THESIS

A COMPILER FOR HIERARCHICAL TASK-BASED PROGRAMMING ON DISTRIBUTED-MEMORY

Submitted by

Alexandre Dubois

Department of Computer Science

In partial fulfillment of the requirements

For the Degree of Master of Science

Colorado State University

Fort Collins, Colorado

Spring 2022

Master's Committee:

Advisor: Louis-Noël Pouchet

Sanjay Rajopadhye
James Wilson

ABSTRACT

A COMPILER FOR HIERARCHICAL TASK-BASED PROGRAMMING ON DISTRIBUTED-MEMORY

Today, computation intensive applications are run on heterogeneous clusters of machines and use the Message Passing Interface (MPI), which provides a library interface for message-passing between non-shared memory computational resources but comes at a high application development cost. Task-based programming, such as the Concurrent Collection (CnC) model, makes parallelism implicit by only describing task dependencies. This model has recently been extended to model programs with a hierarchical task-based representation, which allows to view tasks at different levels of decomposition, allowing to dispatch tasks of different level optimally to the different available architectures.

This thesis main work was to design an algorithm following a graph-based approach to transform a restricted class of single-level regular kernel to a two-level representation in the CnC model extended with hierarchical concepts. This transformation will alleviate performance boost by reducing communication cost and allowing the use of optimized tasks implementation at a coarse level.

After describing the CnC programming model concepts, the structure of the proposed graph based tiling algorithm will be developed. Then, the compiler implementing this algorithm on an Intermediate Representation representing a CnC program and generating a C++ version of the kernel using a new hierarchical CnC runtime.

Finally, the overhead of this runtime on a shared memory system for a 3D synthetic benchmark representing a classic linear-algebra dependency pattern. This characterization is done to help the user choose the target volume of tasks tile that has to be given as input of the tiling algorithm.

The main recommendation is