDIA. We integrated the library annotations into Hedgehog to follow the different statuses of the Hedgehog nodes. The statuses registered and followed are initialization, execution (`execute` method), wait time, shutdown, acquisition, and release of managed memory. This allows us to visualize how the different Hedgehog nodes behave in a time-series graph generated with the NVIDIA Nsight Graphics tool[2]. As such, this library is fully compatible with NVIDIA devices. So it is possible to follow the computation on all the hardware (CPU and GPU) in a heterogeneous computational node. The visualization highlights how the overlapping computation is achieved, which tasks are waiting, and when. This new perspective allows us to study how the computation is achieved dynamically and how the different nodes behave.

For example, in Figures 4.4 and 4.5, we show an output of the NVTX profiling for a matrix multiplication implementation on a heterogeneous node, as designed in Figure 4.10. The tasks named *CUDA Product Task*, *Addition Task*, and *Prefetch in GPU* in the NVTX output correspond to the tasks respectively named *GEMM*, *Accumulation*, and *Prefetch In A or B* in Figure 4.10.

The green rectangles are the execution steps, the orange rectangles are the memory wait steps (when no more free piece of data is available in the memory manager, the task is waiting), the red rectangles are the wait steps (waiting for input data from the queues), and the brown rectangles are the shutdown steps (which may be specialized by the end-user and executed once when the task is terminating, it is the shutdown step in Figure 3.6).

The *CUDA Execution Pipeline* task consists mainly in a shutdown step because it is unused in this example; the code is actually made for multi-GPU computation but this run has only been achieved with one GPU.

[1] `https://docs.nvidia.com/nsight-visual-studio-edition/2020.1/nvtx/index.html`
[2] `https://developer.nvidia.com/nsight-graphics`

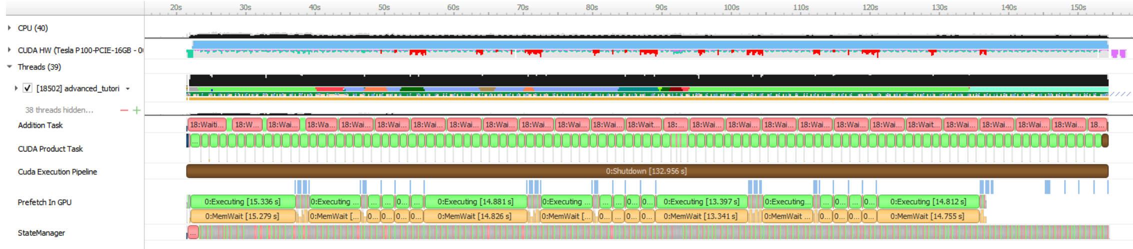


Figure 4.4: Full NVTX profiling of a heterogeneous matrix multiplication implementation

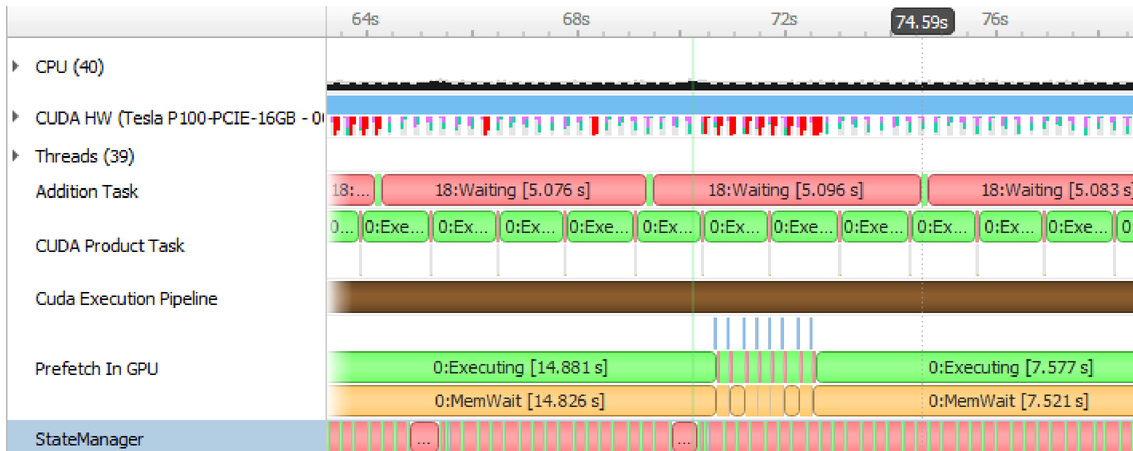


Figure 4.5: Zoom into NVTX profiling of a heterogeneous matrix multiplication implementation

The CPU *Addition Task* is mostly waiting; its sole computation is to aggregate two pieces of data together ($Matrix_C = Matrix_C + Partial$ as shown in Section 4.3.1), which is faster compared to the *CUDA Product Task* made on the GPU.

The GPU tasks *CUDA Product Task* and *Prefetch In GPU* appear in the figure to overlap with the CPU tasks. The prefetch is efficient because it mainly consists of waiting for memory to be available. The GPU main computation task (*CUDA Product Task*) is mainly in an execution state, showing that the device is utilized at most of its capacity by being fed as fast as possible by the prefetch task.

The *StateManager* task deals with pieces of data coming from *CUDA Product Task* and *Addition Task*. It is mainly in a wait state; the computation is minimal, which was expected because it should not do any heavy computation due of the state locking mechanism. If the computation were longer, the state manager would easily become a bottleneck.

This example shows the interest in such profiling tools when dealing with heterogeneous nodes. It exposes that tasks are interdependent, how they overlap, which ones are waiting for memory operations to do the computation, and which ones start right away. The full execution view shows also that both most heavy computation tasks (*Addition Task* and *CUDA Product Task*) end almost at the same time, showing that the work balance between these tasks is efficient. These insights, that clearly refer to the nodes of the data-flow graph, help to debug and to improve a parallel implementation, or to design better algorithms.

4.2 Intrinsic performance

One of our principal concerns when developing Hedgehog was to made it easy to develop with performance in mind. It is difficult to prove the ease of use of a library, but we can provide some metrics about its performance. First, we propose to compare Hedgehog and HTGS by measuring their latencies, the duration to send a piece of data between two tasks, in Section 4.2.1. Then, in Section 4.2.2, we present the cost of our feedback mechanism. Finally, in Section 4.3, we present performance results of matrix multiplication and LU decomposition implementations.

4.2.1 Data transfer latency

Hedgehog inherits from HTGS the execution model as a whole and how an end-user interacts with the graph. When designing Hedgehog, we wanted to have a system that works at least as efficiently as HTGS. The problem is that Hedgehog adds capabilities in the way a graph is expressed (nodes with multiple inputs) and others are simply expressed differently (state manager versus bookkeeper). So, instead of comparing implementations that will inherently present differences in terms of execution, we thought more relevant to compare the latencies in both systems.

We call latency the atomic duration to send a piece of data from a node to another. This duration is variable because the tasks defined with multiple inputs have more work to do than a task with only one input and multi-threaded tasks will share the input queue among the different task copies.

Codes used to achieve this comparison are available in appendices B and C. The main idea of the test consists in a simple graph with only one multi-threaded task that does nothing, so the received data is just transferred to the next node (here to the output of the graph). This task can receive a variable number of different input types in Hedgehog and a structure containing data of different types in HTGS. 1,000,000 elements of different types are sent into the graph. The time between the moment the first piece of data is sent and when the graph terminates is measured. This duration is divided by the number of elements sent to have an estimate of the latency. We repeat this operation for different number of threads attached to the task and for different number of input types. We have gathered these latency measurements in Figure 4.6.

The blue, red, and brown series show the latency measurements for Hedgehog, depending on the number of inputs of the tasks, and the black one shows HTGS measurements. Hedgehog presents a latency from $1\,\mu$s to $10\,\mu$s. HTGS is only presented in one series because the nodes can only have one input type. We have made those measurements for different numbers of threads per task.

We see that in the fairest comparison possible with one input type for Hedgehog and HTGS, we achieve a lower latency in Hedgehog than in HTGS. We have the same analysis with 5 input types per task. We need to have 10 different input types for a receiving node with a high thread count to have higher latency in Hedgehog than in HTGS. This measure is important because (1) this is an indivisible amount of time to transmit a piece of data between two nodes and (2) it determines the granularity of the library parallelism.

The amount of work for each kernel needs to be big enough to balance the latency. This can be adjusted by changing the size of the pieces of data, the block size, stream-

ing through the graph. If the size is "too small", then the underlying hardware may be underutilized, because some tasks do not execute enough instructions compared to the system latency. If the block size is "too big", then this will reduce the parallelism in Hedgehog because fewer pieces of data will feed the nodes. The number of data items streaming through the graph affects the degree of parallelism that can be achieved.

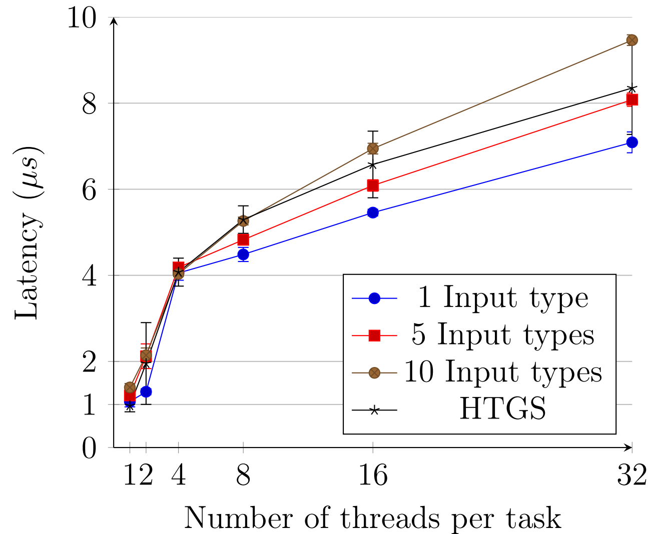


Figure 4.6: Hedgehog and HTGS latency for different task's number of threads

The number of threads assigned to each task will also determine how many elements a task can process in parallel. Setting this number too high can be detrimental if the processor gets oversubscribed. The best methodology for approaching computation in Hedgehog is setting values that the developer estimates are good enough, based on his/her experience, for a first run. This run will be used to generate the DOT file feedback, which will allow identifying bottlenecks, and will help determining better parameters (number of threads and block size) to improve the performance for a specific architecture.

4.2.2 Profiling cost

In Section 4.1, we presented different profiling tools embedded into Hedgehog. We essentially have two profiling approaches: (1) in our implementation the visitor pattern can annotate the data-flow graph with performance measures in a graphical DOT representation, and (2) the integration of the NVTX library to get detailed profiling of the steps of tasks. In order to analyze the impact of these profiling tools, we propose the following statistical analysis.

We compare the performance of Hedgehog's first tutorial (Hadamard product) [3], with and without profiling. The Hadamard product is an element wise product of two matrices. To parallelize the computation, we decompose the matrices into blocks. We execute it 1000 times on $16k \times 16k$ matrices and $2k \times 2k$ blocks on a Mac Book Pro Mid 2015, results are shown in Figure 4.7. This figure shows different end-to-end algorithm execution durations placed into time bins.

[3] `https://pages.nist.gov/hedgehog-Tutorials/tutorials/tutorial1.html`

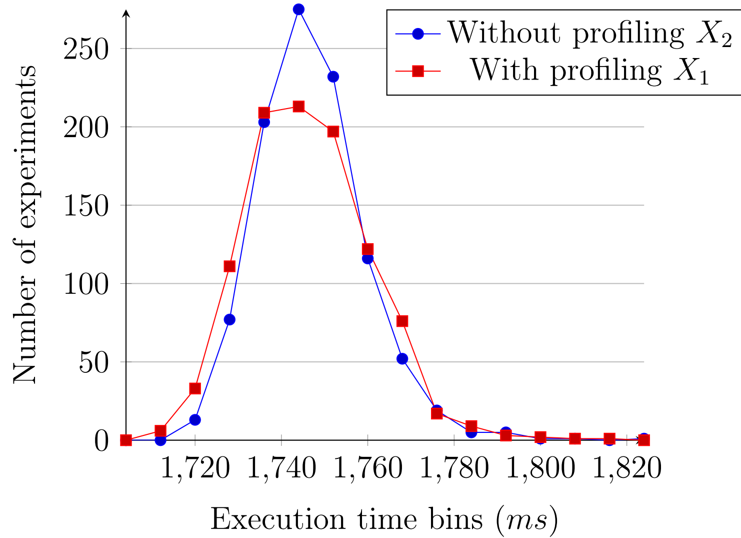


Figure 4.7: Hadamard product using Hedgehog, with and without profiling

We denote X_1 (respectively X_2) as the execution time without (respectively with) profiling. Because of the high number of experiments and thanks to the central limit theorem, the experimental averages, noted $\overline{X_1}$ and $\overline{X_2}$, are distributed following a normal distribution.

We pose $X = \dfrac{\overline{X_1} - \overline{X_2}}{\sqrt{\dfrac{S_1^2}{N} + \dfrac{S_2^2}{N}}}$ which follows a normal distribution of expectation $\mu_1 - \mu_2$ (μ_1 and μ_2 are expectations of X_1 and X_2 respectively) and variance $\sigma^2 = 1$. S_1^2 and S_2^2 are estimators of σ_1^2 and σ_2^2 (variances of X_1 and X_2 respectively).

We test the statistical hypothesis that $\mu_1 = \mu_2$ (the null hypothesis): there is on average no statistical difference between a computation with and without profiling. Consequently, X is transformed into a standard normal distribution.

The size of our sample is N = 1000. For this sample, the values of $\overline{X_1}$ and $\overline{X_2}$ are respectively $\overline{x_1} = 1742.28\,\text{ms}$ and $\overline{x_2} = 1743.06\,\text{ms}$, and the estimations of the variances σ_1^2 and σ_2^2 are respectively $s_1^2 = 14.03\,\text{ms}$ and $s_2^2 = 12.49\,\text{ms}$. With this sample, the value of X is $x = -1.32$, whose p-value is 0.4066, so we can accept the null hypothesis.

Therefore, the average execution runtime with or without profiling does not differ significantly. We believe this is because we gather performance metrics at the node level, which is far less intensive than fine-grained profiling approaches, such as measuring all function invocations.

4.3 Real-life experiments

The experiments presented previously are simple graphs to highlight the measurements. To present real-life parallel computation, we propose in a first section to study matrix multiplication. We present our implementation strategy and our choices, from a CPU-only implementation to an advanced multi-GPU implementation. We also show some results for the latter one. We then present a parallel implementation of a LU decomposition with partial pivoting and some CPU-only results.

4.3.1 Matrix multiplication

For matrix multiplication implementations, we compared our results with ground truth results from the OpenBLAS implementation. The difference between our implementations and OpenBLAS has to be less than $3 \times \epsilon$, ϵ being the hardware precision of the float number representation used. This threshold has been determined with a statistical analysis presented in Annex D.

We propose three implementations for the matrix multiplication algorithm: CPU only in Section 4.3.1, CPU with one NVIDIA GPU in Section 4.3.1, and CPU with multi-GPU in Section 4.3.1.

CPU only matrix multiplication

The first implementation that we propose is CPU only. We use the graph (cf. Figure 1.2) that we have presented in Section 1.7.2. The code is taken from Hedgehog tutorial 3 [4]. The algorithm takes as input matrices A, B, and C, and compute $C' = A \times B + C$. A is a $n \times m$ matrix, B is a $m \times p$ matrix, and C is a $n \times p$ matrix defined as shown in Equations 4.1.

$$
\begin{gathered}
A = \begin{pmatrix} a_{1,1} & a_{1,2} & \cdots & a_{1,m} \\ a_{2,1} & a_{2,2} & \cdots & a_{2,m} \\ \vdots & \vdots & \ddots & \vdots \\ a_{n,1} & a_{n,2} & \cdots & a_{n,m} \end{pmatrix} B = \begin{pmatrix} b_{1,1} & b_{1,2} & \cdots & b_{1,p} \\ b_{2,1} & b_{2,2} & \cdots & b_{2,p} \\ \vdots & \vdots & \ddots & \vdots \\ b_{m,1} & b_{m,2} & \cdots & b_{m,p} \end{pmatrix} \\
C = \begin{pmatrix} c_{1,1} & c_{1,2} & \cdots & c_{1,p} \\ c_{2,1} & c_{2,2} & \cdots & c_{2,p} \\ \vdots & \vdots & \ddots & \vdots \\ c_{n,1} & c_{n,2} & \cdots & c_{n,p} \end{pmatrix}
\end{gathered} \tag{4.1}
$$

In order to get performance by parallelizing the algorithm, we propose to decompose the matrices into blocks. This decomposition forms matrices of blocks A, B, and C of dimension $i \times j$, $j \times k$, and $i \times k$ respectively presented in Equation 4.2. The algorithm can be expressed: $C'_{i,k} = C_{i,k} + \sum_{t=1}^{j} A_{i,t} \times B_{t,k}$.

$$
\begin{gathered}
A = \begin{pmatrix} A_{1,1} & A_{1,2} & \cdots & A_{1,j} \\ A_{2,1} & A_{2,2} & \cdots & A_{2,j} \\ \vdots & \vdots & \ddots & \vdots \\ A_{i,1} & A_{i,2} & \cdots & A_{i,j} \end{pmatrix} B = \begin{pmatrix} B_{1,1} & B_{1,2} & \cdots & B_{1,k} \\ B_{2,1} & B_{2,2} & \cdots & B_{2,k} \\ \vdots & \vdots & \ddots & \vdots \\ B_{j,1} & B_{j,2} & \cdots & B_{j,k} \end{pmatrix} \\
C = \begin{pmatrix} C_{1,1} & C_{1,2} & \cdots & C_{1,k} \\ C_{2,1} & C_{2,2} & \cdots & C_{2,k} \\ \vdots & \vdots & \ddots & \vdots \\ C_{i,1} & C_{i,2} & \cdots & C_{i,k} \end{pmatrix}
\end{gathered} \tag{4.2}
$$

We can decompose the algorithm in two main steps, the block multiplication between blocks of A and B ($P_{i,k,t} = A_{i,t} \times B_{t,k}$) and the accumulation into *C'*: $C'_{i,k} = C_{i,k} + \sum_{t=1}^{j} P_{i,k,t}$. We pose $P_{i,k,t}$ as the partial blocks produced by the product task. The block matrix multiplication can be achieved with a call to the `gemmm` routine from OpenBLAS to reuse its optimized implementation as presented in

[4] `https://pages.nist.gov/hedgehog-Tutorials/tutorials/tutorial3.html`

Code 4.2. The `Order` template parameter is used to indicate the matrix leading dimension (row or column based).

Source Code 4.2: Product task, from CPU matrix multiplication

```
template<class Type, Order Ord = Order::Row>
class ProductTask : public hh::AbstractTask<
  MatrixBlockData<Type, 'p', Ord>,
  std::pair<
    std::shared_ptr<MatrixBlockData<Type, 'a', Ord>>,
    std::shared_ptr<MatrixBlockData<Type, 'b', Ord>> >
> {
  //...
  void execute(std::shared_ptr<
    std::pair<
      std::shared_ptr<MatrixBlockData<Type, 'a', Ord>>,
      std::shared_ptr<MatrixBlockData<Type, 'b', Ord>> >
  > ptr) override {
    auto matA = ptr->first;
    auto matB = ptr->second;
    auto matP = new Type[matA->blockSizeHeight() *
    ↪  matB->blockSizeWidth()]();

    auto res = std::make_shared<MatrixBlockData<Type, 'p',
    ↪  Ord>>(...);

    if constexpr(std::is_same_v<Type, float>) cblas_sgemm(...);
    else if (std::is_same_v<Type, double>) cblas_dgemm(...);
    else {
      std::cerr << "The matrix can't be multiplied" << std::endl;
      exit(43);
    }
    this->addResult(res);
  }
};
```

The accumulation (Code 4.3) into C is only an in-place summation of two matrix blocks.

Source Code 4.3: Addition task, from CPU matrix multiplication

```
template<class Type, Order Ord = Order::Row>
class AdditionTask : public hh::AbstractTask<
  MatrixBlockData<Type, 'c', Ord>,
  std::pair<
    std::shared_ptr<MatrixBlockData<Type, 'c', Ord>>,
    std::shared_ptr<MatrixBlockData<Type, 'p', Ord>> >
> {
  //...
  void execute(std::shared_ptr<
    std::pair<
      std::shared_ptr<MatrixBlockData<Type, 'c', Ord>>,
      std::shared_ptr<MatrixBlockData<Type, 'p', Ord>> >
  > ptr) override {
    auto c = ptr->first;
    auto p = ptr->second;

    if constexpr (Ord == Order::Row) {
      for (size_t i = 0; i < c->blockSizeHeight(); ++i)
        for (size_t j = 0; j < c->blockSizeWidth(); ++j)
          c->blockData()[i * c->leadingDimension() + j] +=
          ↪  p->blockData()[i * p->leadingDimension() + j];
    } else {
      for (size_t j = 0; j < c->blockSizeWidth(); ++j)
        for (size_t i = 0; i < c->blockSizeHeight(); ++i)
```

```
24              c->blockData()[j * c->leadingDimension() + i] +=
                 ↪  p->blockData()[j * p->leadingDimension() + i];
25          }
26
27          delete[] p->blockData();
28          this->addResult(c);
29      }
30  };
```

At this point, there are two problems. First, we push complete matrices into the graph and we manipulate blocks inside the graph. Second, not all blocks need to be multiplied or accumulated altogether. To recall, a block of A, $A_{i,t}$, needs to be multiplied with all the blocks of B of column j. The multiplication task receives a compatible pair of blocks of A and B (created from blocks of A and B by a state manager) in a "chaotic" order, and produces j partial blocks $P_{i,k,t}$. All of these partial blocks $P_{i,k,t}$ need to be accumulated in the same C block $C_{i,k}$.

To answer the second problem, we need to manage the flow of data. The two traversals produce in a specific order blocks of A and B, but we do not know in which order we receive them. In order to manage this flow, we add a state manager in front of the product task to produce a pair of compatible blocks A and B when it receives them. Each block is used multiple times, the state manager keeps them locally as long as there are needed as shown in Code 4.4.

Source Code 4.4: Input state, from CPU matrix multiplication

```
1   template<class Type, Order Ord = Order::Row>
2   class InputBlockState : public hh::AbstractState<
3     std::pair<
4       std::shared_ptr<MatrixBlockData<Type, 'a', Ord>>,
5       std::shared_ptr<MatrixBlockData<Type, 'b', Ord>> >,
6     MatrixBlockData<Type, 'a', Ord>, MatrixBlockData<Type, 'b', Ord>
7   > {
8     //...
9     void execute(std::shared_ptr<MatrixBlockData<Type, 'a', Ord>> ptr)
       ↪  override {
10      // Store the received block of A
11      for(size_t jB = 0; jB < gridWidthRight_; ++jB)
12        if (/* A compatible block of B is available */) {
13          // Update the time to live of A block
14          // If time to live is 0, remove local reference
15          // Create a pair of blocks A and B => "res"
16          this->push(res);
17        }
18    }
19
20    void execute(std::shared_ptr<MatrixBlockData<Type, 'b', Ord>> ptr)
       ↪  override {
21      // Store the received block of B
22      for(size_t iA = 0; iA < gridHeightLeft_; ++iA)
23        if (/* A compatible block of A is available */) {
24          // Update the time to live of B block
25          // If time to live is 0, remove local reference
26          // create a pair of blocks A and B => "res"
27          this->push(res);
28        }
29    }
30  };
```

To split the matrices into blocks, we created two traversal tasks, one that traverses the matrix in row (Code 4.5) and the second that traverses the matrix in column

to create the matrix blocks. This way we can optimize the flow of data to have compatible blocks faster from matrices A and B, instead of traversing both of them in a row fashion.

Source Code 4.5: Row traversal task, from CPU matrix multiplication

```
template<class Type, char Id, Order Ord>
class MatrixRowTraversalTask : public hh::AbstractTask<
  MatrixBlockData<Type, Id, Ord>, MatrixData<Type, Id, Ord> > {
  //...
  void execute(std::shared_ptr<MatrixData<Type, Id, Ord>> ptr)
   ↪  override {
    for (size_t iGrid = 0; iGrid < ptr->numBlocksRows(); ++iGrid)
      for (size_t jGrid = 0; jGrid < ptr->numBlocksCols(); ++jGrid)
        if constexpr (Ord == Order::Row)
          this->addResult(
            std::make_shared<MatrixBlockData<Type, Id, Ord>>(...)
          );
        else
          this->addResult(
            std::make_shared<MatrixBlockData<Type, Id, Ord>>(...)
          );
  }
};
```

The product task needs to have compatible blocks of A and B ($\sum_{t=1}^{j} A_{i,t} \times B_{t,k}$). To execute this task as efficiently as possible, it is best to traverse A in row order and B in column order to produce the blocks. This way, compatible blocks of A and B can be quickly paired to be multiplied. Note here, that the choice of the traversal of C is less impactful because these blocks are used later in the graph for the accumulation.

Finally, we need a state manager to handle the accumulation into C blocks. We have the C blocks coming from the matrix C traversal and partial blocks P coming from the product tasks. The accumulation task accumulates P blocks into blocks of C; the resulting block C is the output of the task that is also sent back to the memory manager. The core computation of the state is to form a pair of blocks C and P, P coming from the product task, and C coming either from the traversal or the accumulation task. There is another problem to tackle there: the directed cycle between the state manager and the accumulation task. We know in advance how many times a specific block of C needs to be accumulated: j times. We can compute then the global task *time to live* which is the number of blocks in C multiplied by t (named `ttl` in Code 4.6). We chose for each block to store how many times each C block has been accumulated and to stop the task when all blocks have been accumulated the right number of times. This state is presented in Code 4.6, its state manager that ultimately breaks the cycle is presented in Code 4.7.

Source Code 4.6: Partial computation state, from CPU matrix multiplication

```
template<class Type, Order Ord = Order::Row>
class PartialComputationState : public hh::AbstractState<
  std::pair<
    std::shared_ptr<MatrixBlockData<Type, 'c', Ord>>,
    std::shared_ptr<MatrixBlockData<Type, 'p', Ord>> >,
  MatrixBlockData<Type, 'c', Ord>, MatrixBlockData<Type, 'p', Ord>
> {
  //...
```

```
9    void execute(std::shared_ptr<MatrixBlockData<Type, 'c', Ord>> ptr)
      ↪ override {
10     if (/* P block is available at same position as C block */) {
11       // Create a pair of C and P
12       // Push the pair in output queue
13       // Decrements the time to live
14     }
15     else // Store the received C block
16   }
17
18   void execute(std::shared_ptr<MatrixBlockData<Type, 'p', Ord>> ptr)
      ↪ override {
19     if (/* C Block is available at same position as P block */) {
20       // Create a pair of C and P
21       // Push the pair in output queue
22       // Decrements the time to live
23     }
24     else // Stores the received P block
25   }
26
27   bool isDone() { return ttl_ == 0; };
28 };
```

Source Code 4.7: Partial computation state manager, from CPU matrix multiplication

```
1  template<class Type, Order Ord = Order::Row>
2  class PartialComputationStateManager : public hh::StateManager<
3    std::pair<
4      std::shared_ptr<MatrixBlockData<Type, 'c', Ord>>,
5      std::shared_ptr<MatrixBlockData<Type, 'p', Ord>> >,
6    MatrixBlockData<Type, 'c', Ord>, MatrixBlockData<Type, 'p', Ord>
7  > {
8    //...
9    // To break the cycle, the "canTerminate" method is overriden
10   bool canTerminate() override {
11     using state_t = PartialComputationState<Type,Ord>;
12     auto s = std::dynamic_pointer_cast<state_t>(this->state());
13     s->lock();
14     auto ret = s->isDone();
15     s->unlock();
16     return ret;
17   }
18 };
```

The last piece is a state manager attached to the output of the addition task that will filter the final C blocks from the ones that need to be accumulated.

The execution of this graph on a Mac Book Pro Mid 2015, with 10000×10000 float matrices divided into blocks of 2048×2048 elements, and with 3 threads for the addition and product tasks, produces the graphical output with performance measurements presented in Appendix E. We see here that the most expensive part of our algorithm is the product task, even with three threads. A way to improve the overall computation duration would be to add more threads to this task.

Heterogeneous matrix multiplication

We now try to achieve the computation with both a CPU and a GPU. The code presented in this section is from Hedgehog tutorial 4[5]. To propose a heterogeneous

[5] https://pages.nist.gov/hedgehog-Tutorials/tutorials/tutorial4.html

implementation of the algorithm, we reuse the graph designed in the CPU-only implementation.

When dealing with small matrices, it is often much better to just keep everything in the GPU and copy the minimum amount of data back to the CPU. When dealing with large matrices, we want to avoid excessive copies as much as possible. The implementation presented targets large out-of-core matrices (i.e., ones that do not fit into GPU memory).

We changed the traversal behavior to operate on a matching column of A with its corresponding row of B. Doing so will reduce the amount of copies required when doing the computation. However, it will also produce an entire copy of the result matrix. To limit memory requirements for the result matrix, we copy the partial results back to the CPU and perform the accumulation on the CPU.

We only change the block product part, because it is the most expensive task in the graph and can benefit from the capabilities of the GPU computation. Moreover, the matrices do not fit in memory, since we limit the usage of C matrix to the CPU and carry out the accumulation on CPU.

Source Code 4.8: Product task on GPU, from heterogeneous matrix multiplication

```
template<class Type>
class CudaProductTask : public AbstractCUDATask<
  CudaMatrixBlockData<Type, 'p'>,
  std::pair<
    std::shared_ptr<CudaMatrixBlockData<Type, 'a'>>,
    std::shared_ptr<CudaMatrixBlockData<Type, 'b'>> >
> {
 private:
   cublasHandle_t handle_ = {};
 public:
  //...
  void initializeCuda() override {
    checkCudaErrors(cublasCreate_v2(&handle_));
    checkCudaErrors(cublasSetStream_v2(handle_, this->stream()));
  }

  void shutdownCuda() override {
    checkCudaErrors(cublasDestroy_v2(handle_));
  }

  void execute(std::shared_ptr<
    std::pair<
      std::shared_ptr<CudaMatrixBlockData<Type, 'a'>>,
      std::shared_ptr<CudaMatrixBlockData<Type, 'b'>> >
  > ptr) override {
    //...
    cublasSgemm_v2(...);
    // or
    cublasDgemm_v2(...);
    //...
    this->addResult(res);
  }
};
```

To port our first CPU-only implementation, we only need to rewrite the product task (cf. Code 4.8) to do the computation on a GPU, and to add two tasks to deal with the data motions (cf. Codes 4.9 and 4.10).

Thanks to the abstraction `AbstractCUDATask`, the CUDA methods `cudaSetDevice`, `cudaStreamCreate`, `cudaStreamDestroy`, and optionally `cudaDeviceCanAccessPeer` are already managed. In the initialization phase (`initializeCuda`),

we call `cublasCreate_v2` and `cublasSetStream_v2`, because they are necessary for `cublasSgemm_v2` or `cublasDgemm_v2` during the execution of the task. In the shutdown phase (`shutdownCuda`), we need to destroy the handle created at initialization by calling `cublasDestroy_v2`. The last modification for this task is to invoke the kernel `cublasSgemm_v2` or `cublasDgemm_v2` depending on the matrix type.

As presented in Section 3.5.1, `initializeCuda` and `shutdownCuda` are the initialization and shutdown steps and are customizable by the end-user for CUDA GPU computation. The `initialize` method from `AbstractTask` is redefined for the `AbstractCUDATask` to bind the threads of a CUDA task to a GPU by calling `cudaSetDevice` to create a CUDA stream `cudaStreamCreate` and to call the customizable initialization step `initializeCuda`. The `shutdown` method is also redefined to call the customizable `shutdownCuda` method and to destroy the CUDA stream. Because the `initialize` method binds the threads to a device, all the CUDA functions in `initializeCuda`, `execute`, and `shutdownCuda` are executed on this device.

We need now to handle the communication between the CPU and the GPU. The order of operations is first a copy of blocks of A and B in the GPU, then the product task, and finally a copy of P blocks out of the GPU. We associate a memory manager to the copy in GPU task, the following GPU computation is then throttled and can run on matrices bigger than the GPU's memory.

In Code 4.9, we show the implementation of the copy in GPU task. When a block is received, it is embedded in a `CudaMatrixBlockData` that is `acquired` from the attached memory manager. The GPU block is then sent to the device asynchronously with `cublasSetMatrixAsync`. The `addResult` is called to inform the next Hedgehog task (the `CudaProductTask`) that a block has been sent to the device.

Source Code 4.9: Copy in GPU task, from heterogeneous matrix multiplication

```
1  template<class MatrixType, char Id>
2  class CudaCopyInGpu : public hh::AbstractCUDATask<
3    CudaMatrixBlockData<MatrixType, Id>,
4    MatrixBlockData<MatrixType, Id, Order::Column> > {
5    //...
6    void execute(std::shared_ptr<MatrixBlockData<MatrixType, Id,
     ↪  Order::Column>> ptr) override {
7      std::shared_ptr<CudaMatrixBlockData<MatrixType, Id>> block =
       ↪  this->getManagedMemory();
8      //...
9      cublasSetMatrixAsync(/*...*/);
10     //...
11     this->addResult(block);
12   }
13 };
```

The following operations are performed by the copy out GPU task, as presented in Code 4.10. When the product task is done, we need to send back the result from the device to the CPU (line 7) and to return the piece of data to the pool (line 8) in order to allow another copy in. Once the block is in CPU memory, it can be sent to the state manager and then to the accumulation task.

Source Code 4.10: Copy out GPU task, from heterogeneous matrix multiplication

```
1  template<class MatrixType>
2  class CudaCopyOutGpu : public hh::AbstractCUDATask<
```

```
3    MatrixBlockData<MatrixType, 'p', Order::Column>,
4    CudaMatrixBlockData<MatrixType, 'p'> > {
5    //...
6    void execute(std::shared_ptr<CudaMatrixBlockData<MatrixType, 'p'>>
     ↪  ptr) override {
7      auto ret = ptr->copyToCPUMemory(this->stream());
8      ptr->returnToMemoryManager();
9      this->addResult(ret);
10   }
11 };
```

These modifications allow porting the CPU-only implementation to a heterogeneous implementation. The resulting data-flow graph is presented in Figure 4.8. The edges holding a memory-managed piece of data are presented thicker than edges holding normally managed data. The green tasks are the CUDA tasks that we have added for this implementation. As usual, the tasks are represented as circles and the memory managers as diamonds. This implementation is limited to one GPU only. We will next discuss how to port this code to a multi-GPU node.

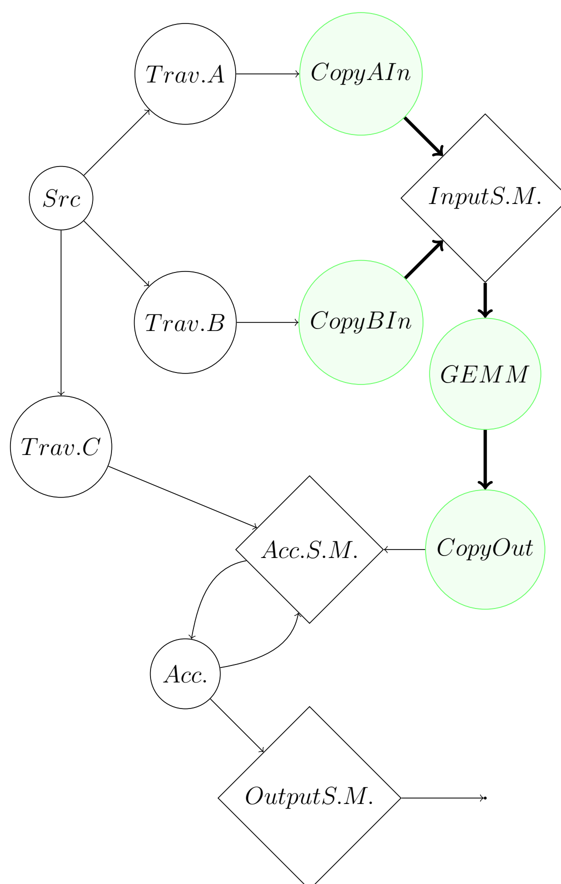


Figure 4.8: Data-flow graph of the heterogeneous matrix multiplication implementation

Multi-GPU matrix multiplication

The code presented in this section has been taken from Hedgehog tutorial 5 [6]. We made some early decisions to achieve multi-GPU computation. First, the computation

[6] `https://pages.nist.gov/hedgehog-Tutorials/tutorials/tutorial5.html`

on all the devices is the same, so we can duplicate the computation on each of them with an execution pipeline. Second, even if optional, we create a separate graph type for the GPU tasks to separate more explicitly the code. Third, to get some performance, we choose to be smart about how we send the blocks to the different GPUs. We start by presenting the GPU's task graph in Code 4.11. In the graph's constructor, the CUDA tasks and the memory managers mentioned in the previous section are instantiated. Then, the data-flow graph is built as usual.

Source Code 4.11: Data-flow graph, from multi-GPU matrix multiplication

```
template<class MatrixType>
class CUDAComputationGraph : public hh::Graph<
  MatrixBlockData<MatrixType, 'p', Order::Column>,
  MatrixBlockData<MatrixType, 'a', Order::Column>,
  MatrixBlockData<MatrixType, 'b', Order::Column> > {
 public:
  CUDAComputationGraph(/*...*/) : hh::Graph</*...*/>("GPU Graph") {
    //...
    // Cuda tasks
    auto copyInATask = /*...*/;
    auto copyInBTask = /*...*/;
    auto productTask = /*...*/;
    auto copyOutTask = /*...*/;

    // Memory managers
    auto cudaMemoryManagerA = /*...*/;
    auto cudaMemoryManagerB = /*...*/;
    auto cudaMemoryManagerProduct = /*...*/;

    // Connect the memory managers
    productTask->connectMemoryManager(cudaMemoryManagerProduct);
    copyInATask->connectMemoryManager(cudaMemoryManagerA);
    copyInBTask->connectMemoryManager(cudaMemoryManagerB);

    // State & state manager
    auto stateInputBlock = /*...*/;
    auto stateManagerInputBlock = /*...*/;

    // Copy the blocks to the device (NVIDIA GPU)
    this->input(copyInATask);
    this->input(copyInBTask);

    // Connect to state manager to get compatible blocks of A and B
    this->addEdge(copyInATask, stateManagerInputBlock);
    this->addEdge(copyInBTask, stateManagerInputBlock);

    // Perform the CUDA product task
    this->addEdge(stateManagerInputBlock, productTask);

    // Send back the memory to the CPU
    this->addEdge(productTask, copyOutTask);

    // Send out the data
    this->output(copyOutTask);
  }
};
```

To use this graph on several GPUs, we can embed it into an execution pipeline as shown in Code 4.12. The execution pipeline holding the different graphs is used as any other node, it is connected to its predecessors and successor tasks.

Source Code 4.12: Usage of execution pipeline, from multi-GPU matrix multiplication

```
// GPU Graph
auto cudaMatrixMultiplication =
  std::make_shared<CUDAComputationGraph<MatrixType>>(n, m, p,
    blockSize, numberThreadProduct);

// Execution pipeline
auto executionPipeline =
  std::make_shared<MultiGPUExecPipeline<MatrixType>>(
    cudaMatrixMultiplication, deviceIds.size(), deviceIds);

// Link the traversal tasks to the execution pipeline
matrixMultiplicationGraph.addEdge(taskTraversalA,
  executionPipeline);
matrixMultiplicationGraph.addEdge(taskTraversalB,
  executionPipeline);

// Link the execution pipeline to the state manager
matrixMultiplicationGraph.addEdge(executionPipeline,
  stateManagerPartialComputation);
```

The only change needed in the implementation of the execution pipeline node is to override the `sendToGraph` method when writing the node's class to fully define our execution pipeline as shown in Code 4.13.

Source Code 4.13: Definition of an execution pipeline, from multi-GPUs matrix multiplication

```
template<class MatrixType>
class MultiGPUExecPipeline : public AbstractExecutionPipeline<
  MatrixBlockData<MatrixType, 'p', Order::Column>,
  MatrixBlockData<MatrixType, 'a', Order::Column>,
  MatrixBlockData<MatrixType, 'b', Order::Column> > {
  //...
  bool sendToGraph(
    std::shared_ptr<
      MatrixBlockData<MatrixType, 'a', Order::Column>> &data,
    size_t const &graphId) override {
    return data->colIdx() % numberGraphDuplication_ == graphId;
  }

  bool sendToGraph(
    std::shared_ptr<
      MatrixBlockData<MatrixType, 'b', Order::Column>> &data,
    size_t const &graphId) override {
    return data->rowIdx() % numberGraphDuplication_ == graphId;
  }
};
```

Our decomposition strategy is to send to the same GPU the matching columns of A and rows of B in a round-robin fashion. This strategy allows us to minimize the number of data motions to the devices and avoid inter-device communication. The final data-flow graph is presented in Figure 4.9. In this figure, the dotted rectangles are the duplicated GPU graphs that we have defined and the triangle is the switch that redirects the different pieces of data to the wanted graphs.

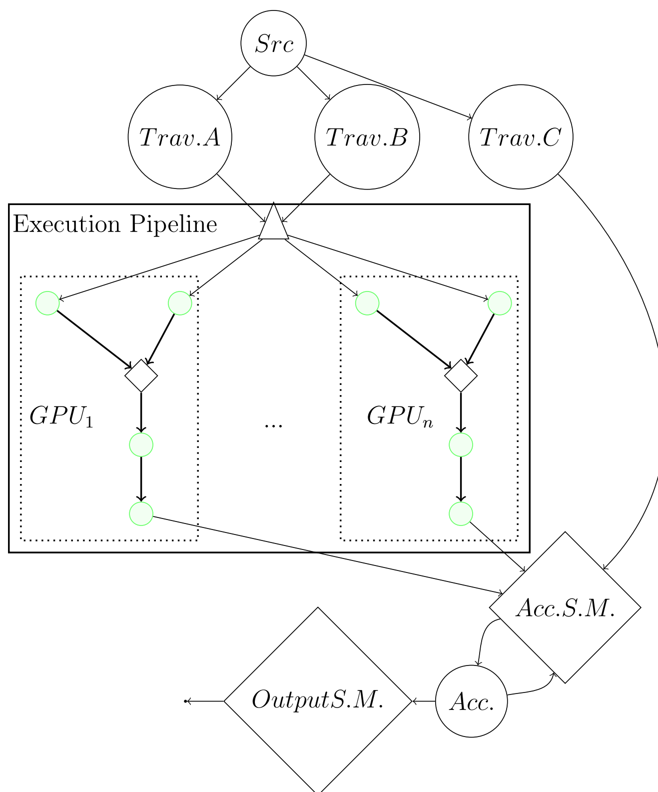


Figure 4.9: Data-flow graph of the multi-GPU matrix multiplication implementation

Advanced multi-GPU matrix multiplication

The most efficient multi-GPU implementation that we have developed and presented is based on the previous implementation with more advanced CUDA features. The code presented in this section is from Hedgehog Advanced Tutorial[7]. Its data-flow graph can be represented by Figure 4.10.

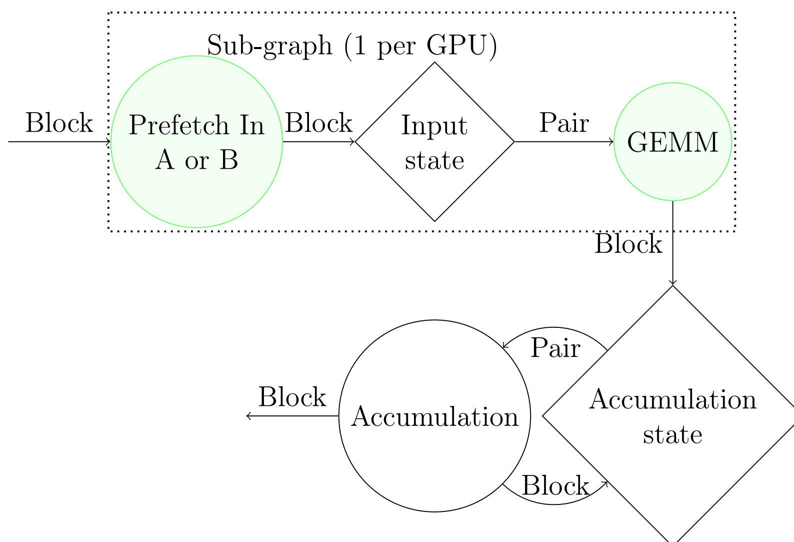


Figure 4.10: Data-flow graph of the advanced multi-GPU matrix multiplication implementation

[7] https://github.com/usnistgov/hedgehog-Tutorials/tree/master/advanced/tutorial1

In this graph, the diamonds are the state managers and the circles are the tasks, the black ones being on CPU and the green ones on GPUs. The dotted rectangle is the GPU graph duplicated in an execution pipeline for each GPU. Properties of these different nodes are given in Table 4.1.

Table 4.1: Properties of the nodes used in the advanced multi-GPU matrix multiplication implementation

Nodes	**Prefetch in**	**Input state**	**GEMM**
Number of Threads	2 (1 stream per thread)	1	5 (1 stream per thread)
Process	Get $Mem_{A\|B}$ Prefetch $Mem_{A\|B}$ to GPU Create $Event_1$	Pair Mem_A and Mem_B	Get Mem_P Prefetch Mem_P to GPU Synchronize $Event_1$ GEMM call Synchronize stream Recycle Mem_A and Mem_B Prefetch Mem_P to CPU Create $Event_2$

Nodes	**Accumulation State**	**Accumulation**
Number of Threads	1	N
Process	Pair Mem_P with C	Synchronize $Event_2$ $C = Mem_P + C$ Recycle Mem_P

Even if the implementation in Section 4.3.1 and this one both target multi-GPU nodes, they present multiple differences.

First, we use CUDA unified memory[8]. This memory is in an address space available to the CPU and the different devices. This data approach expects memory to be contiguous, whereas in our previous implementations the data were stridden, as they were only pointing to different locations in the matrix plus some metadata. This is why we directly feed contiguous unified memory blocks of data from matrices A, B, and C into the graph. These unified blocks are asynchronously prefetched into the GPU or the CPU. The prefetching is synchronized using CUDA events when the data is needed to be used. The memory managers are used to throttle how many prefetches take place in parallel. The product task uses multiple threads to improve the overlap of copying and computing, and provide concurrent kernel execution.

From the matrix decomposition to the product task, the blocks are in-flight. Meaning that when we pair them in the state manager, they are in the process of being sent to the devices, hiding totally the I/O cost. Given a 12 GB/s bandwidth, it takes ≈ 20.8 ms to send a block of $8k \times 8k$ to the 3 tasks, plus the contention on the product tasks (it has 4 threads).

We ran this implementation on a node with two Intel Xeon Silver 4216 CPUs @2.1 GHz (16 physical cores, 32 logical cores) with 768 GiB DDR4 memory and 4 Tesla V100-PCIe with 32 GiB HBM2 GPU, for 64k $\times$ 64k single-precision matrices decomposed in $8k \times 8k$ blocks.

We present in Figure 4.11 the performance obtained with this last Hedgehog implementation compared with NVIDIA implementations from the GPU-accelerated libraries for basic linear algebra, *cuBLASMg* and *cuBLAS-XT* [9]. With the optimal

[8] `https://developer.nvidia.com/blog/unified-memory-cuda-beginners/`
[9] `https://developer.nvidia.com/cublas`

block size on this configuration, we achieved a performance > 85% of the theoretical peak across 4 GPUs.

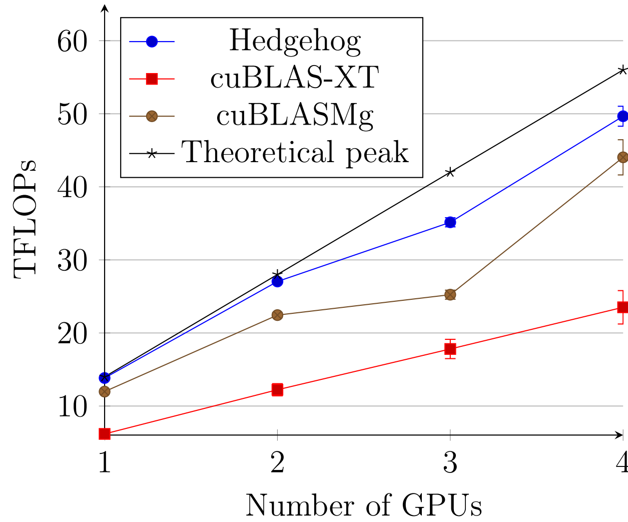


Figure 4.11: Advanced multi-GPU matrix multiplication of $64k \times 64k$ matrices decomposed in $8k \times 8k$ blocks

4.3.2 LU decomposition with partial pivoting

After showing the different phases of development of the matrix multiplication, we present our implementation of the LU decomposition with partial pivoting. Like the matrix multiplication algorithm, we decompose the matrix into blocks to parallelize the algorithm, as presented in Figure 4.12. The implementation is taken from the library *HMBLib* presented in Section 4.4.1.

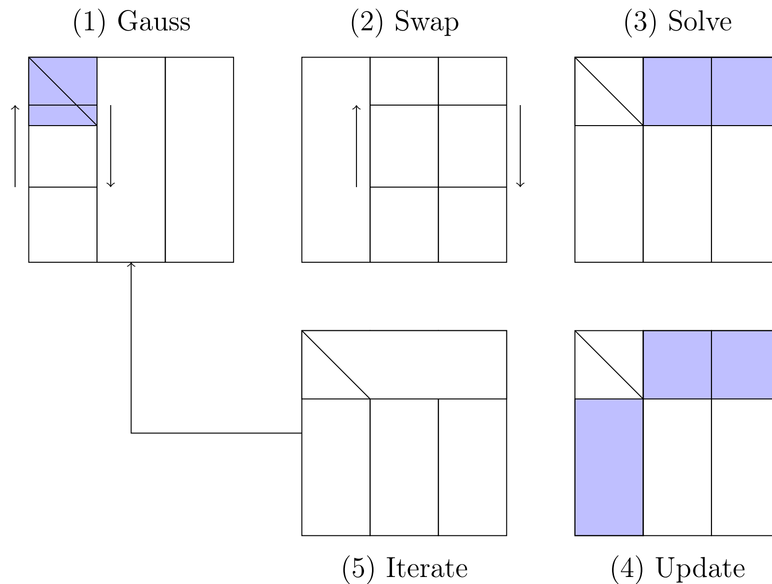


Figure 4.12: LU decomposition algorithm

The algorithm traverses the matrix from left to right (a block of columns), defining the algorithm's iterations. At each iteration the algorithm (1) does a

Gaussian elimination on the panel, (2) swaps the rows on the right or left of the panel, (3) solves the triangular matrix equation on the strip containing the top block, (4) updates the matrix blocks (we decompose the panel in blocks and apply `GEMM` on them in parallel), and (5) as soon as a panel has been updated, it is sent to the Gaussian elimination. This method overlaps the update of the previous iterations (steps 4-5) and the Gaussian elimination for the next iteration (step 1). The partial pivoting drives our choice of using panels, lowering the degree of parallelism. Timothy Blattner in his thesis [Blattner, 2016] conducted experiments over different task-graphs for LU decomposition in order to improve the overall performance. The resulting data-flow graph is shown in Figure 4.13, circles are tasks and diamonds are state managers.

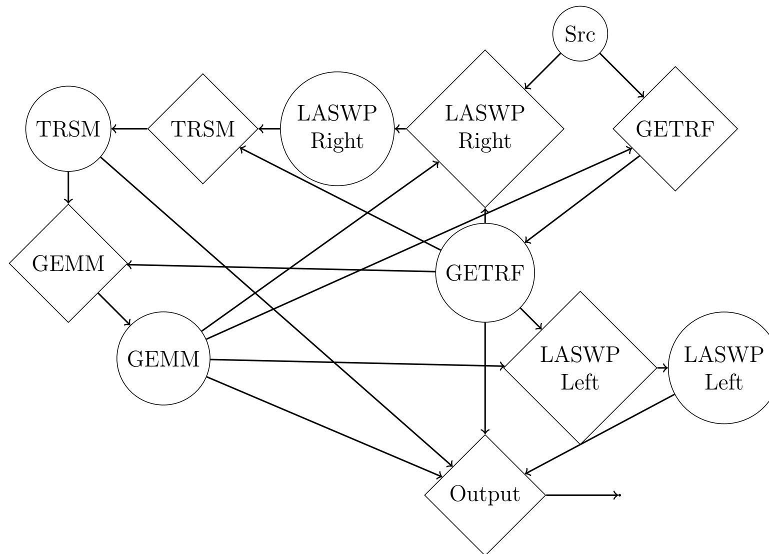


Figure 4.13: Data-flow graph of LU decomposition with partial pivoting

This implementation uses multiple BLAS and LAPACK routines: (1) *GETRF* for a LU factorization using partial pivoting with row interchanges, (2) *LASWP* for row interchanges, (3) *TRSM* for solving a triangular matrix equation, and (4) *GEMM* for matrix multiplication.

We tested this implementation on a compute node with two Intel Xeon E5-2680 @ 2.40 GHz with 28 physical cores (56 logical) and 512 GiB memory (AVX2, 256-bit SIMD vector instruction, is activated). Each experiment was run 10 times for different block sizes from 256×256 to 4096×4096, computing LU decomposition on a $32k \times 32k$ matrix.

From Figure 4.14, we see the importance of choosing the right block size. In this setup, only a block size of 1024×1024 is able to get in par performance (≈ 238 versus ≈ 224 Gflops, for a $1.06\times$ performance improvement) compared to the out-of-the-box LAPACK implementation (`getrf` routine). This is why we have put the developer at the center of our approach and gave him or her all the tools to debug and improve the algorithms that need to be developed. Furthermore, with this block approach, we are able to have a streaming ability as shown in Figure 4.15. In this figure, we measure when the blocks are output and differentiate the first one from the others: (1) the delay between the start of the algorithm and the output of the first block, (2) the average delay between the output of two consecutive blocks.

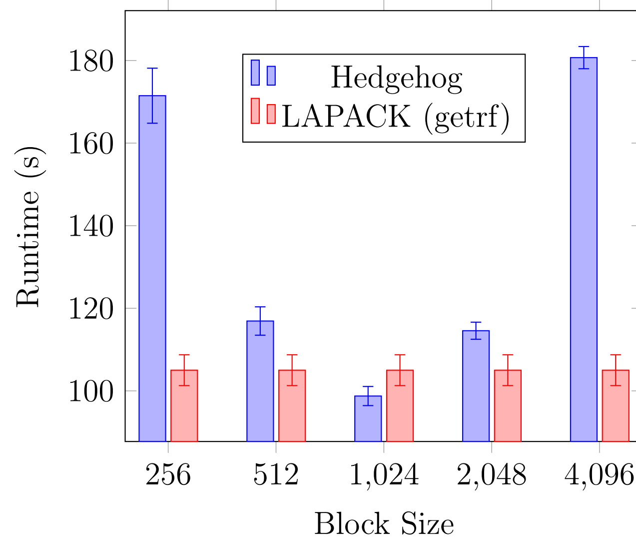


Figure 4.14: CPU implementation performance of LU decomposition with partial pivoting

In fact, we are able to get partial result 42× faster for the first computed block compared to the time needed by LAPACK to get the whole result. So if we chain two graphs, our LU decomposition being the first graph, the second computation can potentially start 42 times earlier than using LAPACK to perform LU decomposition. We also see that on average the blocks are released faster than the first block, showcasing the impact of overlapping computation of several blocks at the same time.

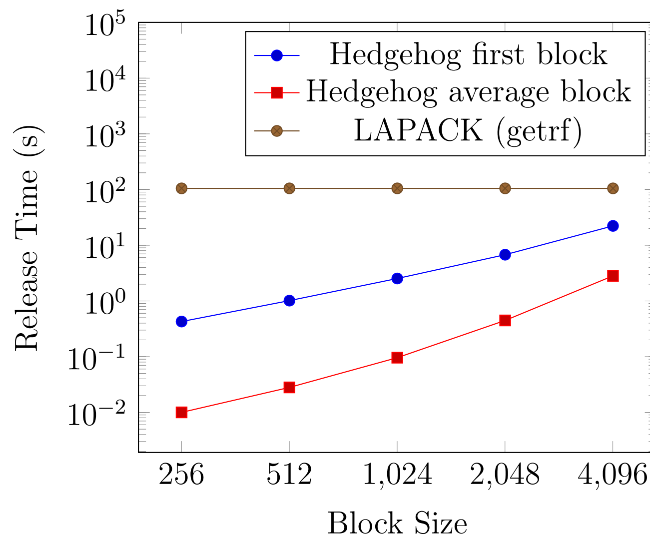


Figure 4.15: Block release time of CPU implementation of LU decomposition

4.4 Libraries based on Hedgehog

We present in this section two libraries constructed using Hedgehog. The first one is closely related to the results from the real-life algorithm we have presented in the

previous section. The second one is a high-performance accessor for loading (tiled) files.

4.4.1 HMBLib

HMBLib, for *Hedgehog Matrix Block Library*, is a linear algebra library built on Hedgehog [Kroiz et al., 2021] to help designing linear algebra algorithms. It proposes abstraction to represent matrices and blocks that are useful to decompose the matrix data and to feed a Hedgehog graph. A block is constructed with only a pointer to a piece of matrix data and some metadata with it to help in the design.

The implementation and achieved results presented in Section 4.3.2 for LU decomposition with partial pivoting came from this library. It also proposes implementations for LU decomposition without partial pivoting and matrix multiplication on CPU.

In this last implementation, we get the same kind of results as for LU decomposition with partial pivoting. These results are obtained on the same computer as for the results already presented (two Xeon E5-2680 @ 2.40 GHz with 28 physical cores - 56 logical - and 512 GiB Memory), over 10 runs for each experiment on a matrix of $32k \times 32k$ elements. We observe that the performance differs depending on the block size. With this implementation, the best performance (≈660 Gflops) is achieved with a block size of 2048×2048 elements, as shown in Figure 4.16.

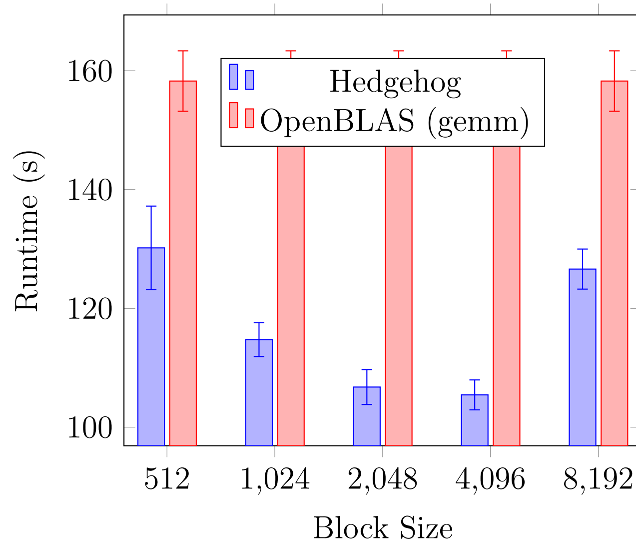


Figure 4.16: Performance of HMBLib matrix multiplication implementation

Moreover, if we study the release rate of blocks, we see that we obtain the first fully computed block 57 times faster with our library than the fully computed matrix using OpenBLAS, and the impact of the overlapped computation on the release rate of the following blocks, as shown in Figure 4.17.

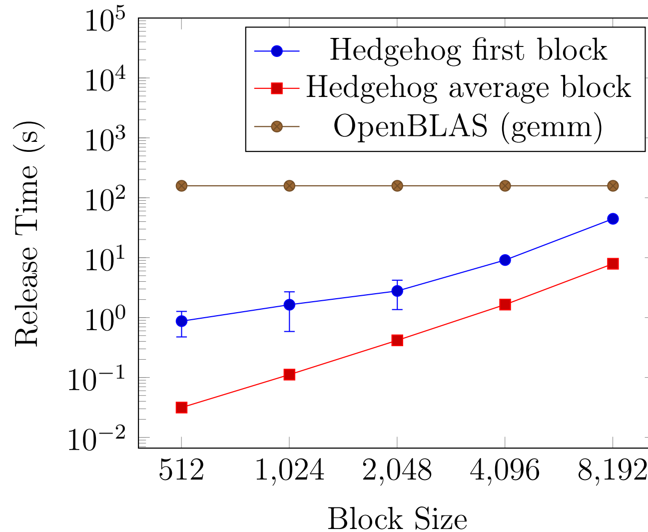


Figure 4.17: Block release time of CPU implementation of matrix multiplication with HMBLib

4.4.2 FastLoader

The second library that we present here is FastLoader [10]. It is an efficient open-source parallel file accessor library, successor of FastImage, an image accessor library based on HTGS. FastLoader proposes to access, with the same API, 2D or 3D, planar or pyramidal, files. The data-flow graph of the library is presented in Figure 4.18.

The library never presents the full file to the end-user, but a portion called a *view*. A view is a contiguous piece of data that is centered on a region of the file, called a *tile*, plus the data of its surrounding area. The views, like the tiles, can be in 2D or 3D, and are either on CPU memory or NVIDIA unified memory to help with GPU computation.

When the end-user, through his or her algorithm (in red in the figure), requests a view to the graph, the library will request the needed tiles from the file through the `Abstract Tile Loader` abstraction (in blue in the figure). The core library is agnostic towards file formats; only a specialized tile loader for a specific file format accesses the file (in brown in the figure), does the necessary conversion depending on what is requested by the user, and provides the data to the library. To get performance, the tile loaders are used with multiple threads, and to avoid too many I/O operations, tile caches are used to temporarily store file tiles.

When the view is fully constructed, it is sent to the algorithm. It is possible to request multiple views at the same time, thereby improving the overall performance by overlapping the different graph's tasks. However, if we request unified memory, we may want to stay in the limit of a GPU, and then memory managers (in violet in the graph) have to be added for each sub-graph.

This library shows another usage of execution pipelines. Here, we do not have a sub-graph per GPU, as we had for the matrix multiplication implementation with multiple GPUs, but we have a sub-graph per pyramid level.

[10] `https://github.com/usnistgov/FastLoader`

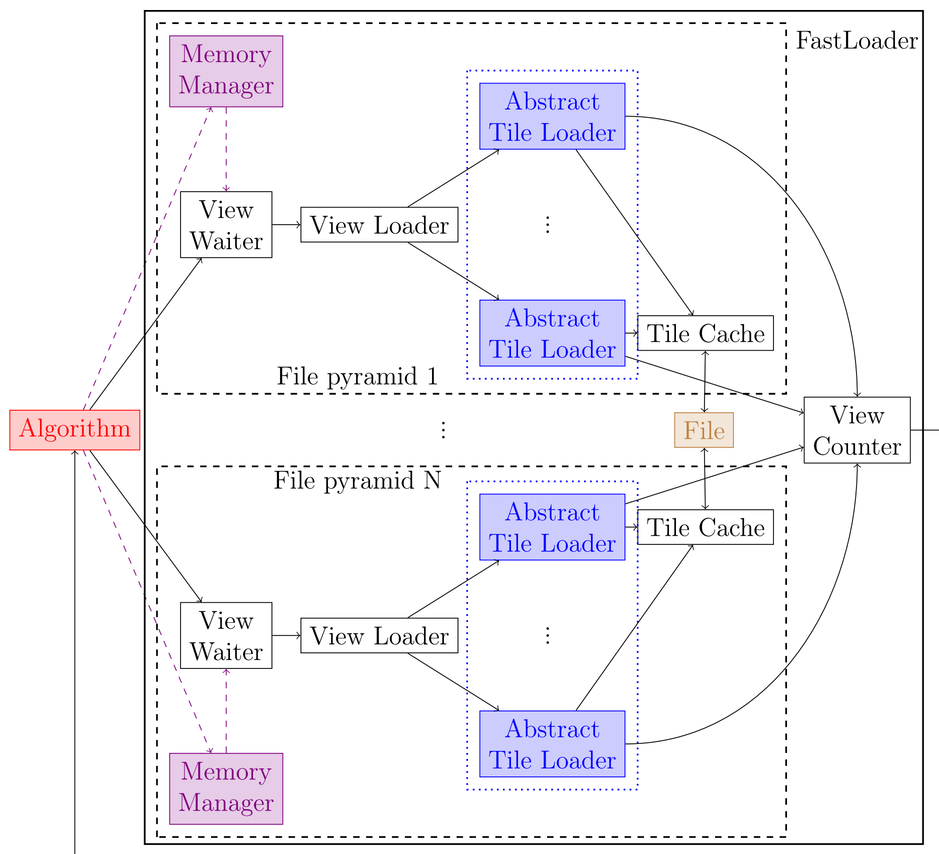


Figure 4.18: FastLoader data-flow graph

This library is recent and features are currently added to the library. We are currently working on data bindings between FastLoader and OpenCV plus TensorRT to ease the usage of the library for image processing and deep learning algorithms. Efforts are put in place to propose a binding in Python and Java, to let users reluctant to use C++ the possibility to use the library to access big files.

4.5 Conclusion

Hedgehog puts the developer at the center of the approach to design parallel algorithms. The explicit data-flow model and the multiple costless graphical feedback mechanisms available help the user to understand with details how the computation is really conducted on specific hardware, and allow him/her to iterate in the implementation process to improve performance.

We have also shown that despite being explicit and accessible, the library is totally amenable to building efficient parallel algorithms and also to creating libraries to ease the design of domain-specific parallel algorithms. We briefly presented two libraries, FastLoader for image processing and HMBLib for linear algebra, built upon Hedgehog.

We have also shown that the data pipelining inherent in the model of Hedgehog presents a latency between $1\,\mu s$ and $10\,\mu s$ to transfer a piece of data between two nodes of the data-flow graph. The advanced multi-GPU implementation of the matrix multiplication explained how to reach above 95% of the theoretical peak on a node with 4 GPUs. With our LU decomposition CPU implementation, we showed

that additionally to get good performance, we enabled, thanks to our block approach, streaming properties that allow chaining algorithms more than 40 times sooner than executing algorithms in sequence, each time waiting for the full result of the previous task.

To the accessibility and performance of the library, we want to add safety in the development of parallel algorithms. In the next chapter, we propose compile-time mechanisms to secure the use of the library by the developer, and to detect potential issues in the designed data-flow graph.

Chapter 5

Hedgehog at compile-time

Hedgehog provides an API for end-users to build a data-flow graph from separate elements (tasks, managers, other graphs...) by connecting them together. During the graph assembling phase, some checking is necessary to be performed at compile-time by the library, to provide clear error or warning messages as soon as possible in the design process. We would like as much of the checking as possible to take place before running the program, to provide maximal guarantees to the end-user on the built graph structure and avoid failure later at runtime. Furthermore, some IDEs could be "smart" enough to provide the result of this checking while the developer is still coding.

Some conformity checking can be done directly when connecting two elements, for instance, examining the output type of a node to determine if it is compatible with the input type of the node one wants to connect it with. We present in Section 5.1 an approach for achieving this type of checking, using either template metaprogramming's usual techniques solely (cf. Section 2.3.9), or combined with C++ 20's concepts (cf. Section 2.3.8). As we have mentioned in Section 3.6.1, these two approaches have been implemented in two versions of the library, one compliant with C++ 17 (v.1) and the other one with C++ 20 (v.2).

More sophisticated analyses of the whole resulting graph structure can also be performed at compile-time. For instance, one would be alerted of directed cycles in the graph, which can lead to deadlocks if not handled properly before runtime, or of data races that could occur when data are broadcast to multiple nodes (possible concurrent read and write operations). This is made possible with constant expressions as explained in Section 5.2, but it requires the graph to be handled both at compile-time and runtime. That implies the end-user must separate his/her design of a data-flow graph in two parts, and this necessitates a mechanism inside the library that automatically merges those two parts. This advanced compile-time functionality is compliant with C++ 20 (Hedgehog v.2).

In Section 5.3, we propose experiments on these checking capabilities to draw some limits and determine the compile-time performance of these features.

5.1 Conformity checking at graph construction

When building a Hedgehog graph, an end-user needs to specialize different API abstractions to create his or her own kinds of nodes. Nodes are then instantiated from these user-defined classes and connected in a graph through our API, mainly methods `input`, `output`, and `addEdge` of class `Graph`. These methods are template and the

compiler usually deduces the true types bound to the template parameters when the methods are called. More precisely, the compiler attempts to instantiate them, and then use them, notably when connecting nodes, their respective type is normally automatically deduced (cf. Section 2.3.3). However, we have to add constraints on these template parameters that will be asserted during the instantiation of the template methods (or template classes) to ensure compatibility on the node usage when connecting nodes together.

5.1.1 Basic checking

In the simplest cases, the types needed to achieve checking are directly available from the context. For example, a memory manager can be connected to a task using method `connectMemoryManager` of class `AbstractTask`, and this operation should be allowed only if the type handled by the memory manager, the same type as the output type of the task (`TaskOutput`), has specific properties (i.e. inherits from `MemoryData` and has a default constructor). Such checking can be achieved as shown in Code 5.1.

Source Code 5.1: Static checking at memory manager connection

```
1  template<class TaskOutput, class... TaskInputs>
2  class AbstractTask {
3    //...
4    void
       connectMemoryManager(std::shared_ptr<MemoryManager<TaskOutput>>
   ↪   mm){
5      static_assert(
6        traits::is_managed_memory_v<TaskOutput>,
7        "The type given to the managed memory should inherit
          ↪  \"MemoryData\", and be default constructible!");
8      mm_ = mm;
9    }
10   //...
11 };
```

In this case, the context is the class `AbstractTask`. It presents immediately the necessary information, the `TaskOutput` type, to constrain what type is accepted by the `connectMemoryManager` method (line 4). Once this first filtering step is done, a `static_assert(p,m)` tests at compile-time a predicate `p` (lines 5-7). The predicate for this example is `is_managed_memory_v`: it tests if the type `TaskOutput` is manageable through a `MemoryManager`, if not, we print an error message `m` in the compiler's output.

Not all checks in Hedgehog are as direct as this one; the following sections show more sophisticated type checking tests and we discuss many ways of implementing them.

5.1.2 Input node connection

To connect a node `x` as input of a graph `g` by calling `g.input(x)`, the node needs to (1) be a receiver and (2) share at least one input type with the graph. As shown in Figure 5.1, if we consider `g` of type `Graph<O1,I1,I2,I3>`, node `x` of type `Node<O2,I1,I4>` is compatible (left case in the figure), whereas of type `Node<O2,I4,I5>` is not (right case).

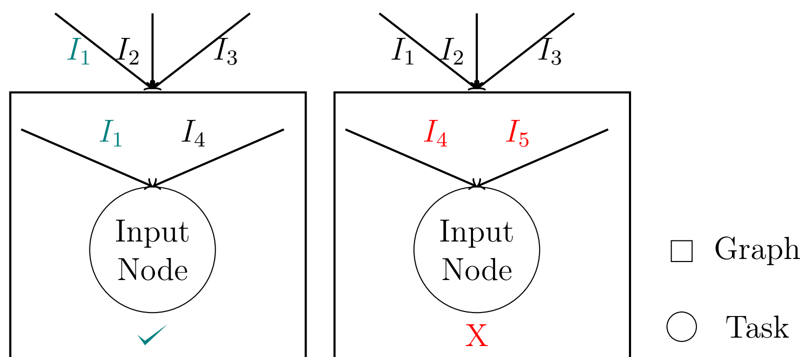


Figure 5.1: Input conformity check between a node and a graph

Implementation with template metaprogramming

In the first version of Hedgehog, the check is made with template metaprogramming techniques as shown in Code 5.2. Method `input` receives a shared pointer of type `Node` (line 19), and there are two checks at compile-time: (1) `Node` is a `MultiReceivers` (constraint at lines 9-14), (2) the input types of `Node` are compatible with those of the graph (constraint at lines 15-17).

To achieve this test, we need first to get the input types of type `Node` (`InputsN` at line 6); this is the type `inputs_t`, which is defined in superclass `MultiReceivers` with a `using` statement and which provides a tuple with the input types. The same technique is used to get the input types of the graph at lines 7-8. Later in this section, the output type of a node will be accessed similarly, with the inner type `output_t` defined in class `Sender`.

Source Code 5.2: Static checking of graph `input` method, using template metaprogramming

```
template<class GraphOutput, class... GraphInputs>
class Graph{
  //...
  template<
    class Node,
    class InputsN = typename Node::inputs_t,
    class InputsG =
      typename behavior::MultiReceivers<GraphInputs...>::inputs_t,
    class isMultiReceiver = std::enable_if_t<
      std::is_base_of_v<
        typename helper::HelperMultiReceiversType<InputsN>::type,
        Node
      >
    >,
    class isInputCompatible = std::enable_if_t<
      traits::is_included_v<InputsN, InputsG>
    >
  >
  void input(std::shared_ptr<Node> input) { /*...*/ }
};
```

We test if type `Node` inherits from our multi-receiver class as shown at lines 9-14 of Code 5.2, using `HelperMultiReceiversType` defined in Code 5.3 that returns class `MultiReceivers` instantiated with, as arguments, the input types of type `Node`.

Source Code 5.3: Helper to get `MultiReceivers` type from input type list

```
// Primary version: only declaration
template<class Inputs>
struct HelperMultiReceiversType;

// Specialized version: the parameter is "tuple<Inputs...>"
template<class ...Inputs>
struct HelperMultiReceiversType<std::tuple<Inputs...>> {
  using type = behavior::MultiReceivers<Inputs...>;
};
```

Finally, we need to check if at least one of the input types of `Node` is part of the input types of the graph at lines 15-17 of Code 5.2. This test is achieved by trait `is_included` (defined in Code 5.4) that is used in conjunction with the `enable_if` metafunction to remove the `input` method from the overload resolution, and leading to compilation error if providing a `Node` type failing this test.

Source Code 5.4: Metafunction checking if at least one of the set of types `T1` is included in set `T2`

```
// "T1" and "T2" must be tuples

// Primary version (hidden sub-part): only declaration
template<class T1, class T2, class Is>
struct _is_included_;

// Specialized version (hidden sub-part): generate sequence of
↪  indexes to test each element
template<class T1, class T2, std::size_t... Is>
struct _is_included_<
  T1, T2,
  std::integer_sequence<std::size_t, Is...> > {
  static bool const
    value = std::disjunction_v<
      Contains<typename std::tuple_element<Is, T1>::type, T2>...
    >;
};

// Primary version (visible part)
template<class T1, class T2>
struct is_included :
  _is_included_<
  T1, T2,
  std::make_integer_sequence<
    std::size_t, std::tuple_size<T1>::value> > {};

// Helper
template<class T1, class T2>
inline constexpr bool is_included_v = is_included<T1, T2>::value;
```

`is_included` tests in sequence if each type (thanks to `std::integer_sequence` that enumerates the indexes) of the `Node` input types (tuple `InputsN`) bound to parameter `T1` is contained in the graph input types (tuple `InputsG`) bound to `T2` (lines 8-16). The result of the trait is the logical OR (`disjunction_v`) of these tests (lines 12-15).

In order to check if a type `T` of `T1` is in the set `T2`, we call the trait `Contains` (line 14 of Code 5.4) defined in Code 5.5 that checks sequentially if the type `T` (bound to a type of `T1`) is the same as the type contained in `Ts` (bound to `T2`). The result is the disjunction of these tests.

Source Code 5.5: Metafunction checking if type `T` is part of set `Ts`

```
1  // Primary version: the set is parameter pack "Ts"
2  template<class T, class... Ts>
3  struct Contains {
4    constexpr static bool value =
5      std::disjunction_v<std::is_same<T, Ts>...>;
6  };
7
8  // Specialized version: the set is "tuple<Ts...>"
9  template<class T, class... Ts>
10 struct Contains<T, std::tuple<Ts...> > : Contains<T, Ts...> {};
```

Some IDEs like CLion are capable of exploiting these statics tests and can show during the development, directly in the code, that the method is not defined (Figure 5.2, result of compilation with Clang).

```
#include "hedgehog/hedgehog.h"

class TestTask : public hh::AbstractTask<double, double> {
 public:
  void execute(std::shared_ptr<double> ptr) override {
    this→addResult( output: ptr);
  }
};
int main() {
  hh::Graph<int, int, float> g;
  auto t1 = std::make_shared<TestTask>();
  g.input(t1);
}
         No matching member function for call to 'input'
         candidate template ignored: requirement 'traits::is_included_v<std::tuple<double>, std::tuple<int,
         float>>' was not satisfied [with Node = TestTask, InputsN = std::tuple<double>, InputsG = std::
         tuple<int, float>, isMultiReceiver = void]
```

Figure 5.2: CLion IDE hint of Hedgehog v.1 static checking (failure of `enable_if`)

In this example, the graph accepts `int` or `float` as input and the task accepts only `double`, which is an error because `double` is neither a `int` or a `float`.

This could be difficult to debug for an end-user, he or she needs to read the comments and figure out what the problem is. That is why we propose in the following section to use concepts to constrain template parameters, instead of using the `enable_if` construct, and sometimes static assertions to provide clearer output error messages from the compiler.

Implementation with concepts

As shown in Section 2.3.8, concepts are a C++ 20 addition that is an alternative to the `enable_if` construct to constrain template parameters, first with a syntax that is easier and clearer to use, and second with much more expressiveness that allows defining classes of types by specifying a set of requirements.

As a first implementation, we propose to define several concepts (Code 5.6) to express requirements that Hedgehog nodes must satisfy to use the connection API. First, concept `HedgehogNode` (lines 2-4) requires that a type `DynamicHedgehogNode` inherits from `Node` class. Then, the two concepts of (1) a node sending data (concept `HedgehogSender`, lines 18-24) and (2) a node receiving data of multiple types (concept `HedgehogMultiReceiver`, lines 7-15) are defined. Finally, we define the `HedgehogConnectableNode` concept (lines 27-30) as the conjunction of the three previous concepts to define the requirements of a Hedgehog node that can receive and send data.

Source Code 5.6: Definition of concepts for Hedgehog nodes.

```
// Concept "HedgehogNode"
template<typename DynamicHedgehogNode>
concept HedgehogNode =
  std::is_base_of_v<hh::behavior::Node, DynamicHedgehogNode>;

// Concept "HedgehogMultiReceiver"
template<typename DynamicHedgehogNode>
concept HedgehogMultiReceiver =
  HedgehogNode<DynamicHedgehogNode>
    && std::is_base_of_v<
      typename hh::helper::HelperMultiReceiversType<
        typename DynamicHedgehogNode::inputs_t
      >::type,
      DynamicHedgehogNode
    >;

// Concept "HedgehogSender"
template<typename DynamicHedgehogNode>
concept HedgehogSender = HedgehogNode<DynamicHedgehogNode>
    && std::is_base_of_v<
        hh::behavior::Sender<
          typename DynamicHedgehogNode::output_t>,
        DynamicHedgehogNode
    >;

// Concept "HedgehogConnectableNode"
template<typename DynamicHedgehogNode>
concept HedgehogConnectableNode = HedgehogNode<DynamicHedgehogNode>
    && HedgehogMultiReceiver<DynamicHedgehogNode>
    && HedgehogSender<DynamicHedgehogNode>;
```

`HedgehogMultiReceiver` is used to constrain the template parameter of method `input` of class `Graph`, as shown in Code 5.7, providing an alternative to the `enable_if` construct of Code 5.2. Notice that in this new version of the `input` method and when the concepts defined here are used in the following sections, that, a tighter constraint is put on the `Node` template parameter, compared to the primary version of the previous section.

In this function definition, we use the concept to filter the types and static assertion (line 4) to check the compatibility with the graph reusing the same trait.

When developing the same code as Figure 5.2, we have an output from the compiler that could be exposed by an IDE (cf. Figure 5.3) that is clearer than previously. Here, `t1` has a type that satisfies concept `HedgehogMultiReceiver`, but fails the static assertion of line 10 in Code 5.7.

Source Code 5.7: Static checking of graph `input` method, using concepts and static assertion

```
template<class GraphOutput, class... GraphInputs>
class Graph {
  //...
  template<HedgehogMultiReceiver Node>
  void input(std::shared_ptr<Node> input) {
    using InputsN = typename Node::inputs_t;
    using InputsG = typename
      behavior::MultiReceivers<GraphInputs...>::inputs_t;

    static_assert(
      traits::is_included_v<InputsN, InputsG>,
        "The input node should share at least one input type with
          the graph.");
    //...
  }
  //...
};
```

```
int main() {
  hh::Graph<int, int, float> g;
  auto t1 = std::make_shared<TestTask>();
  g.input( input: t1);
}
```

In template: static_assert failed due to requirement 'traits::is_included_v<std::tuple<double>, std::tuple<int, float>>' "The input node should share at least one input type with the graph."
error occurred here
in instantiation of function template specialization 'hh::Graph<int, int, float>:: input<TestTask>' requested here

Figure 5.3: CLion IDE hint of Hedgehog v.2 static checking (failure of static assertion)

We can go one step further with concepts and define a concept that requires what has been statically asserted in the previous code (let us call it the *connectivity requirement* here). As shown in Code 5.8, concept `HedgehogInputGraphNode` with two template parameters is defined as the conjunction of concept `HedgehogConnectableNode` and the connectivity requirement (that checks if one of the inputs of template parameter `DynamicHedgehogNode` is one of the inputs of template parameter `GraphInputs`).

In this version, we define the concept with an additional template parameter that represents the input types of the graph (it is assumed to be a tuple), allowing us to embed the trait requirement directly into the concept. With the same code as Figure 5.2, the message of the compiler exposed by an IDE (cf. Figure 5.4) is not fully clear (compared to the previous version with the static assertion), but compilers give a link in their message to the traits in the code, and the user can read the comments to solve the problem.

Source Code 5.8: Static checking of graph `input` method, using concepts only

```
// Concept "HedgehogInputGraphNode"
template<typename DynamicHedgehogNode, class GraphInputs>
concept HedgehogInputGraphNode =
  HedgehogConnectableNode<DynamicHedgehogNode>
  // The node must share at least one input type with the graph.
  && traits::is_included_v<typename DynamicHedgehogNode::inputs_t,
   ↪  GraphInputs>;

// Method "input"
template<class GraphOutput, class... GraphInputs>
class Graph {
  template<HedgehogInputGraphNode<
    typename behavior::MultiReceivers<GraphInputs...>::inputs_t>
     ↪  Node>
  void input(std::shared_ptr<Node> input) { /*...*/ }
};
```

```
int main (){
  hh::Graph<int, int, float> g;
  auto t1 = std::make_shared<TestTask>();
  g.input(t1);
}
```

No matching member function for call to 'input'
candidate template ignored: constraints not satisfied [with Node = TestTask]
because 'HedgehogInputGraphNode<TestTask, typename behavior::MultiReceivers<int, float>::inputs_t>' evaluated to false
because 'hh::traits::is_included_v<typename TestTask::inputs_t, std::tuple<int, float> >' evaluated to false

Figure 5.4: CLion IDE hint of Hedgehog v.2 static checking (full concept)

We chose the implementation in Code 5.6 for our library, as it presents a simpler definition of the checking and the method, compared to the template metaprogramming one (Code 5.2), and a clearer output message at compilation for the end-user than the versions in Code 5.2 and Code 5.8.

5.1.3 Output node connection

In this section, we present the static checking needed to connect an output node to a graph. The rule is that a node `x` can be connected as output of a graph `g`, achieved by calling `g.output(x)`, if (1) `x` is a sender and (2) it has the same output type as the graph. As seen in Figure 5.5, if we consider `g` of type `Graph<O1,I1,I2,I3>`, node `x` of type `Node<O1,I4,I5>` is compatible (left case of the figure), whereas of type `Node<O2,I4,I5>` is not (right case).

Implementation with template metaprogramming

Similarly to input conformity in the previous section, we implement the check for output in Hedgehog v.1 using template metaprogramming techniques, as shown in Code 5.9.

Source Code 5.9: Static checking of graph `output` method, using template metaprogramming

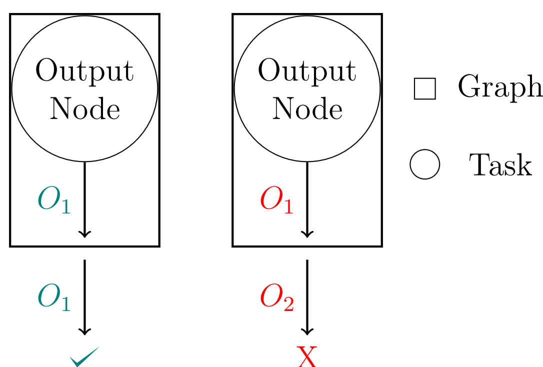


Figure 5.5: Output conformity check between a node and a graph

```
template<class GraphOutput, class... GraphInputs>
class Graph{
  //...
  template<
    class Node,
    class IsSender = std::enable_if_t<
      std::is_base_of_v<behavior::Sender<GraphOutput>, Node>>
  >
  void output(std::shared_ptr<Node> output) { /*...*/ }
};
```

The code here is simpler than the one to check the input node connection, because we get directly the output type information needed from the graph definition. We only need to check if the type of the object sent by the user is a sender with the graph output type `GraphOutput` (line 7). This is done with a call to a trait defined in the standard library called `is_base_of` that tests the inheritance between the type `Node` and class `behavior::Sender<GraphOutput>` of our library.

Implementation with concepts

The same checking can be achieved with concepts and a static assertion, similarly to what has been done for input connection. In Hedgehog v.2, concept `HedgehogSender previously defined in Code 5.6` is used to constrain method `output` as shown in Code 5.6, with the static assertion defined in the previous section to test the compatibility of the output types of the node and the graph.

Source Code 5.10: Static checking of graph `output` method, using concepts and static assertion

```
template<class GraphOutput, class... GraphInputs>
class Graph {
  //...
  template<HedgehogSender Node>
  void output(std::shared_ptr<Node> output) {
    static_assert(
      std::is_base_of_v<behavior::Sender<GraphOutput>, Node>,
      "The output node should have the same output type as the
       ↪ graph.");
    //...
  }
  //...
};
```

5.1.4 Edge connection

The last conformity check is the edge connection. To connect two nodes `x` and `y` in a graph `g`, an edge must be added between them by calling `g.addEdge(x,y)`. The rule to allow connection is the following: (1) the antecedent node `x` needs to be a Hedgehog sender, (2) the successor node `y` needs to be a Hedgehog receiver, and (3) the output type of the antecedent node `x` needs to be part of the input types of the successor node `y`. As shown in Figure 5.6, if we consider a node of type `Node<I2,I5,I6>`, this node is compatible with a node of type `Node<O1,I1,I2,I3>` (left case of figure) but not with a node of type `Node<O1,I1,I4,I3>` is not (right case).

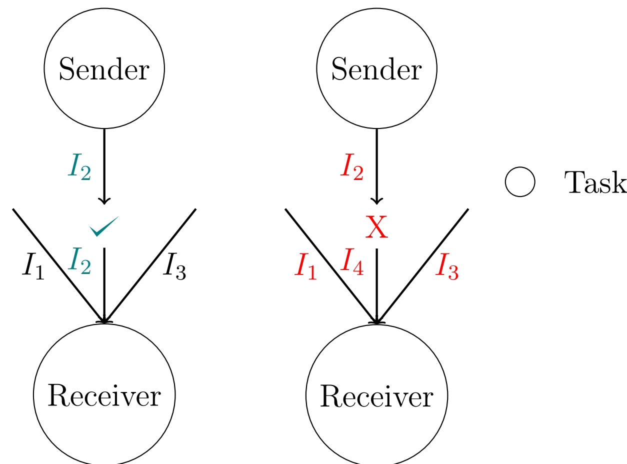


Figure 5.6: Edge conformity check between two nodes

Implementation with template metaprogramming

For the implementation with template metaprogramming, we follow the same logic as previously described and we reuse the traits defined earlier, as shown in Code 5.2.

Source Code 5.11: Static checking of graph `addEdge` method, using template metaprogramming

```
1  template<class GraphOutput, class... GraphInputs>
2  class Graph {
3  //...
4    template<
5      class Sender, class Receiver,
6      class Output = typename Sender::output_t,
7      class Inputs = typename Receiver::inputs_t,
8      class isSender = std::enable_if_t<
9        std::is_base_of_v<behavior::Sender<Output>,Sender>
10     >,
11     class isMultiReceiver = std::enable_if_t<
12       std::is_base_of_v<
13         typename helper::HelperMultiReceiversType<Inputs>::type,
14         Receiver
15       >
16     >
17   >
18   void addEdge(std::shared_ptr<Sender> from,
      ↪  std::shared_ptr<Receiver> to) {
19     static_assert(traits::Contains_v<Output, Inputs>,
20       "The given io cannot be linked to this io: No common types.");
21     //...
```

```
22      }
23    };
```

However, we have chosen to locate the compatibility check of sender's output with receiver's inputs in a static assertion, while the node first filtering step (inheritance check) is made in the template definition. That way in case of failure because of the type compatibility between the sender and the receiver, the output of compilers produces an explicit error message, as show in Figure 5.7

```
#include "hedgehog/hedgehog.h"

class TestTask : public hh::AbstractTask<float, int>{
 public:
  void execute(std::shared_ptr<int> ptr) override {
    this->addResult( output: std::make_shared<float>(*ptr));
  }
};

int main() {
  hh::Graph<float, int> g;
  auto t1 = std::make_shared<TestTask>();
  auto t2 = std::make_shared<TestTask>();
  g.addEdge( from: t1,  to: t2);
}
```

In template: static_assert failed due to requirement 'traits::Contains_v<float, std::tuple<int>>' "The given io cannot be linked to this io: No common types."
error occurred here
in instantiation of function template specialization 'hh::Graph<float, int>::addEdge<TestTask, TestTask, float, std::tuple<int>, void, void>' requested here

Figure 5.7: CLion IDE hint of Hedgehog static checking of edge connection

Implementation with concepts

In Hedgehog v.2, we achieve a similar checking using concepts previously defined in Code 5.6. The types of the sender and receiver nodes of the attempted connection are constrained with `HedgehogMultiReceiver` and `HedgehogSender`, and the sender's output and receiver's inputs compatibility is done with the static assertion described in previous section, as shown in Code 5.12.

Source Code 5.12: Static checking of graph `addEdge` method, using concepts and static assertion

```
1   template<class GraphOutput, class... GraphInputs>
2   class Graph {
3     //...
4     template<
5       HedgehogSender Sender,
6       HedgehogMultiReceiver Receiver>
7     void addEdge(std::shared_ptr<Sender> from,
     ↪   std::shared_ptr<Receiver> to) {
8       using output_t = typename Sender::output_t;
9       using inputs_t = typename Receiver::inputs_t;
10
11      static_assert(traits::Contains_v<output_t, inputs_t>,
12        "The given io cannot be linked to this io: No common types.");
13      //...
14    }
```

```
15    //...
16  };
```

5.1.5 Conclusion

In this section, we presented the main conformity checks that are performed at compile-time by our library when building the graph. They provide some guarantee on the coherence of the resulting graph structure. With them, we are able to secure the use of our library by accepting only types already defined in the library, or specialized by the end-user, and correctly used in a given context.

We also presented different ways of implementing these static checks, that have lead to two versions of the library. It appears that with usual template metaprogramming techniques, the code seems more complicated and the output less helpful for an end-user. At the opposite, with concepts, constraints on types can be expressed more easily, and filtering types is much straightforward. However, we show that for some checking, it is still better to use static assertions, in order to provide a customized error message to the end-user, instead of the one emitted automatically by the compiler.

In Section 5.3, we propose a comparative study about the compilation performance of both approaches (template metaprogramming and concepts).

5.2 Static analysis of data-flow graph

In the previous section, we presented template metaprogramming techniques to secure the use of the library, and guarantee some coherence of the graph structure built by the end-user. These checks are performed at compile-time, when the API is used to construct the graph. In this section, we propose an additional library called HedgehogCX, whose goal is to add a mechanism to analyze a Hedgehog graph structure and give insight on potential errors at compile-time.

For this purpose, constant expressions are used. However, because of the limitations explained in Section 2.4, it was not possible just to add `constexpr` in front of every method of the API. Building the graph is therefore split in two steps, while keeping the API of HedgehogCX as close as possible to the one of Hedgehog. More precisely, two graph representations will coexist, one static and the other one dynamic. The end-user will first specify the data-flow with placeholder nodes (i.e., without knowing their true type). As such, HedgehogCX holds a simple representation with basic information about the graph structure (mainly connections and input/output types).

With this representation, we also propose an extensible mechanism to define and apply complex algorithms to analyze and check for valid and well-formed graphs. Then, if the graph is valid against the series of tests chosen by the end-user, true nodes will be created and associated with the placeholders that will initiate a conversion to generate automatically the final Hedgehog graph, that can be used with the original Hedgehog API.

The class structure added by HedgehogCX is presented in the simplified UML diagram in Figure 5.8. In Section 5.2.1, we briefly discuss the limitations of our approach based on constant expressions. In Section 5.2.2, we present elements of HedgehogCX's architecture that are used to build the first Hedgehog graph (the

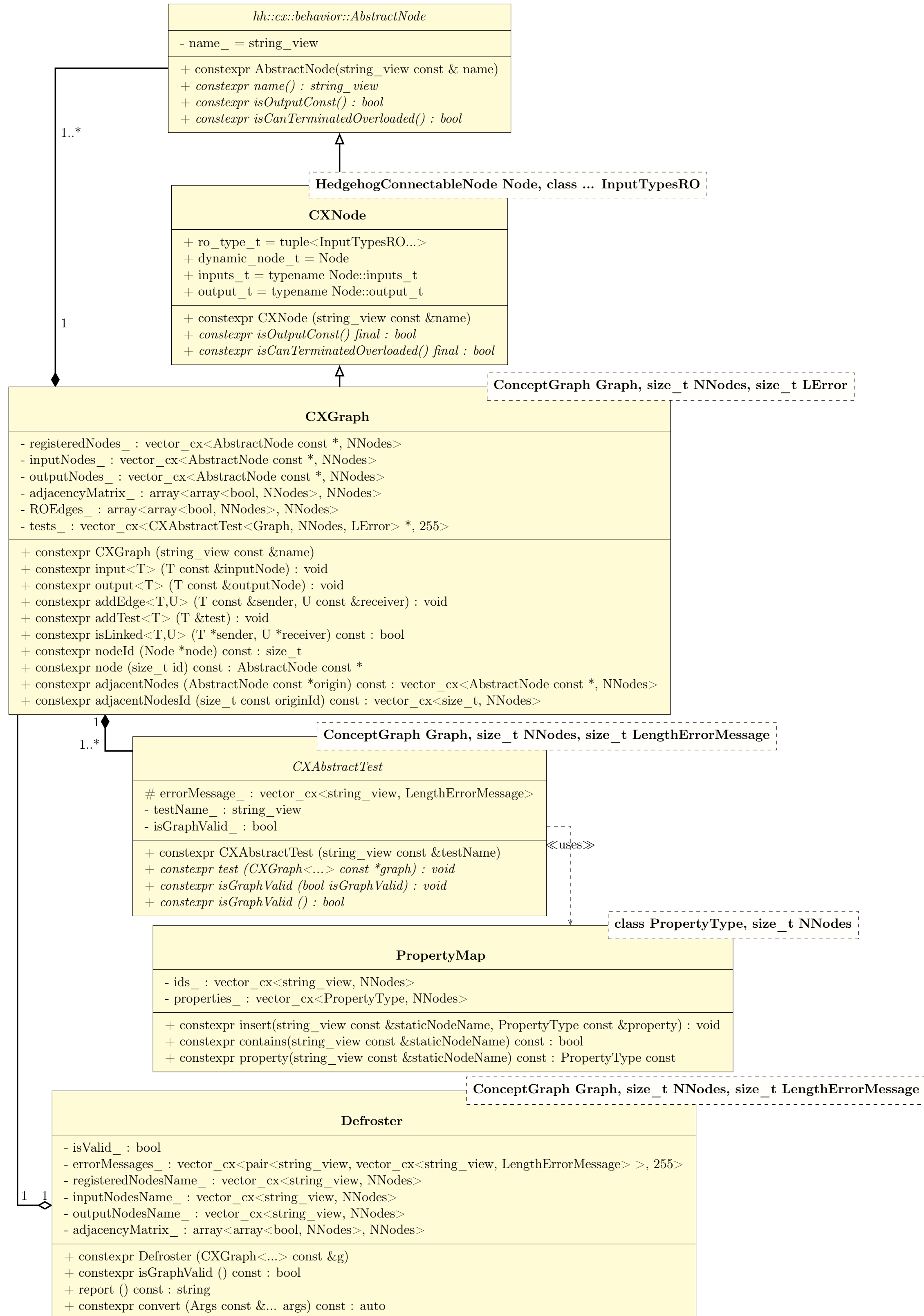


Figure 5.8: Simplified UML class diagram of HedgehogCX

static one), and in Section 5.2.3, those necessary to define and apply tests on the graph. We close in Section 5.2.4 by overviewing the conversion step, from the static graph structure to the full-fledged Hedgehog graph.

5.2.1 Limitations of `constexpr`

To implement the static part of the library, we chose to heavily use constant expressions to express the graph representation and the computation of analysis upon it. The reason is that we do not only handle types, but we wanted to develop and use complex algorithms at compile-time as a user would have written classically in C++ for runtime purpose.

We would have preferred to simply add keyword `constexpr` everywhere it seems necessary in the library, to keep the original Hedgehog API and to enable the execution of any method or function at compile-time. However, this is not possible. In Section 3.6.1, we presented our heavy usage of smart pointers in the Hedgehog API to secure memory. Smart pointers can not be used as `constexpr` function parameters because they are not *literal types* (cf. Section 2.4).

So, instead of modifying the base library and diminishing its overall safety, we preferred to make another library on the side, that respects the limitations of the language and that can be used at will by the end-user, while keeping the original Hedgehog API: two representations of graph will coexist, one with an API for static manipulation close to the original Hedgehog API, and the other with the original Hedgehog API. Notice that this new static step is not mandatory, and an end-user can still build and execute a Hedgehog graph without it.

The other reason that made us choose to use constant expressions is that, even with some limitations, they come up with many capabilities that we need to create our extensible library: virtuality, containers, etc. Notably, it is possible to create `constexpr std::array`, and it is announced in C++ 20 that dynamic containers such as vectors should be available for constant evaluation. The `constexpr` vector would be pretty useful in our case, because it could adapt itself to the number of elements that are added into it, such as the number of nodes in a graph or the number of tests in a graph.

At the time of writing, the `constexpr` vector from the standard library was not available. We chose to reuse an implementation from Jason Turner in his presentation *"constexpr ALL the Things!"* [Deane and Turner, 2017], which was an inspiration for this work. The drawback of this implementation is to be only a wrapper around a `std::array` with a static size. To have an adaptable library, we added template parameter artefacts such as `NNodes` and `LError` used in classes `CXGraph`, `CXAbstractTest`, `Defroster`, and `PropertyMap` (cf. Figure 5.8). These parameters are used to fix the "vector" size with reasonable default values. When the `constexpr` vector from the standard library will be available, we will be able to remove these parameters quite easily, and simplify even further the API and the usage of the library.

5.2.2 Compile-time representation

The first step is to represent statically a Hedgehog graph, using objects of classes `CXNode` and `CXGraph` (cf. Figure 5.8) to construct the graph. By inheritance, a `CXGraph` is a `CXNode` which is an `AbstractNode`. `AbstractNode` is a class used (1)

to allow abstraction and hence composability, which enables storing graph elements of different kinds in the same container (cf. design pattern *composite* [Gamma et al., 1994]); and (2) to set a name for each node, which allows the end-user to identify the nodes in the test reports.

Class `CXNode` is the compile-time counterpart representation of the Hedgehog nodes. It has two template parameters, `Node` that is the type of node it represents (in the original Hedgehog API), and `InputTypesRO` that lists the input types read-only accessed by the node (to allow implementing the data race test for instance). Class `CXGraph` is the compile-time counterpart representation of the Hedgehog graph. It has three template parameters: (1) `Graph` the type of graph it represents (in the original Hedgehog API), and two parameter artefacts, (2) `NNodes` that sets the maximum number of nodes the graph can hold (default is 20), and (3) `LError` that sets the length of error messages (default is 255).

An example of definition of `CXNode` and `CXGraph` objects is shown in Code 5.13.

Source Code 5.13: Definition of static node and graph

```
class TaskIntInt : public hh::AbstractTask<int, int> {
 //...
};

class GraphIntInt : public hh::Graph<int, int> {
 public:
  explicit GraphIntInt(std::string_view const &name)
    : Graph(name) {}
};

constexpr hh::cx::CXNode<TaskIntInt> node("Task1");
hh::cx::CXGraph<GraphIntInt> g("Graph without cycle");
```

Two classes are first defined from the original Hedgehog API to represent user-defined Hedgehog nodes (`TaskIntInt`, lines 1-3) and graphs (`GraphIntInt`, lines 5-9). Then, static nodes and graph - representing respectively `TaskIntInt` and `GraphIntInt` objects - are built.

An internal graph can be represented by a `CXNode` or a `CXGraph` object. In the static representation, internal graphs are black boxes. Inner `CXGraph`s are considered as simple nodes, they are not expanded for the compile-time analysis. Only the inside graph inputs and output are used to check the compatibility when connecting nodes. This choice has been made because when a Hedgehog `Graph` is shared, the internals are not necessarily exposed to the user.

Even if we are not able to use directly the original Hedgehog library for static representation, we have thrived to have the same API to build a graph: `CXGraph` proposes `input`, `output`, and `addEdge` methods. We propose to build the graph presented in Figure 5.9. This graph is interesting because it presents multiple directed cycles and possible data races.

There are three cycles that need to be taken care of between: (1) node 1, node 2, node 3, and node 4; (2) node 2, node 3, node 5, and node 6; and (3) node 2, node 5, and node 6. There are data races possible between: (1) node 3 and node 5, because they receive the same data from node 2; (2) node 4 and node 5, receiving the same data from node 3; and (3) node 7 and node 1, receiving the same data from node 4. For the sake of simplicity, all of the nodes are of type `TaskIntInt` in our example, and the graph of type `GraphIntInt`, as defined in Code 5.13.

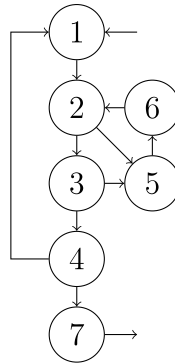


Figure 5.9: Graph example for compile-time analysis

To create this graph, we can write the code presented in Code 5.14.

Source Code 5.14: Graph construction at compile-time

```
// Static nodes
constexpr hh::cx::CXNode<TaskIntInt> node1("Task1");
constexpr hh::cx::CXNode<TaskIntInt> node2("Task2");
constexpr hh::cx::CXNode<TaskIntInt> node3("Task3");
constexpr hh::cx::CXNode<TaskIntInt> node4("Task4");
constexpr hh::cx::CXNode<TaskIntInt> node5("Task5");
constexpr hh::cx::CXNode<TaskIntInt> node6("Task6");
constexpr hh::cx::CXNode<TaskIntInt> node7("Task7");

// "constexpr" context
constexpr auto defroster = [&]() {
  hh::cx::CXGraph<GraphIntInt> g(
    "Graph with multiple cycles and data races");
  // Graph connections
  g.input(node1);
  g.addEdge(node1, node2); g.addEdge(node2, node3);
  g.addEdge(node3, node4); g.addEdge(node4, node7);
  g.addEdge(node4, node1); g.addEdge(node3, node5);
  g.addEdge(node5, node6); g.addEdge(node6, node2);
  g.addEdge(node2, node5); g.output(node7);
  //...
}();
```

In this code, the definition of each `CXNode` object is specified with `constexpr`, allowing them to be used at compile-time (lines 2-8). But, we cannot do the same for the graph object. `constexpr` implies `const`. So if we specify `constexpr` for the graph definition, we are not able to call non constant methods of this instance, and thus not able to connect nodes.

To be able to build the graph, we need to encapsulate the graph definition and its construction (and later the tests attached to it) in a "`constexpr` context": a `constexpr` lambda here (lines 11-27). A lambda tagged with `constexpr` can be evaluated at compile-time only if its arguments and the computation inside can also be evaluated at compile-time. All `CXGraph` methods used for the construction are marked with `constexpr`, which is why we can build our graph at compile-time that way. Notice that the nodes could have been defined inside this `constexpr` context, but we will see later that they need to be used from outside.

5.2.3 Compile-time tests

Once the graph is built, we can analyze its structure. Two tests are proposed by the library: to detect directed cycles and to detect possible data races in the graph. A test is represented by a class, and Code 5.15 shows how to use the two tests of classes `CycleTest` and `DataRaceTest` provided by the library.

Source Code 5.15: Tests addition for compile-time analysis

```
// Static nodes
//...

constexpr auto defroster = [&]() {
  hh::cx::CXGraph<GraphIntInt> g("Graph with multiple cycles and
   ↪  data races");

  // Graph connections
  //...

  auto cycle = hh::cx::test::CycleTest<GraphIntInt>{};
  auto datarace = hh::cx::test::DataRaceTest<GraphIntInt>{};

  g.addTest(cycle);
  g.addTest(datarace);
  //...
}();
```

The graph tests are first instantiated at lines 10-11. Their class needs the type of the dynamic graph as the template parameter, like class `CXGraph` does. The test objects are then added to the static graph at lines 13-14 to be later executed (at compile-time).

Built-in tests

For `CycleTest`, a deadlock can occur in a Hedgehog graph if there is a directed cycle in the graph. To fix the deadlock due to a cycle, the `canTerminate` method needs to be overridden in a node of the cycle (cf. Section 3.5.1). To detect this kind of deadlock, the test searches for connected components (Tarjan's Algorithm [Tarjan, 1972]), finds *simple* directed cycles (Johnson's Algorithm [Johnson, 1975]), and removes the cycles in which a Hedgehog node overrides the `canTerminate` method (checked by template metaprogramming). This test is not enough to prevent any deadlock, as we would need to understand the computation, the flow of data, and the termination condition in the `canTerminate` method. This test only checks for the found cycles if at least a node has the `canTerminate` method redefined, showing that the end-user has taken some action to break the cycle and hence the deadlock.

For `DataRaceTest`, a data race can occur if mutable data are broadcast to more than one node. There are two ways to prevent such data races: (1) broadcast `const` data (e.g., `class TaskCIntCInt : public hh::AbstractTask<int const, int const>`) or (2) declare an input type as read-only. That is why our `CXNode` has a template parameter pack (`InputTypesRO`) as second parameter. This pack lists the input types that are declared read-only. For instance, in definition `constexpr hh::cx::CXNode<TaskIntInt, int> node1("Task1")`, `node1` represents a task with `int` as input and `int` as output (`TaskIntInt`), and declares that data of type `int` it receives are only used in a read-only fashion. Therefore, to detect data races, the test traverses the graph to find the broadcast cases, keeps the ones with non-constant

output, and removes the remaining ones in which all receiver nodes declare this type as read-only.

User-defined tests

Our API allows end-users to create their own tests. As shown in Figure 5.8, we have an abstract class `CXAbstractTest` that represents a static test on a graph structure (both `CycleTest` and `DataRaceTest` inherit from it) and a `PropertyMap` class to associate a property with each node if the algorithm requires it.

With this design, any developer can create a new test by defining a subclass of `CXAbstractTest` and implementing the abstract method `test` with the algorithm, as shown in Code 5.16 that finds the critical path of a graph. The code in `test` needs to indicate whether the graph passes the test (by calling method `isGraphValid` of the graph with the answer, cf. line 17), and the error message if not passed (by feeding the attribute `errorMessage_`, cf. lines 19-25) of the graph. The problem of creating the error message can be summed up as string manipulation in a `constexpr` function. `std::string` objects are not available in a `constexpr` function yet, so we built the error message by adding to a vector (attribute `errorMessage_` of class `CXGraph`) different `std::string_view` objects that can be manipulated in a `constexpr` context. Once the `constexpr std::string` becomes available, it shall replace this vector of `std::string_view`.

This test does not only count the number of nodes in a path to determine its length, but uses some weight of each node, which could for instance represent its estimated execution time. This weight does not exist in class `CXNode`, and we cannot anticipate the needs of any algorithm. This is why we added class `PropertyMap` to associate at will any information with a node, as seen in lines 34-43. One just needs to declare a new property for nodes, by defining a PropertyMap object, and then associate a value for this property with each node. The property map is then provided to the `CriticalPathTest` object at construction, to allow it to access the properties (line 45). The graph is provided as a parameter to the `test` method where the algorithm is implemented. For searching the critical path, each node is visited (lines 13-15). In any way, the graph will not pass the test because there is always a critical path (line 17), and the error message is composed with the path found (lines 19-23).

Source Code 5.16: Partial code of the critical path test

```
1  // Definition of the test
2  template<class Graph, size_t NNodes = 20>
3  class CriticalPathTest
4  : public hh::cx::CXAbstractTest<Graph, NNodes> {
5   public:
6    constexpr explicit CriticalPathTest(PropertyMap<double, NNodes>
     ↪  const &propertyMap)
7      : hh::cx::CXAbstractTest<Graph, NNodes>("Critical Path"),
8        propertyMap_(propertyMap) {}
9
10   constexpr ~CriticalPathTest() override = default;
11
12   constexpr void test(hh::cx::CXGraph<Graph, NNodes> const *graph)
     ↪  override {
13     for (auto inputNode : graph->inputNodes()) {
14       visitNode(inputNode);
15     }
```

```
16
17      this->isGraphValid(false);
18
19      this->errorMessage_.push_back("The critical path is:\n\t");
20      for (auto node : criticalVector_) {
21        this->errorMessage_.push_back(node->name());
22        this->errorMessage_.push_back(" -> ");
23      }
24
25      this->errorMessage_.pop_back();
26    }
27  //...
28  };
29
30  // Usage of the test
31  int main (){
32    //...
33    constexpr auto defroster = [&]() {
34      PropertyMap<double> propertyMap;
35
36      propertyMap.insert("Task0", 4.9);
37      propertyMap.insert("Task1", 17.3);
38      propertyMap.insert("Task2", 49);
39      propertyMap.insert("Task3", 47);
40      propertyMap.insert("Task4", 9.5);
41      propertyMap.insert("Task5", 1);
42      propertyMap.insert("Task6", 12.4);
43      propertyMap.insert("Task7", 1);
44
45      auto criticalPath = CriticalPathTest<GraphIntInt>(propertyMap);
46      //...
47    }();
48    //...
49  }
```

Tests execution

Once the tests are defined and added to the graph in the `constexpr` context, they are executed at the construction of the `defroster`, the object that will help pass from the static graph structure to the dynamic Hedgehog one, as shown in Code 5.17 at line 16.

Source Code 5.17: Tests execution and "defrosting"

```
1   // Static nodes
2   //...
3
4   constexpr auto defroster = [&]() {
5     hh::cx::CXGraph<GraphIntInt> g("Graph with multiple cycles and
      ↪  data races");
6
7     // Graph connections
8     //...
9
10    auto cycle = hh::cx::test::CycleTest<GraphIntInt>{};
11    auto datarace = hh::cx::test::DataRaceTest<GraphIntInt>{};
12
13    g.addTest(cycle);
14    g.addTest(datarace);
15
16    return hh::cx::Defroster(g);
17  }();
18
19  static_assert(defroster.isGraphValid(), "The Graph is not valid.");
20
21  // OR
22
```

```
23 if constexpr(!defroster.isGraphValid())
24   std::cout << defroster.report() << "\n";
25 else {/*...*/}
```

During construction, the `Defroster` object extracts the graph structure and runs the tests. The static graph holds pointers to its internal nodes. The address space at compile-time (i.e., used to execute constant expressions) is not the same as the one used at runtime. It is not possible to use the same pointer at compile-time and runtime. As such, an intermediate graph representation, free of pointers, is necessary. For this purpose, we need an identifier for each node, the `name_` attribute defined in `AbstractNode` class. The `Defroster` object contains an internal representation of the graph where nodes are referenced by their name only in the data structures, as shown in Figure 5.8.

At construction, the `Defroster` instance also runs the tests, and can provide then information about whether the graph passed all the tests or not (call to method `isGraphValid` at lines 19 and 23), and a full report which is the concatenation in a `std::string` of all the test reports (call to method `report()` at line 24).

We can determine at compile-time if the graph passes its tests, and so, with a `static_assert` stop the compilation with an error message, or with a `constexpr if` change the execution depending on the test result. Then, if the graph is not valid, we can print at runtime the full report instead of building the ultimate Hedgehog graph and starting its execution. For example, for the graph presented in Figure 5.9, with the two tests provided by the library, the report contains:

```
In graph Graph with multiple cycles:
Johnson: Cycles found, the canTerminate() method needs to be defined
for each of these cycles.
 Task1  ->  Task2  ->  Task3  ->  Task4  ->  Task1
 Task2  ->  Task3  ->  Task5  ->  Task6  ->  Task2
 Task2  ->  Task5  ->  Task6  ->  Task2
Data races test: Potential data races found between these nodes:
 Task2 -> Task3 / Task5
 Task3 -> Task4 / Task5
 Task4 -> Task1 / Task7
```

5.2.4 From compile-time to runtime

The final purpose of a `Defroster` object is to construct the Hedgehog graph that will ultimately be executed from its internal representation. This is achieved with the `convert` method that needs pairs of static and runtime nodes as shown in Code 5.18. The runtime nodes are first instantiated (lines 2-9), and then are mapped to the static ones (lines 13-20). Once created (line 12), the graph can be used as any usual Hedgehog graph (lines 24-26).

Source Code 5.18: Conversion from static to runtime graph

```
1 // Runtime nodes
2 auto task0 = std::make_shared<TaskIntInt>("Task0");
3 auto task1 = std::make_shared<TaskIntInt>("Task1");
4 auto task2 = std::make_shared<TaskIntInt>("Task2");
5 auto task3 = std::make_shared<TaskIntInt>("Task3");
```

```
auto task4 = std::make_shared<TaskIntInt>("Task4");
auto task5 = std::make_shared<TaskIntInt>("Task5");
auto task6 = std::make_shared<TaskIntInt>("Task6");
auto task7 = std::make_shared<TaskIntInt>("Task7");

// Runtime graph generation
auto graph = defroster.convert(
  node0, task0, // Pair #1
  node1, task1, // Pair #2
  node2, task2, // Pair #3
  node3, task3, // Pair #4
  node4, task4, // Pair #5
  node5, task5, // Pair #6
  node6, task6, // Pair #7
  node7, task7  // Pair #8
);

// Graph execution
graph->executeGraph();
graph->finishPushingData();
graph->waitForTermination();
```

The `convert` method internally first validates each pair of nodes `(x,y)`, by checking if the node type that the static node `x` represents is the same as the Hedgehog node `y`. Once validated, the list of nodes is then split into two tuples, one with the static nodes and the other with the runtime ones. The graph is generated; for this purpose, the tuples need to be transformed into vectors of `std::variant`. We have one vector per tuple and each variant holds one node (a variant can hold a data of one of its alternative types, here all the types provided to method `convert`). This enables storing in a dynamic data structure all the nodes.

To create the dynamic graph we need to parse the graph structure (more precisely the adjacency matrix and the vectors of input and output nodes) held in the `Defroster` object, and each time a connection is necessary, to find the associated runtime nodes that can be identified by their names only. This requires us to parse the tuple of static nodes with an index, but there is no dynamic way of accessing the element at position `I` in such a structure, the only possible way is the `std::get<I>` function, where `I` is a static information.

Once the vectors are constructed, the defroster visits (using `std::visit`) its data structures containing the graph structure (vectors of input and output nodes, adjacency matrix), and searches the node(s) concerned by the connection (using the last generated vectors); we apply this connection into the Hedgehog graph.

5.3 Experiments and results

As we heavily use metaprogramming techniques that operate at compile-time, we propose various experiments to study the performance of the compilation phase of our system. All the presented results are statistics obtained from 20 repetitions of the same execution (that is just compilation). They were conducted on an Intel Core i7-10700 with 8 cores at 2.90 GHz (with turbo boost activated, up to 4.6 GHz[1]) and 2 × 8GB of DDR4 RAM at 2666 MHz, using compilers GCC v.10.3 and Clang v.10.0, libraries OpenBLAS v.0.3.8 and LAPACK v.3.9.0, and Ubuntu 20.04 LTS (GNU/Linux 5.8.0-63-generic x86_64).

[1]Compilation done with affinity set to one core only, so maximum frequency was triggered.

The tests concern the compilation of matrix multiplication and LU decomposition (without partial pivoting) algorithms to measure the performance of Hedgehog, and some fictitious data-flow graphs, which structure is described later, to measure the performance of HedgehogCX. Hedgehog matrix multiplication is from Hedgehog tutorial 3 [2]. Hedgehog LU decomposition is from HMBLib (cf. Section 4.4.1). HTGS matrix multiplication is a custom implementation that is as close as possible to the Hedgehog version. HTGS LU decomposition is from HTGS tutorial 6 [3]. They are all CPU-only to ease the requirements on the target environment we use for running the tests. There is no difference in the implementation of the tests between Hedgehog v.1 (with template metaprogramming) and v.2 (with constant expressions). When not stipulated, the additional HedgehogCX library is not used.

We first study in Section 5.3.1 the compilation time of HTGS, Hedgehog v.1, and Hedgehog v.2 for both matrix multiplication and LU decomposition. Then, in Section 5.3.2, we measure the cost of the Hedgehog conformity checks for the same algorithms. Finally, in Section 5.3.3, we delimit the possibility of our HedgehogCX implementation and its performance.

5.3.1 Compilation performance

Figure 5.10 shows the compilation times for the matrix multiplication and LU decomposition algorithms, using GCC and Clang, and with HTGS and both versions of Hedgehog. The full compilation process has been measured here.

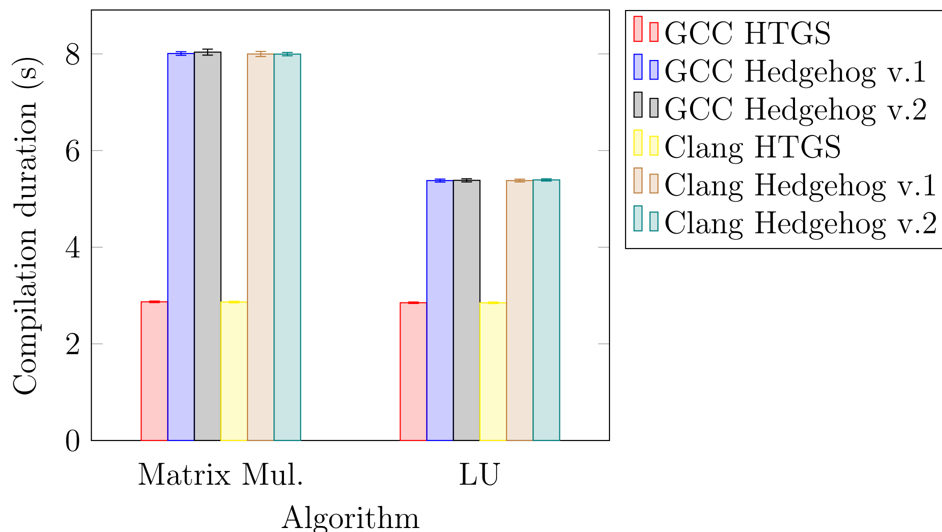


Figure 5.10: Compilation times of HTGS and both Hedgehog versions, for matrix multiplication and LU decomposition

We observe that compilation with Hedgehog is around 3 times slower than with HTGS for both algorithms using GCC, and around 2 times slower using Clang. The compilation times between Hedgehog v.1 and v.2 are approximately the same, so we do not see any impact of using concepts instead of usual template metaprogramming techniques.

[2]https://pages.nist.gov/hedgehog-Tutorials/tutorials/tutorial3.html
[3]https://github.com/usnistgov/HTGS-Tutorials/tree/master/tutorial6

We propose to investigate further the compilations times by using the profiling statistics provided by the compiler. A detailed report can be output by GCC by adding the `-ftime-report` flag when invoking the compiler. We only show the report of GCC even if Clang has the same option, because GCC proposes an output simpler to parse and understand. More notably, it provides a first macroscopic view of the compilations stages, that we refer to as *phases* here. There are 9 phases, but we only show here the three that present significant times. Reports also offer a view with more detailed stages, that we refer to as *steps* here. There are more than 60 steps, some of them being achieved concurrently, but we choose to show only the most significant ones (those that, together, contribute to most of the compilation time). Notably, we will focus on step *template instantiation* first, and later on step *constant expression evaluation*. Unfortunately, to our knowledge, these is no precise documentation on what the phases and steps precisely refer to, however we think that the reports can provide us some insight.

From the profiling statistics of GCC on the same experiments, Figures 5.11 and 5.12 presents the time spent by the compiler is some phases and steps respectively. The first figure show that mainly phases "lang. deferred" and "opt and generate" are impacted, suggesting that the call graph used by the compiler is longer to generate and to analyze, more template functions specializations need to be instantiated, and/or a larger virtual table need to be created.

The second figure shows that all the presented steps (that have the most significant times here) are equally impacted, and not only the *template instantiation* step as we could have expected. Notice that steps *overload resolution* and *name lockup* are concurrent steps.

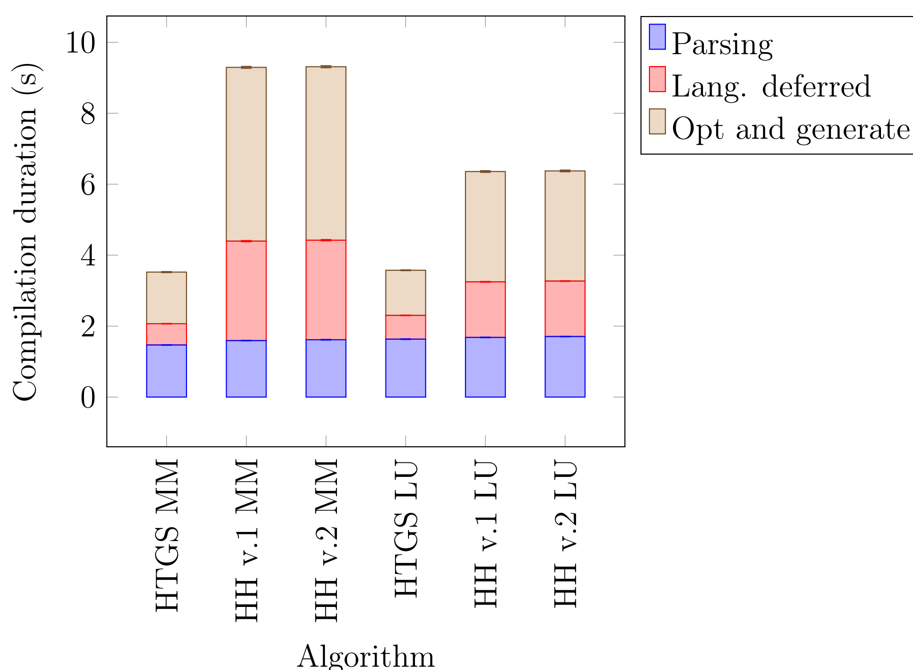


Figure 5.11: Phase compilation times of HTGS and both Hedgehog versions, for matrix multiplication and LU decomposition

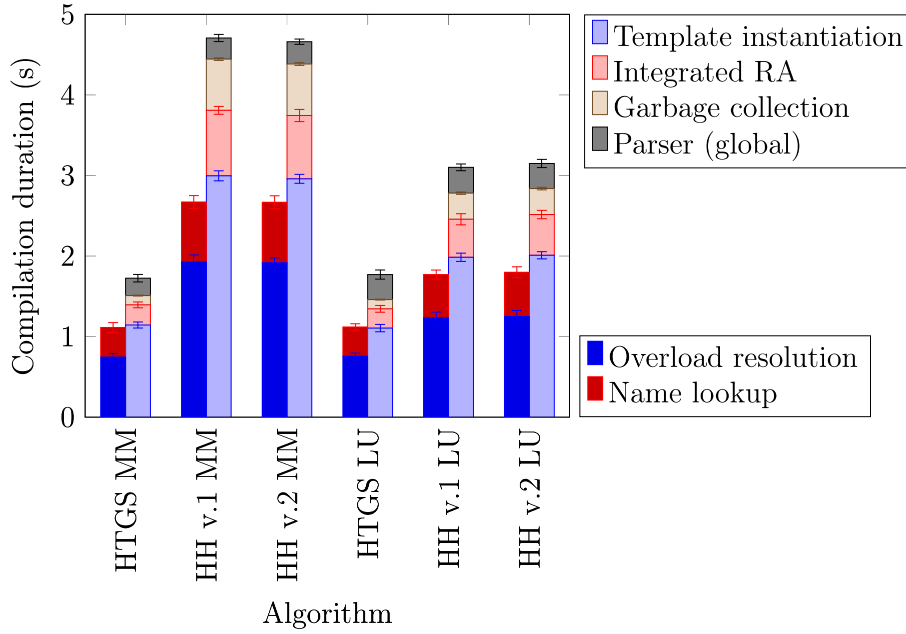


Figure 5.12: Step compilation times of HTGS and both Hedgehog versions, for matrix multiplication and LU decomposition

5.3.2 Overhead of conformity checking

In this section, we measure the impact of the conformity checking that is done at compile-time, when building the graph structure. For this purpose, we measure the performance of Hedgehog with static checking removed from methods `input`, `output`, and `addEdge` of class `Graph` that are the only methods used in our tests to build the graphs. This version of Hedgehog is referred to as "w/o check". The results are shown in Figure 5.13. We see that there are no benefits or penalties in terms of compilation duration between the runs with and without compile-time conformity checking.

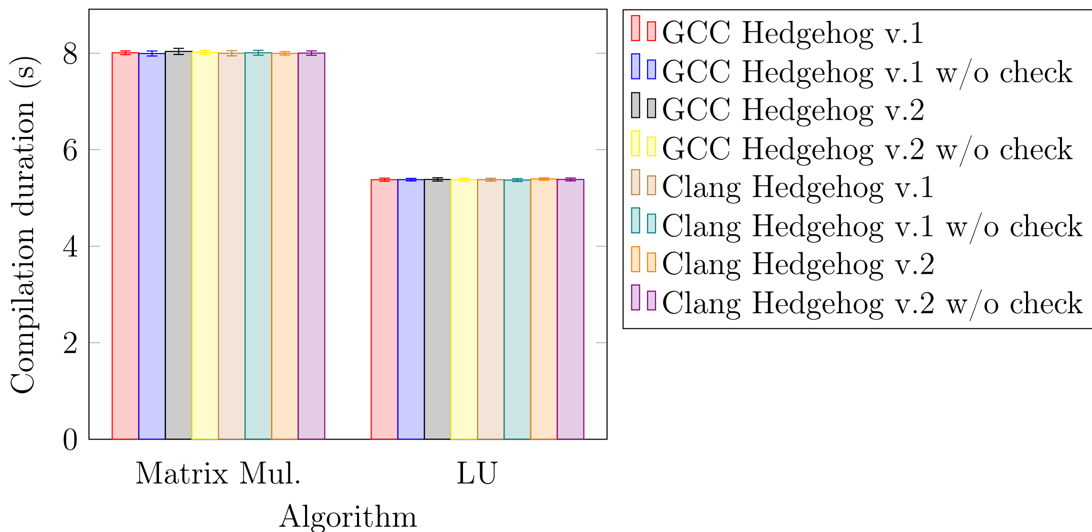


Figure 5.13: Impact of static conformity checking in both Hedgehog versions, for matrix multiplication and LU decomposition

5.3.3 Overhead of graph analysis

In this section, we study how the use of HedgehogCX to achieve the construction of a static graph and its analysis at compile-time impacts the overall compilation times. We consider the same algorithms, matrix multiplication and LU decomposition, and compare the compilation times when (1) a Hedgehog graph is directly built and run, and (2) a static graph is first built, then the two built-in static tests for data races and deadlocks detection are performed, and the conversion from the static graph to the final Hedgehog graph is done and the graph run (because tests will pass). The second configuration is referred to as "w/ static". Only Hedgehog v.2 is concerned by this experiment, because it is the only version compatible with HedgehogCX. The results are presented in Figure 5.14.

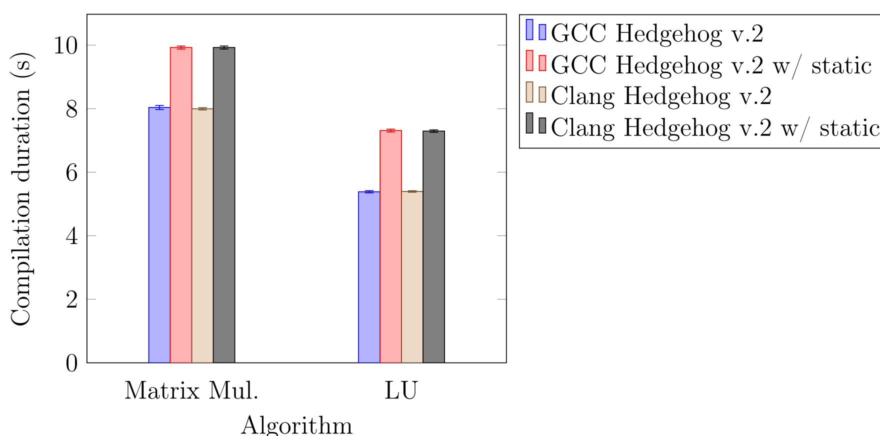


Figure 5.14: Impact of the static analysis library on the compilation duration, for matrix multiplication and LU decomposition

Between 25% to 35% increase of the compilation duration can be seen with the static analysis of the graph. To understand more precisely what happens, we propose to profile the compilation. Figures 5.15 and 5.16 show, for matrix multiplication and LU decomposition, the time GCC passes on each of its phases and steps for executing the static analysis.

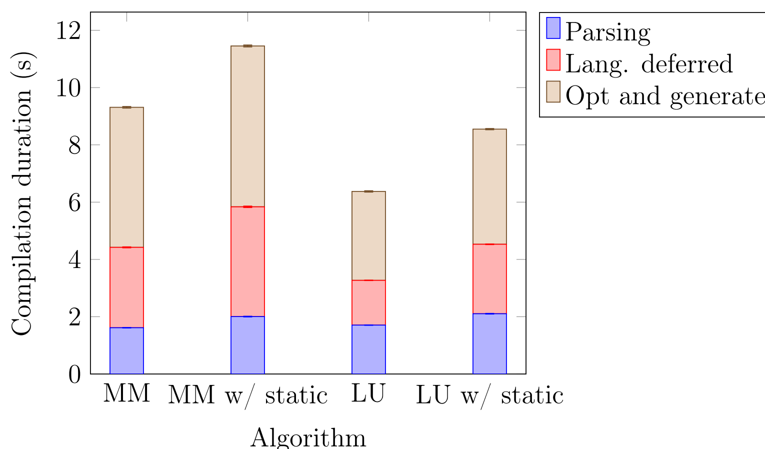


Figure 5.15: Phase compilation times of the static analysis, for matrix multiplication and LU decomposition

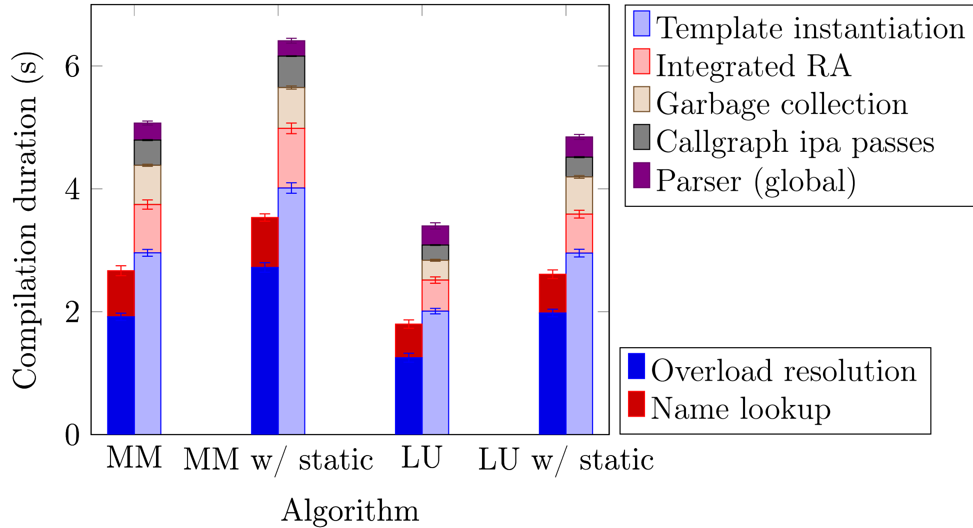


Figure 5.16: Step compilation times of the static analysis, for matrix multiplication and LU decomposition

Similarly to Figure 5.11, we observe that phases *lang. deferred* and *opt and generate* are impacted, $\approx +45\%$ and $\approx +22\%$ respectively, conducting to the same analysis. The parsing phase also shows a compilation time increase of $\approx 24\%$, indicating that the base code parsed by the compiler is bigger.

For compilation steps, the *overload resolution* has the largest increase ($\approx +50\%$), followed by the template instantiation ($\approx +41\%$). We also noticed an increase of *garbage collection* ($\approx +45\%$) for LU decomposition. These numbers indicate that the compiler uses more time to perform name lookup because of our heavy usage of templates in HedgehogCX (confirmed by the increase in the *template instantiation* step).

In addition to the tests with matrix multiplication and LU decomposition, we propose to study the impact of the compile-time analysis of graph structures of different kinds and sizes. The first kind of graphs are *path* graphs, having n nodes, each connected to the next one by an edge, and with an extra edge between the last node and the first one, which results in n edges as shown in Figure 5.17, forming a *simple* directed cycle implicating all the nodes.

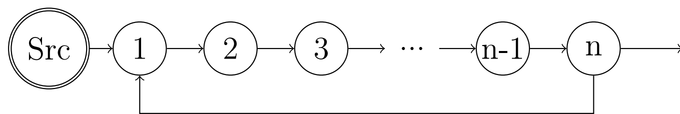


Figure 5.17: Path graph structure

The second kind of graphs are random *connected* (i.e., each node is connected at least with another node) graphs with n nodes and $2 \times n + 1$ edges, and with possibly directed cycles, produced by a graph random generator [4].

We first experiment with the two kinds of graphs with different sizes for two built-in test algorithms (data races and directed cycles detection) and measure the number of constant expression operations needed to perform the compile-time tests. Compilers have an option to limit the number of *constant expression operations*

[4] *Build Graph* module of B++ Library: `https://perso.isima.fr/bachelet/build_graph`

allowed to be performed by a constant expression context at compile-time. With a dichotomic search that repeats experiments by changing the limit, we were able to find the lowest value for this limit, which is the value presented in the following figures.

We use `constexpr-ops-limit` and `constexpr-steps` flags for the `GCC` and `Clang` compilers respectively. These compilers have different definitions of what a *constant expression operation* is. The flag of GCC is defined as: *"Set the maximum number of operations during a single constexpr evaluation. Even when number of iterations of a single loop is limited with the above limit, if there are several nested loops and each of them has many iterations but still smaller than the above limit, or if in a body of some loop or even outside of a loop too many expressions need to be evaluated, the resulting constexpr evaluation might take too long"* [5]. The second (`constexpr-steps`) is defined by Clang as: *"Sets the limit for the number of full-expressions evaluated in a single constant expression evaluation"* [6].

Figures 5.18 and 5.19 present the number of constant expression operations performed by both compilers for path and random graphs respectively. As the definition is different, we propose relative values v_n where each measure m_n (that is the number of operations of a compiler for a graph of size n) is divided by the first measure for a graph of size 2: $v_n = m_n/m_2$.

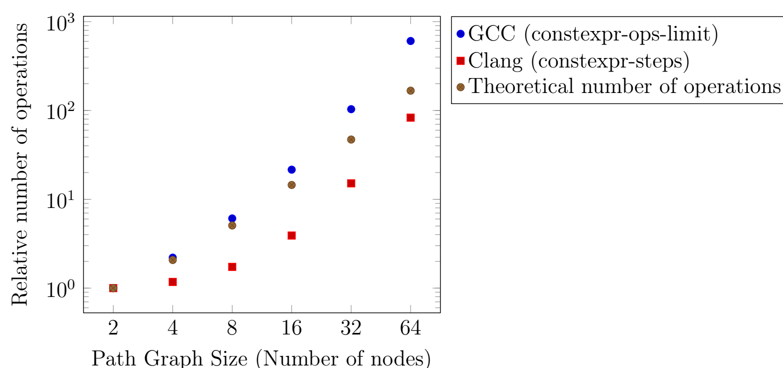


Figure 5.18: Relative number of constant expression operations, depending on the path graph size

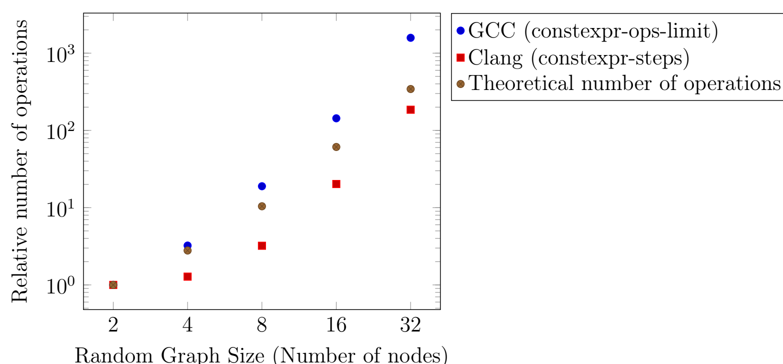


Figure 5.19: Relative number of constant expression operations, depending on the random graph size

[5] https://gcc.gnu.org/onlinedocs/gcc/C_002b_002b-Dialect-Options.html
[6] https://clang.llvm.org/docs/UsersManual.html

In the figures, the theoretical number of operations is also shown. It is computed from the complexity of both test algorithms. The data race test has a complexity of $\mathcal{O}(|V| \times |E|)$. The directed cycles detection is a three steps algorithm: (1) Tarjan's algorithm ($\mathcal{O}(|V| + |E|)$), (2) Johnson's algorithm ($\mathcal{O}((V + E)(C + 1))$), and (3) a filtering ($\mathcal{O}(V)$). V is the number of vertices, E is the number of edges, and C the number of cycles in the graph.

In the path graph case, $C = 1$ and $E = V + 2$. The global complexity of running both tests is $\mathcal{O}(V^2 + 9V + 6) = \mathcal{O}(V^2)$. In the random graph case, $V = 2 \times E + 1$ and we have used the results of the library to determine C. The global complexity of running both tests for a random graph is $\mathcal{O}(2V^2 + 3CV + 8V + C + 2) = \mathcal{O}(V^2)$.

Note here that the number of constant expression operations measured by the compilers includes the graph construction, the tests, the report constructions, and the creation and use of the defroster. In Figures 5.20 and 5.21, we see the time GCC passes on each of its phases for executing the static analysis of graphs of different sizes, for path and random graphs respectively.

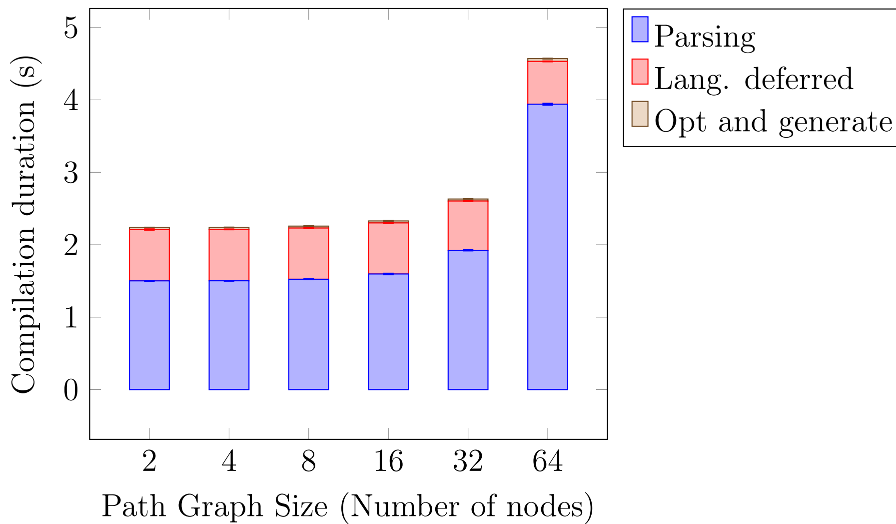


Figure 5.20: Phase compilation times for analyzing path graphs of different sizes

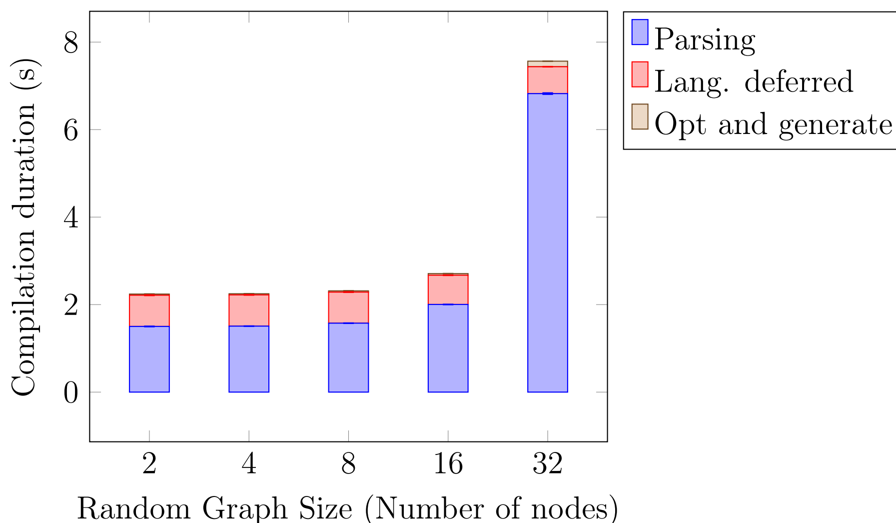


Figure 5.21: Phase compilation times for analyzing random graphs of different sizes

Among the different phases, only one grows in comparison of the others depending on the graph size: the parsing phase. Because only the number of constant expression operations increases significantly in function of the number of graph nodes, either the constant expressions are evaluated during this phase or the constant expressions lead to more parsing.

The compiler's report gives us a more precise view on what is happening. The five most significant steps are presented in Figures 5.22 and 5.23. GCC reports a step for the evaluation of constant expressions that follows the same trend as the number of constant expression operations presented in Figures 5.18 and Figure 5.19. The other steps are stable showing that there is no other impact on compilation when increasing the number of nodes in the graph.

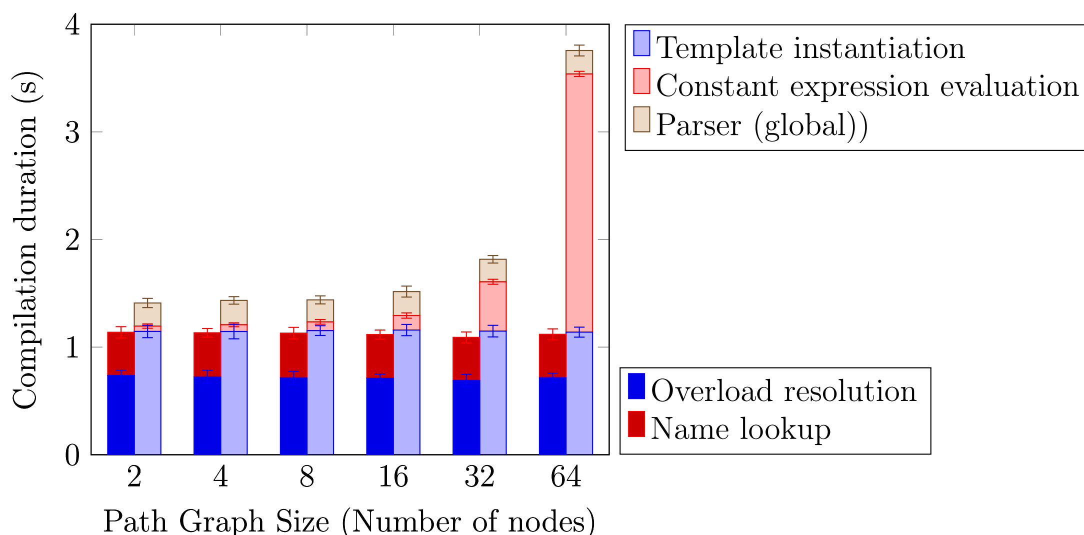


Figure 5.22: Step compilation times for analyzing path graphs of different sizes

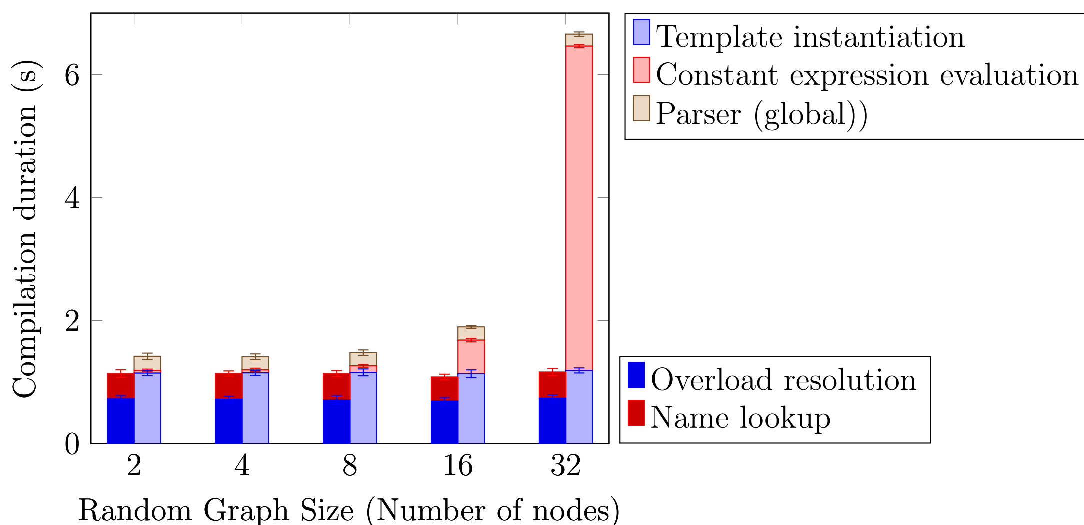


Figure 5.23: Step compilation times for analyzing random graphs of different sizes

5.4 Conclusion

In this section, we presented two ways of using metaprogramming techniques in Hedgehog. We chose to use compile-time language features to give hints to the developer about design problems while he or she is developing (while writing code if the IDE is advanced enough) and during the compilation process.

The first way is the usage of template metaprogramming techniques or concepts (depending on the library version) inside the runtime API to secure its use, mainly to only accept the rightful types when the user extends classes from the API or while building the data-flow graph by connecting nodes. The concept version allows having a more straightforward code with clearer error messages served to the user in conjunction with static assertions.

The second way is the usage of constant expressions to build a full compile-time library that provides an API to build a simplified data structure of the data-flow graph, to test properties of the graph, to change the compilation outcome or the final dynamic code produced depending on the test results, and to automatically convert this static graph representation into a full-fledged dynamic Hedgehog graph. This new C++ feature allows creating compile-time code that is easily extensible by any end-user through the use of only inheritance and virtuality. This property combined with our explicit abstraction makes compile-time computation more accessible to people than with template metaprogramming techniques.

Our experiments show first that there is a negative impact (2 to 3$\times$) of using Hedgehog over HTGS when considering compilation duration. From the analysis made from the compilations profiling, we have determined that this increase is due to a longer time to generate and analyze the call graph, to instantiate more templates, and/or to create a larger virtual table. This overhead seems acceptable compared to the advantages offered by Hedgehog, extensibility and static checking mainly.

Second, the experiments reveal no impact in terms of compilation performance by adding conformity checking, using either concepts or template metaprogramming (they show equivalent performance). Third, using our compilation-time library to check the data-flow graph structure statically, we have seen a 25% to 35% increase of the compilation time for graphs up to 64 nodes, while only impacting the constant expression evaluation step of the compilation process. This increase seems to follow the complexity of the checking algorithms used, and again, the advantage of being able to execute complex analysis algorithms at compile-time should worth the overhead.

Conclusion

In this manuscript, we presented the design of the Hedgehog parallel library, the next iteration of the HTGS library developed at NIST. We proposed a general-purpose solution offering an explicit and accessible model with intrinsic parallelism and advanced composability options. We did not find all these properties in the existing approaches we examined, from domain-specific parallel libraries to general-purpose parallel models or languages. We target coarse-grained parallel implementation of algorithms on an heterogeneous single powerful computational node using a task-based approach with a graph representation. It allows one to simply represent the steps of a complex algorithm with nodes and the flow of data between them with edges. Our runtime strategy is different from the usual task graph solutions as it relies exclusively on the operating system to schedule the threads, which are bound to the same graph nodes from start to finish of the algorithm execution. This leads to a simpler library as there is no need for an extra scheduler or an extra balancing algorithm.

To get performance, Hedgehog relies on the data pipelining inherent in the model to overlap communication delays between the different devices. However, we also rely on the developer to adapt the data decomposition strategy feeding the graph specifically for the hardware. For this purpose, we give the developer the keys to clearly assess the performance of all parts of his/her algorithm. First, the same model is used from the early design to the execution, and we propose different visual tools to help him/her debug, understand how the computation is exactly conducted, and then discern where the critical path is to improve the overall algorithm performance. Our feedback solution appeared to be statistically costless by collecting measures at the node level. We have also integrated the NVIDIA tagging library NVTX to propose a time-series graph presenting the life cycle of each graph node.

Hedgehog is also amenable to create libraries. We have been able to create FastLoader and HMBLib, image processing and linear algebra libraries. With a latency between $1\,\mu$s and $10\,\mu$s, we were able to achieve $> 85\%$ of theoretical peak performance on an heterogeneous node with 4 GPUs for an implementation of the multiplication of matrices with $64k \times 64k$ elements. For our CPU-only implementation of the LU decomposition with partial pivoting, we have shown the importance of data decomposition to get performance on par with other parallel libraries. Moreover, Hedgehog presents a streaming aspect allowing, in the case of chained graphs (for example LU decomposition or matrix multiplication pushing data to a second graph), to start the next graph's computation more than 40 times faster than waiting the complete result of the first graph to start the second one.

This library is made possible thanks to compile-time capabilities available in the C++ language. Using template metaprogramming techniques combined with concepts (a recent addition to the language), we were able to secure our library by only accepting user-defined objects (necessary for extensibility) that correspond to

what we are expecting at compilation, and node connections that conform to their respective inputs and outputs. This checking is meant to detect issues at an early stage and avoid failures when at executing of the task graph. Some IDEs are capable of analyzing the compiler's output to offer feedbacks during development. Experiments reveal that this static checking brings no significant overhead at compile-time.

Even with this safety that ensures some coherence of the graph structure, other issues remain possible. Hedgehog accepts directed cycles and proposes by default for each of its nodes to broadcast their output. These features can lead to deadlocks or data races. Even if the developer has possibilities with the library to solve these problems, there is no guarantee that he/she will always use them, as deadlocks and data races can be hard to detect. Therefore, we have proposed a side library, HedgehogCX, that is based on constant expressions, another recent feature of the C++ language that allows manipulating complex data structures and algorithms at compile-time as easily, and with the same language, as usual code for runtime. This library allows the analysis of a graph structure at compile-time to detect potential issues. In case of problems, the compilation can be stopped and/or a report can be produced. The developer can create his/her own tests for compile-time using a fully extensible API. Experiments with data race and deadlock detection show that the compilation times follow the complexity of the checking algorithms used and seem acceptable compared with the benefit of more safety.

Future work

Several improvements are currently under investigation to bring new capabilities to Hedgehog. Some are motivated by increasing the expressiveness of the library, others focus on targeting new hardware configurations, or adding algorithms for analyzing or transforming the data-flow graph.

First, we are considering multiple output capabilities to the nodes. This new feature should greatly increase the expressiveness of the library. It allows one to do some branching depending on the type of output produced. With the current implementation, we can choose to which nodes a piece of data is sent with multiple layers of state managers. The problem is that we can not express this new feature with the same logic as the current library, notably due to limitations of the variadic template features of C++.

Moreover, we want to add a cleaning step to reinitialize automatically the whole graph and its nodes' instances. This can be helpful to reuse a graph for multiple different data (for example: different pairs of matrices to multiply). Between each piece of data, the cleaning would reset the state of each node to prepare the computation for the next piece of data. The current design is to create a new graph for each input data. We need to study the consequences on the general graph workflow of this addition before implementing it.

In the same vein, we would like to lower the compilation duration, a way would be to use tools external to compilers such as Templight [7] for Clang to understand more precisely the cost of each compilation step, and ultimately to identify where to focus our improvement/optimization effort.

Second, there are current efforts surrounding Hedgehog. Experiments are con-

[7] https://github.com/mikael-s-persson/templight

ducted by researchers from the University of Utah to port Hedgehog to a cluster with the Uintah library. Cluster computing presents its own challenges, especially for inter-node communication, computation management, and memory distribution. The main idea is to use Uintah to take care of these aspects and use one Hedgehog graph on each node. We also assist the University of Maryland College Park to study the impact of a specialized scheduler on the library's performance. Their idea is to analyze a data-flow graph and to propose as a preprocessing step an improved version of the graph without cycles and with a specialized scheduler. They have created their own task (core and API) which embeds all the information needed to follow their supervised computation led by their own scheduler. They have previously made this kind of study on HTGS [Wu et al., 2021].

Third, the libraries FastLoader and HMBLib based on Hedgehog are currently under development to bring Hedgehog parallel capabilities to specific domains. We plan to add Python or Java library bindings to Hedgehog, and bridges to established libraries like OpenCV or TensorRT, or even using Hedgehog-based calls to the libraries. These additions allow users reluctant to use C++ to take advantage of the library's performance with other languages. The bridge from FastLoader to OpenCV or TensorRT could help researchers to easily exploit and manipulate large 2D or 3D microscopic images and develop deep learning inference workflows for such images.

Bibliography

[Aho et al., 1986] Aho, A. V., Sethi, R., and Ullman, J. D. (1986). *Compilers: Principles, Techniques, and Tools*. Pearson Education, Inc.

[Angerson et al., 1990] Angerson, E., Bai, Z., Dongarra, J., Greenbaum, A., McKenney, A., Du Croz, J., Hammarling, S., Demmel, J., Bischof, C., and Sorensen, D. (1990). Lapack: A portable linear algebra library for high-performance computers. In *Proceedings of the 1990 ACM/IEEE Conference on Supercomputing*, pages 2–11. IEEE.

[Augonnet et al., 2011] Augonnet, C., Thibault, S., Namyst, R., and Wacrenier, P.-A. (2011). Starpu: a unified platform for task scheduling on heterogeneous multicore architectures. *Concurrency and Computation: Practice and Experience*, 23(2):187–198.

[Austern, 2005] Austern, M. (2005). Draft technical report on c++ library extensions. Technical Report N1836=05-0096, International Standardization Working Group ISO/IEC JTC1/SC22/WG21.

[Bachelet and Yon, 2017] Bachelet, B. and Yon, L. (2017). Designing expression templates with concepts. *Software: Practice and Experience*, 47(11):1521–1537.

[Bardakoff et al., 2020a] Bardakoff, A., Bachelet, B., Blattner, T., Keyrouz, W., Kroiz, G. C., and Yon, L. (2020a). Hedgehog: Understandable scheduler-free heterogeneous asynchronous multithreaded data-flow graphs. In *2020 IEEE/ACM 3rd Annual Parallel Applications Workshop: Alternatives To MPI+X (PAW-ATM)*, pages 1–15. IEEE.

[Bardakoff et al., 2020b] Bardakoff, A., Blattner, T., Bachelet, B., Keyrouz, W., and Yon, L. (2020b). Hedgehog: a performance-oriented general purpose library for multi-gpu systems. Talk, GPU Technology Conference (GTC).

[Bauer et al., 2012] Bauer, M., Treichler, S., Slaughter, E., and Aiken, A. (2012). Legion: Expressing locality and independence with logical regions. In *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, pages 1–11. Supercomputing, IEEE.

[Bennett et al., 2015] Bennett, J., Clay, R., Baker, G., Gamell, M., Hollman, D., Knight, S., Kolla, H., Sjaardema, G., Slattengren, N., Teranishi, K., et al. (2015). Asc atdm level 2 milestone# 5325: Asynchronous many-task runtime system analysis and assessment for next generation platforms. Technical Report SAND2015-8312, US Department of Energy, Sandia National Laboratories.

[Binkley, 2007] Binkley, D. (2007). Source code analysis: A road map. In *Future of Software Engineering (FOSE 0́7)*, pages 104–119. IEEE.

[Blattner, 2016] Blattner, T. (2016). *The Hybrid Task Graph Scheduler*. PhD thesis, University of Maryland.

[Blattner et al., 2017] Blattner, T., Keyrouz, W., Bhattacharyya, S. S., Halem, M., and Brady, M. (2017). A hybrid task graph scheduler for high performance image processing workflows. *Journal of signal processing systems*, 89(3):457–467.

[Bracha, 2004] Bracha, G. (2004). Generics in the java programming language. Technical report, Sun Microsystems / Oracle.

[Catalán et al., 2019] Catalán, S., Martorell, X., Labarta, J., Usui, T., Toledo Díaz, L. A., and Valero-Lara, P. (2019). Accelerating conjugate gradient using ompss. In *20th International Conference on Parallel and Distributed Computing, Applications and Technologies*, pages 121–126.

[Culjak et al., 2012] Culjak, I., Abram, D., Pribanic, T., Dzapo, H., and Cifrek, M. (2012). A brief introduction to opencv. In *Proceedings of the 35th international convention MIPRO*, pages 1725–1730. IEEE.

[Czarnecki and Eisenecker, 2000] Czarnecki, K. and Eisenecker, U. (2000). *Generative Programming: Methods, Tools, and Applications*. Addison-Wesley.

[Dagum and Menon, 1998] Dagum, L. and Menon, R. (1998). Openmp: an industry standard api for shared-memory programming. *Computational Science & Engineering, IEEE*, 5(1):46–55.

[Damaševičius and Štuikys, 2008] Damaševičius, R. and Štuikys, V. (2008). Taxonomy of the fundamental concepts of metaprogramming. *Information Technology and Control*, 37(2):124–132.

[Darlington et al., 1993] Darlington, J., Field, A. J., Harrison, P. G., Kelly, P. H., Sharp, D. W., Wu, Q., and While, R. L. (1993). Parallel programming using skeleton functions. In *International Conference on Parallel Architectures and Languages Europe*, pages 146–160. Springer.

[Deane and Turner, 2017] Deane, B. and Turner, J. (2017). constexpr in practice. Technical Report P0810R0, International Standardization Working Group ISO/IEC JTC1/SC22/WG21.

[Dennis, 2011] Dennis, J. (2011). *Data Flow Graphs*, pages 512–518. Springer.

[Dos Reis et al., 2007] Dos Reis, G., Stroustrup, B., and Maurer, J. (2007). Generalized constant expressions. Technical Report N2235=07-0095, International Standardization Working Group ISO/IEC JTC1/SC22/WG21.

[Doxey, 2016] Doxey, C. (2016). *Concurrency*, page Chapter 10. O'Reilly Media.

[Edwards and Sunderland, 2012] Edwards, H. C. and Sunderland, D. (2012). Kokkos array performance-portable manycore programming model. In *Proceedings of the 2012 International Workshop on Programming Models and Applications for Multicores and Manycores*, pages 1–10.

[Edwards et al., 2014a] Edwards, H. C., Trott, C. R., and Sunderland, D. (2014a). Kokkos: Enabling manycore performance portability through polymorphic memory access patterns. *Journal of parallel and distributed computing*, 74(12):3202–3216.

[Edwards et al., 2014b] Edwards, H. C., Trott, C. R., and Sunderland, D. (2014b). Kokkos: Enabling manycore performance portability through polymorphic memory access patterns. *Journal of parallel and distributed computing*, 74(12):3202–3216.

[Ellson et al., 2001] Ellson, J., Gansner, E., Koutsofios, L., North, S., and Woodhull, G. (2001). Graphviz — open source graph drawing tools. In *Lecture Notes in Computer Science*, pages 483–484. Springer-Verlag.

[Falcou, 2019] Falcou, J. (2019). C++ informations broker. GitHub repository.

[Flynn, 1972] Flynn, M. J. (1972). Some computer organizations and their effectiveness. *IEEE Transactions on Computers*, C-21(9):948–960.

[Frigo and Johnson, 1997] Frigo, M. and Johnson, S. G. (1997). The fastest fourier transform in the west. Technical Report MIT-LCS-TR-728, Massachusetts Institute of Technology.

[Frigo and Johnson, 1998] Frigo, M. and Johnson, S. G. (1998). Fftw: An adaptive software architecture for the fft. In *Proceedings of the 1998 IEEE International Conference on Acoustics, Speech and Signal Processing*, volume 3, pages 1381–1384. IEEE.

[Gamma et al., 1994] Gamma, E., Helm, R., Johnson, R., Vlissides, J., and Patterns, D. (1994). *Elements of reusable object-oriented software*. Addison-Wesley.

[Garland et al., 2008] Garland, M., Le Grand, S., Nickolls, J., Anderson, J., Hardwick, J., Morton, S., Phillips, E., Zhang, Y., and Volkov, V. (2008). Parallel computing experiences with cuda. *IEEE Micro*, 28(4):13–27.

[Ghosh, 2004] Ghosh, D. (2004). Generics in java and c++ a comparative model. *ACM SIGPLAN Notices*, 39(5):40–47.

[Glück and Jørgensen, 1996] Glück, R. and Jørgensen, J. (1996). Fast binding-time analysis for multi-level specialization. In *International Andrei Ershov Memorial Conference on Perspectives of System Informatics*, pages 261–272. Springer.

[Gmys et al., 2020] Gmys, J., Carneiro, T., Melab, N., Talbi, E.-G., and Tuyttens, D. (2020). A comparative study of high-productivity high-performance programming languages for parallel metaheuristics. *Swarm and Evolutionary Computation*, 57:100720. 14 pages.

[Gregor et al., 2007] Gregor, D., Stroustrup, B., Siek, J., and Widman, J. (2007). Proposed Wording for Concepts (Revision 3). Technical Report N2421=07-0281, International Standardization Working Group ISO/IEC JTC1/SC22/WG21.

[Hoare, 1971] Hoare, C. A. (1971). Proof of a program: Find. *Communications of the ACM*, 14(1):39—-45.

[Hoberock, 2016] Hoberock, J. (2016). The Parallelism TS Should be Standardized. Technical Report P0024R2, International Standardization Working Group ISO/IEC JTC1/SC22/WG21.

[Hollman et al., 2016] Hollman, D. S., Bennett, J. C., Kolla, H., Lifflander, J., Slattengren, N., and Wilke, J. (2016). Metaprogramming-enabled parallel execution of apparently sequential c++ code. In *Second International Workshop on Extreme Scale Programming Models and Middleware*, pages 24–31. IEEE.

[Holmen et al., 2017] Holmen, J. K., Humphrey, A., Sunderland, D., and Berzins, M. (2017). Improving uintah's scalability through the use of portable kokkos-based data parallel tasks. In *Proceedings of the Practice and Experience in Advanced Research Computing 2017 on Sustainability, Success and Impact*, PEARC17, page 27. ACM. 8 pages.

[Järvi and Freeman, 2010] Järvi, J. and Freeman, J. (2010). C++ lambda expressions and closures. *Science of Computer Programming*, 75(9):762–772.

[Johnson, 1975] Johnson, D. B. (1975). Finding all the elementary circuits of a directed graph. *SIAM Journal on Computing*, 4(1):77–84.

[Jones, 1996] Jones, N. D. (1996). An introduction to partial evaluation. *ACM Computing Surveys*, 28(3):480–503.

[Järvi et al., 2006] Järvi, J., Stroustrup, B., and Dos Reis, G. (2006). Deducing the type of variable from its initializer expression. Technical Report N1984=06-0054, International Standardization Working Group ISO/IEC JTC1/SC22/WG21.

[Järvi et al., 2003] Järvi, J., Willcock, J., and Lumsdaine, A. (2003). Concept-controlled polymorphism. In *Proceedings of the International Conference on Generative Programming and Component Engineering (GPCE)*, volume 2830.

[Kaiser et al., 2020] Kaiser, H., Diehl, P., Lemoine, A. S., Lelbach, B. A., Amini, P., Berge, A., Biddiscombe, J., Brandt, S. R., Gupta, N., Heller, T., et al. (2020). Hpx-the c++ standard library for parallelism and concurrency. *Journal of Open Source Software*, 5(53):2352. 9 pages.

[Kale and Krishnan, 1993] Kale, L. V. and Krishnan, S. (1993). Charm++: A portable concurrent object oriented system based on c++. *SIGPLAN Notices*, 28(10):91–108.

[Klabnik and Nichols, 2019a] Klabnik, S. and Nichols, C. (2019a). *Fearless Concurrency*, pages 347–370. No Starch Press, San Francisco.

[Klabnik and Nichols, 2019b] Klabnik, S. and Nichols, C. (2019b). *Unsafe Rust*, pages 418–427. No Starch Press, San Francisco.

[Klimiankou, 2019] Klimiankou, Y. (2019). Constexpr: A great good but wrong idea. In *2019 Ivannikov Ispras Open Conference (ISPRAS)*, pages 3–8. IEEE.

[Kroiz et al., 2021] Kroiz, G. C., Bardakoff, A., Blattner, T., and Keyrouz, W. (2021). Study of exploiting coarse-grained parallelism in block-oriented numerical linear algebra routines. *Proceedings in Applied Mathematics and Mechanics*, 20(1):e202000089. 2 pages.

[Kukanov and Voss, 2007] Kukanov, A. and Voss, M. J. (2007). The foundations for scalable multicore software in intel threading building blocks. *Intel Technology Journal*, 11(4).

[Lilis and Savidis, 2019] Lilis, Y. and Savidis, A. (2019). A survey of metaprogramming languages. *ACM Computing Surveys (CSUR)*, 52(6):1–39.

[Luk et al., 2009] Luk, C.-K., Hong, S., and Kim, H. (2009). Qilin: exploiting parallelism on heterogeneous multiprocessors with adaptive mapping. In *42nd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 45–55. IEEE.

[Martinez and Cupitt, 2005] Martinez, K. and Cupitt, J. (2005). Vips-a highly tuned image processing software architecture. In *IEEE International Conference on Image Processing*, volume 2, pages 574–578. IEEE.

[Mattson et al., 2004] Mattson, T. G., Sanders, B., and Massingill, B. (2004). *Patterns for parallel programming*. Pearson Education.

[Meng and Berzins, 2012] Meng, Q. and Berzins, M. (2012). Uintah hybrid task-based parallelism algorithm. In *SC Companion: High Performance Computing, Networking Storage and Analysis*, pages 1431–1432. Supercomputing, IEEE.

[Musser and Stepanov, 1989] Musser, D. R. and Stepanov, A. A. (1989). Generic programming. In *Symbolic and Algebraic Computation*, pages 13–25. Springer.

[Myers, 1996] Myers, N. C. (1996). A New and Useful Template Technique: Traits. In *C++ Gems*, pages 451–457. SIGS Books.

[Niebler et al., 2011] Niebler, E., Gregor, D., and Widman, J. (2011). Us22/de9 revisited: Decltype and call expressions. Technical Report n3276, International Standardization Working Group ISO/IEC JTC1/SC22/WG21.

[Parnin et al., 2013] Parnin, C., Bird, C., and Murphy-Hill, E. (2013). Adoption and use of java generics. *Empirical Software Engineering*, 18(6):1047–1089.

[Pasalic, 2004] Pasalic, E. (2004). *The role of type equality in meta-programming*. PhD thesis, Oregon Health & Science University.

[Patel and Wen-mei, 2008] Patel, S. and Wen-mei, W. H. (2008). Accelerator architectures. *IEEE micro*, 28(4):4–12.

[Penuchot et al., 2018] Penuchot, J., Falcou, J., and Khabou, A. (2018). Modern generative programming for optimizing small matrix-vector multiplication. In *International Conference on High Performance Computing & Simulation (HPCS)*, pages 508–514. IEEE.

[Poldner and Kuchen, 2008] Poldner, M. and Kuchen, H. (2008). On implementing the farm skeleton. *Parallel Processing Letters*, 18(01):117–131.

[Qin et al., 2020] Qin, B., Chen, Y., Yu, Z., Song, L., and Zhang, Y. (2020). Understanding memory and thread safety practices and issues in real-world rust programs. In *Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI 2020, page 763–779. ACM.

[Ritchie et al., 1988] Ritchie, D. M., Kernighan, B. W., and Lesk, M. E. (1988). *The C programming language*. Prentice Hall Englewood Cliffs.

[Robert, 2011] Robert, Y. (2011). *Task Graph Scheduling*, pages 2013–2025. Springer.

[Robison et al., 2008] Robison, A., Voss, M., and Kukanov, A. (2008). Optimization via reflection on work stealing in tbb. In *IEEE International Symposium on Parallel and Distributed Processing*, pages 1–8. IEEE.

[Sheard, 2001] Sheard, T. (2001). Accomplishments and research challenges in meta-programming. In *International Workshop on Semantics, Applications, and Implementation of Program Generation*, pages 2–44. Springer.

[Siek and Lumsdaine, 2000] Siek, J. and Lumsdaine, A. (2000). Concept checking: Binding parametric polymorphism in c++. In *First Workshop on C++ Template Programming*.

[Silberschatz et al., 1991] Silberschatz, A., Peterson, J. L., and Galvin, P. B. (1991). *Operating system concepts*. Addison-Wesley.

[Sinnen, 2007] Sinnen, O. (2007). *Task scheduling for parallel systems*. Wiley.

[Stroustrup, 1997] Stroustrup, B. (1997). *The C++ Programming Language, Third Edition*. Addison-Wesley.

[Sutton, 2015] Sutton, A. (2015). Working Draft, C++ Extensions for Concepts. Technical Report N4361, International Standardization Working Group ISO/IEC JTC1/SC22/WG21.

[Sutton et al., 2013] Sutton, A., Stroustrup, B., and Dos Reis, G. (2013). Concepts Lite: Constraining Templates with Predicates. Technical Report N3580, International Standardization Working Group ISO/IEC JTC1/SC22/WG21.

[Taha, 1999] Taha, W. (1999). *Multi-stage programming: Its theory and applications*. PhD thesis, Oregon Graduate Institute School of Science & Engineering.

[Tarjan, 1972] Tarjan, R. (1972). Depth-first search and linear graph algorithms. *SIAM journal on computing*, 1(2):146–160.

[Teodoro et al., 2012] Teodoro, G., Hartley, T. D., Catalyurek, U. V., and Ferreira, R. (2012). Optimizing dataflow applications on heterogeneous environments. *Cluster Computing*, 15(2):125–144.

[Touraille, 2012] Touraille, L. (2012). *Application of model-driven engineering and metaprogramming to DEVS modeling & simulation*. PhD thesis, Université Blaise Pascal-Clermont-Ferrand II.

[Tu et al., 2019] Tu, T., Liu, X., Song, L., and Zhang, Y. (2019). Understanding real-world concurrency bugs in go. In *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '19, page 865–878. ACM.

[Unruh, 1994] Unruh, E. (1994). Prime number computation. Technical report, ANSI X3J16-94-0075/ISO WG21-462.

[Vandevoorde et al., 2017] Vandevoorde, D., Josuttis, N. M., and Gregor, D. (2017). *C++ Templates: The Complete Guide*. Addison-Wesley, 2nd edition.

[Veldhuizen, 2003] Veldhuizen, T. L. (2003). C++ templates are turing complete. Technical report, Computer Science Department, Indiana University.

[Veldhuizen and Gannon, 1998] Veldhuizen, T. L. and Gannon, D. (1998). Active libraries: Rethinking the roles of compilers and libraries. In *Proceedings of the SIAM Workshop on Object Oriented Methods for Inter-operable Scientific and Engineering Computing*, volume 99. SIAM.

[Voufo et al., 2011] Voufo, L., Zalewski, M., and Lumsdaine, A. (2011). Concept-clang: An implementation of c++ concepts in clang. In *Proceedings of the Seventh ACM SIGPLAN Workshop on Generic Programming*, pages 71—82. ACM.

[Wang et al., 2013] Wang, Q., Zhang, X., Zhang, Y., and Yi, Q. (2013). Augem: Automatically generate high performance dense linear algebra kernels on x86 cpus. In *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, pages 1–12. Supercomputing.

[Wu et al., 2021] Wu, J., Xie, J., Bardakoff, A., Blattner, T., Keyrouz, W., and Bhattacharyya, S. S. (2021). Cgmbe: a model-based tool for the design and implementation of real-time image processing applications on cpu–gpu platforms. *Journal of Real-Time Image Processing*, 18(3):561–583.

Appendices

Appendix A

Pi approximation

Source Code A.1: Monte Carlo simulation to estimate π with `std::async`

```
#include <iostream>
#include <vector>
#include <random>
#include <future>
#include <numbers>
#include <chrono>
#include <numeric>

class ExperimentPI{
  unsigned long long int const
      numberSamples_,
      samplesPerWorker_;

  int const numberWorkers_;

  std::vector<std::chrono::duration<double>> durations_;

 public:
  // Creates the experiment for a specific number of samples and
  ↪ number of workers
  ExperimentPI(unsigned long long int const numberSamples, int const
  ↪ numberOfWorkers)
      : numberSamples_(numberSamples),
      ↪ samplesPerWorker_(numberSamples / numberOfWorkers),
      ↪ numberWorkers_(numberOfWorkers) {
    durations_.reserve(numberOfWorkers);
  }

  // Runs the experiment
  double approximatePi(){
    durations_.clear();
    // Creates a vector of futures to gather the number of inside
    ↪ points
    std::vector<std::future<unsigned long long int>> futures;
    futures.reserve(numberWorkers_);
    durations_.reserve(numberWorkers_);

    // Runs the different workers asynchronously
    for(int t = 0; t < numberWorkers_; ++t){
      futures.emplace_back(std::async(std::launch::async,
      ↪ &ExperimentPI::piMonteCarlo, this));
    }

    unsigned long long int numberPointsInside = 0;
    // Gathers the results of each workers when they are done
    for(std::future<unsigned long long int>& f : futures){
      numberPointsInside += f.get();
    }

    // Computes the estimate value of pi
```

```
    double pi = 4.0 * (double)((long double) numberPointsInside / (long double)numberSamples_);
    return pi;
  }

  // Accessor to get the mean duration per workers
  std::chrono::duration<double> meanDurationPerWorker(){
    return std::accumulate(durations_.cbegin(), durations_.cend(), std::chrono::duration<double>::zero()) / durations_.size();
  }

  // Accessor to get the duration per workers standard deviation
  double stdvDurationPerWorker(){
    std::chrono::duration<double> mean = meanDurationPerWorker();
    std::vector<double> diff(durations_.size());
    std::transform(durations_.cbegin(), durations_.cend(), diff.begin(), [mean](std::chrono::duration<double>const & x) { return x.count() - mean.count(); });
    double sq_sum = std::inner_product(diff.begin(), diff.end(), diff.begin(), 0.);
    return std::sqrt(sq_sum / durations_.size());
  }

 private:
  // Monte Carlo simulation to estimate pi used by every workers
  [[nodiscard]] unsigned long long int piMonteCarlo(){
    std::random_device rd;
    std::mt19937 gen(rd());
    std::uniform_real_distribution<double> dis(0.0, 1.0);
    double x, y;
    unsigned long long int points_inside = 0;
    std::chrono::steady_clock::time_point start = std::chrono::steady_clock::now();
    for(unsigned long long int i = 0; i < samplesPerWorker_; ++i){
      // Generates the points
      x = dis(gen);
      y = dis(gen);
      // Counts the number of inside points
      if(x*x + y*y <= 1.0){ ++points_inside; }
    }
    std::chrono::steady_clock::time_point end = std::chrono::steady_clock::now();
    // Computes the duration of the algorithm
    auto duration = duration_cast<std::chrono::duration<double>>(end - start);
    // Stores the duration
    registerDuration(duration);
    return points_inside;
  }

  // Thread safe duration storage
  void registerDuration(std::chrono::duration<double> const & duration){
    // Creates a shared mutex to protect the duration storage
    static std::mutex m;
    m.lock(); // Locks the mutex, only one thread can lock it at a time
    durations_.push_back(duration);
    m.unlock(); // Unlocks the mutex, another thread can lock it again
  }
};

int main() {
  std::cout.precision(std::numeric_limits<double>::max_digits10);
  double const pi = std::numbers::pi_v<double>;

```

```
101    unsigned long long int numberSamples = 100000000;
102    int numberWorkers = 12;
103
104    // Creates the experiment
105    ExperimentPI experiment(numberSamples, numberWorkers);
106
107    std::chrono::steady_clock::time_point start =
       ↪  std::chrono::steady_clock::now();
108    double piApproximation = experiment.approximatePi(); // Runs the
       ↪  experiment
109    std::chrono::steady_clock::time_point end =
       ↪  std::chrono::steady_clock::now();
110    std::cout << "PI: " << pi << "\n";
111    std::cout << "Approximation PI: " << piApproximation << "\n";
112    std::cout << "Difference PI: " << abs(pi - piApproximation) <<
       ↪  "\n";
113    std::cout << "TimePerWorker: " <<
       ↪  experiment.meanDurationPerWorker().count() << " s +- " <<
       ↪  experiment.stdvDurationPerWorker() << "\n";
114    std::cout << "End to end computation: " <<
       ↪  duration_cast<std::chrono::duration<double>>(end -
       ↪  start).count() << "s\n";
115
116    return 0;
117  }
```

Appendix B

Hedgehog latency measurement

The following code has been used to measure the latency of Hedgehog library. Results are presented in Chapter 4.

Source Code B.1: C++ code to measure Hedgehog latency

```
#include <iostream>
#include <hedgehog/hedgehog.h>

struct A {};
struct B {};
struct C {};
struct D {};
struct E {};
struct F {};
struct G {};
struct H {};
struct I {};
struct J {};
struct R {};

class Task1 : public AbstractTask<R, A> {
  size_t c =0;
 public:
  Task1(size_t numberThreads) : AbstractTask("T1", numberThreads) {}
  void execute([[maybe_unused]]std::shared_ptr<A> ptr) override { }
  std::shared_ptr<AbstractTask<R, A>> copy() override { return
   ↪  std::make_shared<Task1>(this->numberThreads()); }
};

class Task5 : public AbstractTask<R, A, B, C, D, E> {
 public:
  Task5(size_t numberThreads) : AbstractTask("T5", numberThreads) {}
  void execute([[maybe_unused]]std::shared_ptr<A> ptr) override {}
  void execute([[maybe_unused]]std::shared_ptr<B> ptr) override {}
  void execute([[maybe_unused]]std::shared_ptr<C> ptr) override {}
  void execute([[maybe_unused]]std::shared_ptr<D> ptr) override {}
  void execute([[maybe_unused]]std::shared_ptr<E> ptr) override {}
  std::shared_ptr<AbstractTask<R, A, B, C, D, E>> copy() override {
    return std::make_shared<Task5>(this->numberThreads());
  }
};

class Task10 : public AbstractTask<R, A, B, C, D, E, F, G, H, I, J>
 ↪  {
 public:
  Task10(size_t numberThreads) : AbstractTask("T10", numberThreads)
   ↪  {}
  void execute([[maybe_unused]]std::shared_ptr<A> ptr) override {}
  void execute([[maybe_unused]]std::shared_ptr<B> ptr) override {}
  void execute([[maybe_unused]]std::shared_ptr<C> ptr) override {}
```

```
43    void execute([[maybe_unused]]std::shared_ptr<D> ptr) override {}
44    void execute([[maybe_unused]]std::shared_ptr<E> ptr) override {}
45    void execute([[maybe_unused]]std::shared_ptr<F> ptr) override {}
46    void execute([[maybe_unused]]std::shared_ptr<G> ptr) override {}
47    void execute([[maybe_unused]]std::shared_ptr<H> ptr) override {}
48    void execute([[maybe_unused]]std::shared_ptr<I> ptr) override {}
49    void execute([[maybe_unused]]std::shared_ptr<J> ptr) override {}
50    std::shared_ptr<AbstractTask<R, A, B, C, D, E, F, G, H, I, J>>
       ↪  copy() override {
51      return std::make_shared<Task10>(this->numberThreads());
52    }
53  };
54
55  template<class TASK, class ...T>
56  void runExperiment(size_t &numberElements, std::shared_ptr<TASK>
     ↪  &task) {
57    std::chrono::time_point<std::chrono::high_resolution_clock>
58        start,
59        finish;
60
61    auto g = Graph<R, T...>();
62    auto t = std::static_pointer_cast<AbstractTask<R, T...>>(task);
63
64    g.input(t);
65    g.output(t);
66    g.executeGraph();
67
68    start = std::chrono::high_resolution_clock::now();
69    for (size_t element = 0; element < numberElements; ++element) {
70      (g.pushData(std::make_shared<T>()), ...);
71    }
72    g.finishPushingData();
73    g.waitForTermination();
74    finish = std::chrono::high_resolution_clock::now();
75
76    auto duration =
       ↪  std::chrono::duration_cast<std::chrono::microseconds>(finish -
       ↪  start).count();
77    std::cout << (duration * 1.) / (numberElements * sizeof...(T)) <<
       ↪  ",";
78  }
79
80  int main() {
81
82    size_t numberElements = 1000000;
83
84    std::cout << "ClusterSize,1InputType,5InputTypes,10InputTypes" <<
       ↪  std::endl;
85
86    for (size_t clusterSize = 1; clusterSize <=
       ↪  std::thread::hardware_concurrency(); clusterSize *= 2) {
87      std::cout << clusterSize << ",";
88      { auto t1 = std::make_shared<Task1>(clusterSize);
89        runExperiment<Task1, A>(numberElements, t1);
90      }
91      { auto t5 = std::make_shared<Task5>(clusterSize);
92        runExperiment<Task5, A, B, C, D, E>(numberElements, t5);
93      }
94      { auto t10 = std::make_shared<Task10>(clusterSize);
95        runExperiment<Task10, A, B, C, D, E, F, G, H, I,
           ↪  J>(numberElements, t10);
96      }
97      std::cout << std::endl;
98    }
99  }
```

Appendix C

HTGS latency measurement

The following code has been used to measure the latency of HTGS library. Results are presented in Chapter 4.

Source Code C.1: C++ code to measure HTGS latency

```
#include <htgs/api/TaskGraphConf.hpp>
#include "../HTGS/src/htgs/api/ITask.hpp"

struct A : public htgs::IData {};
struct B : public htgs::IData {};
struct C : public htgs::IData {};
struct D : public htgs::IData {};
struct E : public htgs::IData {};
struct F : public htgs::IData {};
struct G : public htgs::IData {};
struct H : public htgs::IData {};
struct I : public htgs::IData {};
struct J : public htgs::IData {};
struct R : public htgs::IData {};

enum class Type {A, B, C, D, E, F, G, H, I, J};
struct Container : public htgs::IData {
  Container(A a) : a(a), type(Type::A) {}
  Container(B b) : b(b), type(Type::B) {}
  Container(C c) : c(c), type(Type::C) {}
  Container(D d) : d(d), type(Type::D) {}
  Container(E e) : e(e), type(Type::E) {}
  Container(F f) : f(f), type(Type::F) {}
  Container(G g) : g(g), type(Type::G) {}
  Container(H h) : h(h), type(Type::H) {}
  Container(I i) : i(i), type(Type::I) {}
  Container(J j) : j(j), type(Type::J) {}

  Type getType() const { return type; }
  const A &getA() const { return a; }
  const B &getB() const { return b; }
  const C &getC() const { return c; }
  const D &getD() const { return d; }
  const E &getE() const { return e; }
  const F &getF() const { return f; }
  const G &getG() const { return g; }
  const H &getH() const { return h; }
  const I &getI() const { return i; }
  const J &getJ() const { return j; }

 private:
  Type type;
  A a;
  B b;
```

```
45   C c;
46   D d;
47   E e;
48   F f;
49   G g;
50   H h;
51   I i;
52   J j;
53 };
54
55 class Task1 : public htgs::ITask<A, R> {
56   size_t c;
57  public:
58   Task1(size_t numberThreads) : ITask(numberThreads) {}
59
60   void executeTask(std::shared_ptr<A> data) override {}
61
62   std::string getName() override {
63     return "T1";
64   }
65
66   ITask <A, R> *copy() override {
67     return new Task1(this->getNumThreads());
68   }
69 };
70
71 class Task5 : public htgs::ITask<Container, R> {
72  public:
73   Task5(size_t numThreads) : ITask(numThreads) {}
74
75   void executeTask(std::shared_ptr<Container> data) override {
76     switch(data->getType()) {
77       case Type::A:break;
78       case Type::B:break;
79       case Type::C:break;
80       case Type::D:break;
81       case Type::E:break;
82       case Type::F:break;
83       case Type::G:break;
84       case Type::H:break;
85       case Type::I:break;
86       case Type::J:break;
87     }
88   }
89
90   std::string getName() override {
91     return "T5";
92   }
93
94   ITask <Container, R> *copy() override {
95     return new Task5(this->getNumThreads());
96   }
97 };
98
99 class Task10 : public htgs::ITask<Container, R> {
100  public:
101   Task10(size_t numThreads) : ITask(numThreads) {
102
103   }
104
105   void executeTask(std::shared_ptr<Container> data) override {
106     switch(data->getType()) {
107       case Type::A:break;
108       case Type::B:break;
109       case Type::C:break;
110       case Type::D:break;
111       case Type::E:break;
112       case Type::F:break;
113       case Type::G:break;
114       case Type::H:break;
115       case Type::I:break;
```

```
116          case Type::J:break;
117        }
118      }
119
120      std::string getName() override {
121        return "Task10";
122      }
123
124      ITask <Container, R> *copy() override {
125        return new Task10(this->getNumThreads());
126      }
127    };
128
129
130    void runExperimentOne(size_t &numberElements, Task1 *task) {
131      std::chrono::time_point<std::chrono::high_resolution_clock>
132          start,
133          finish;
134
135      auto graph = new htgs::TaskGraphConf<A, R>();
136      graph->setGraphConsumerTask(task);
137      graph->addGraphProducerTask(task);
138
139      auto executor = new htgs::TaskGraphRuntime(graph);
140      executor->executeRuntime();
141
142      start = std::chrono::high_resolution_clock::now();
143      for (size_t element = 0; element < numberElements; ++element) {
144        graph->produceData(new A());
145      }
146      graph->finishedProducingData();
147      executor->waitForRuntime();
148      finish = std::chrono::high_resolution_clock::now();
149      auto duration =
       ↪    std::chrono::duration_cast<std::chrono::microseconds>(finish -
       ↪    start).count();
150      std::cout << (duration * 1.) / (numberElements) << ",";
151
152      delete executor;
153    }
154
155    void runExperimentFive(size_t &numberElements, Task5 *task) {
156      std::chrono::time_point<std::chrono::high_resolution_clock>
157          start,
158          finish;
159
160      auto graph = new htgs::TaskGraphConf<Container, R>();
161      graph->setGraphConsumerTask(task);
162      graph->addGraphProducerTask(task);
163
164      auto executor = new htgs::TaskGraphRuntime(graph);
165      executor->executeRuntime();
166
167      start = std::chrono::high_resolution_clock::now();
168      for (size_t element = 0; element < numberElements; ++element) {
169        graph->produceData(new Container(A()));
170        graph->produceData(new Container(B()));
171        graph->produceData(new Container(C()));
172        graph->produceData(new Container(D()));
173        graph->produceData(new Container(E()));
174      }
175      graph->finishedProducingData();
176      executor->waitForRuntime();
177      finish = std::chrono::high_resolution_clock::now();
178      auto duration =
       ↪    std::chrono::duration_cast<std::chrono::microseconds>(finish -
       ↪    start).count();
179      std::cout << (duration * 1.) / (numberElements * 5) << ",";
180
181      delete executor;
```

```
182 }
183
184 void runExperimentTen(size_t &numberElements, Task10 *task) {
185   std::chrono::time_point<std::chrono::high_resolution_clock>
186       start,
187       finish;
188
189   auto graph = new htgs::TaskGraphConf<Container, R>();
190   graph->setGraphConsumerTask(task);
191   graph->addGraphProducerTask(task);
192
193   auto executor = new htgs::TaskGraphRuntime(graph);
194   executor->executeRuntime();
195
196   start = std::chrono::high_resolution_clock::now();
197   for (size_t element = 0; element < numberElements; ++element) {
198     graph->produceData(new Container(A()));
199     graph->produceData(new Container(B()));
200     graph->produceData(new Container(C()));
201     graph->produceData(new Container(D()));
202     graph->produceData(new Container(E()));
203     graph->produceData(new Container(F()));
204     graph->produceData(new Container(G()));
205     graph->produceData(new Container(H()));
206     graph->produceData(new Container(I()));
207     graph->produceData(new Container(J()));
208   }
209   graph->finishedProducingData();
210   executor->waitForRuntime();
211   finish = std::chrono::high_resolution_clock::now();
212   auto duration =
    ↪   std::chrono::duration_cast<std::chrono::microseconds>(finish -
    ↪   start).count();
213   std::cout << (duration * 1.) / (numberElements * 10) << ",";
214
215   delete executor;
216 }
217
218 int main() {
219   size_t numberElements = 1000000;
220
221   std::cout << "ClusterSize,1InputType,5InputTypes,10InputTypes" <<
    ↪   std::endl;
222
223   for (size_t clusterSize = 1; clusterSize <=
    ↪   std::thread::hardware_concurrency(); clusterSize *= 2) {
224     std::cout << clusterSize << ",";
225     {
226       auto t1 = new Task1(clusterSize);
227       runExperimentOne(numberElements, t1);
228     }
229     {
230       auto t5 = new Task5(clusterSize);
231       runExperimentFive(numberElements, t5);
232     }
233     {
234       auto t10 = new Task10(clusterSize);
235       runExperimentTen(numberElements, t10);
236     }
237     std::cout << std::endl;
238   }
239 }
```

Appendix D

Accuracy matrix multiplication

In order to validate our matrix multiplication computation with Hedgehog, we decided to estimate the accuracy of our results. The matrix multiplication computation is defined as, $C = A \cdot B + C$ with:

- A an $n \times m$ matrix,
- B an $m \times p$ matrix,
- C an $n \times p$ matrix.

The values of A, B, and C are real random values taken between $[0, 1)$. Because we are using limited size floating point numbers, the matrices are defined as:

$$\begin{aligned}
A &= \left[a_{ij} + \alpha_{ij}\right] \{i \in \mathbf{N} | 0 \leq i < n\} \{j \in \mathbf{N} | 0 \leq j < m\} \{\alpha \in \mathbf{R} | \alpha \ll 0\} \\
B &= \left[b_{jk} + \beta_{jk}\right] \{j \in \mathbf{N} | 0 \leq j < m\} \{k \in \mathbf{N} | 0 \leq k < p\} \{\beta \in \mathbf{R} | \beta \ll 0\} \\
C &= \left[c_{ik} + \gamma_{ik}\right] \{i \in \mathbf{N} | 0 \leq i < n\} \{k \in \mathbf{N} | 0 \leq k < p\} \{\gamma \in \mathbf{R} | \gamma \ll 0\}
\end{aligned}$$

α, β, γ model the errors due to the finite representation of real values. The representation is the same in a machine, so these errors are all upper bounded by a value ϵ, so we will use:

$$\forall i, \forall j, \forall k, |\alpha_{ij}| = |\beta_{jk}| = |\gamma_{ik}| = \epsilon.$$

So:

$$\begin{aligned}
\forall i, \forall k, (A \cdot B)_{ij} &= \sum_{j} (a_{ij} + \epsilon) \cdot (b_{jk} + \epsilon) \\
&= \sum_{j} a_{ij} \cdot b_{jk} + \epsilon \cdot (a_{ij} + b_{jk})
\end{aligned}$$

We do not consider ϵ^2, because ϵ is defined as the representation limit, and we know:

$$\begin{aligned}
0 < \epsilon << 1 \\
0 < \epsilon^2 << \epsilon
\end{aligned}$$

So ϵ^2 cannot be represented.

And:

$$\forall i, \forall k, (A \cdot B + C)_{ik} = c_{ik} + \epsilon + \sum_j a_{ij} \cdot b_{jk} + \epsilon \cdot (a_{ij} + b_{jk})$$
$$= \sum_j a_{ij} \cdot b_{jk} + c_{ik} + \sum_j \epsilon \cdot (a_{ij} + b_{jk}) + \epsilon$$

Let's define the $n \times p$ matrix R, which is the true result of the matrix multiplication $A \cdot B + C$ as:

$$\forall i, \forall k, R_{ik} = \sum_j a_{ij} \cdot b_{jk} + c_{ik}$$

So we have the following results with our finite representation of floating point numbers:

$$\forall i, \forall k, (A \cdot B + C)_{ik} = R_{ik} + \epsilon \cdot (1 + \sum_j (a_{ij} + b_{jk})) \tag{D.1}$$

To compute the accuracy of our algorithm, we first do the matrix multiplication computation with a given library (OpenBLAS), which gives a result T. Then, we do the computation with our library, that gives a result M. With a perfect representation of numbers $T - M = 0$. Because of D.1, T and M are affected by the same representation problem, so the same error applies:

$$\forall i, \forall k, T_{ik} = R^T_{ik} + \epsilon^T \cdot (1 + \sum_j (a^T_{ij} + b^T_{jk}))$$
$$\forall i, \forall k, M_{ik} = R^M_{ik} + \epsilon^M \cdot (1 + \sum_j (a^M_{ij} + b^M_{jk}))$$

The threshold used to test our accuracy is:

$$\forall i, \forall j, \forall k,$$
$$|T_{ij} - M_{ij}| = \left| R^T_{ik} + \epsilon^T \cdot (1 + \sum_j (a^T_{ij} + b^T_{jk})) - (R^M_{ik} + \epsilon^M \cdot (1 + \sum_j (a^M_{ij} + b^M_{jk}))) \right|$$

Or,

$$\forall i, \forall j, \forall k, a^M_{ij} = a^T_{ij}, b^M_{jk} = b^T_{jk}, c^M_{ik} = c^T_{ik}$$

So,

$$R^T = R^M$$
$$\sum_j (a^T_{ij} + b^T_{jk}) = \sum_j (a^M_{ij} + b^M_{jk}) = S_{ik}$$

We also consider that $\epsilon^T = -\epsilon^M$ to represent the worst-case scenario that no error compensates. We have:

$$|T_{ik} - M_{ik}| = |2 \cdot \epsilon \cdot (1 + S_{ik})|$$

Taking the absolute value will add an ϵ, and $|S_{ik}| = S_{ik}$ because the values of A, B, and C are real random values taken between $[0, 1)$.

Finally we have:

$$\forall i, \forall k, Error_{ik} = 2 \cdot \epsilon \cdot (1 + S_{ik}) + \epsilon \tag{D.2}$$

We can have a lower bound as:

$$\forall i, \forall k, ApproximateError_{ik} = 3 \cdot \epsilon \tag{D.3}$$

Appendix E

Execution of CPU-only matrix multiplication

The following figure shows the data-flow graph with performance measures of the CPU-only matrix multiplication Hedgehog implementation presented in Section 4.3. It exposes the results of an execution on a Mac Book Pro Mid 2015, for 10000×10000 float matrices divided into blocks of 2048×2048 elements, and with 3 threads for the addition and multiplication tasks.

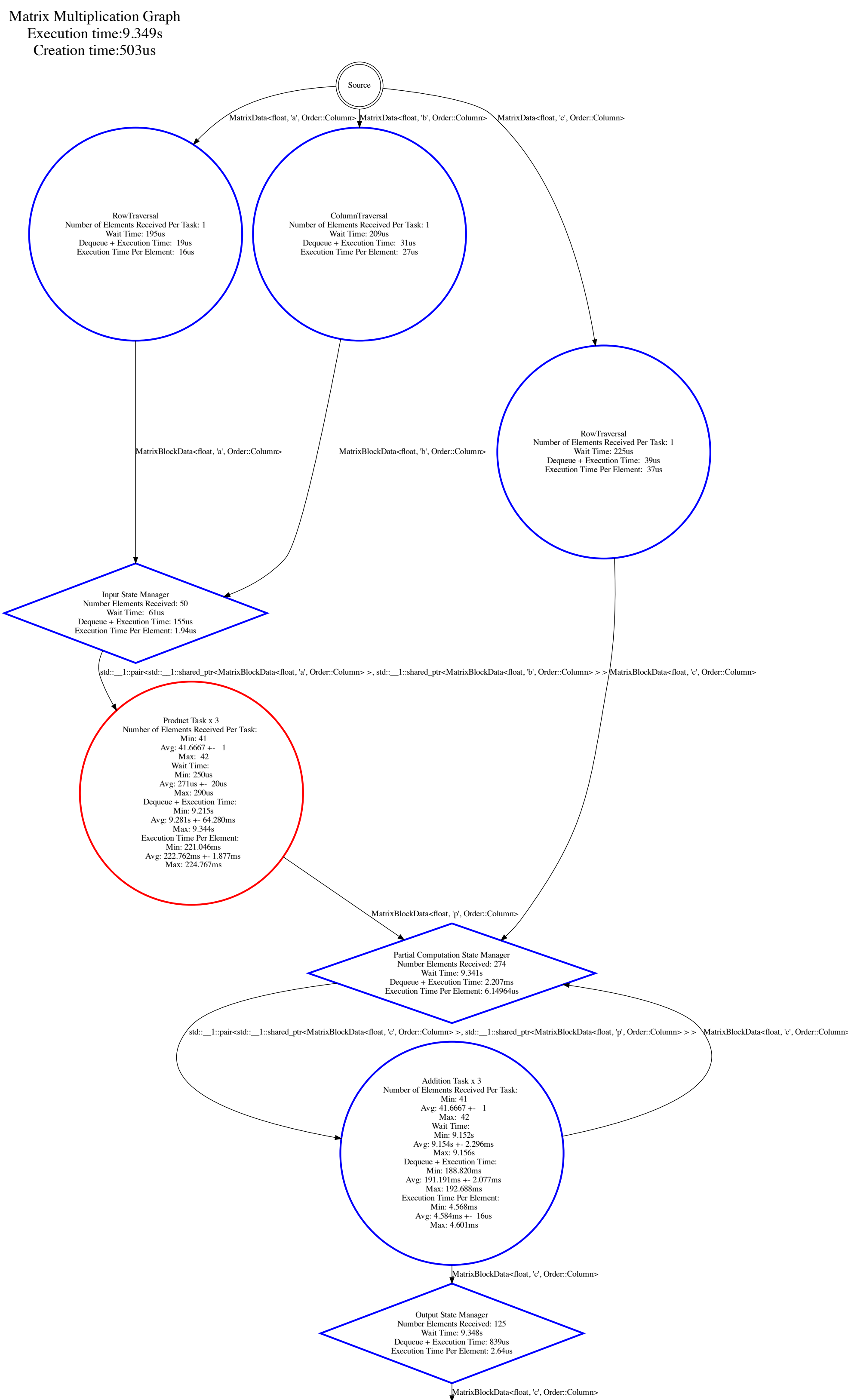


Figure E.1: Performance of the CPU-only matrix multiplication algorithm implemented with Hedgehog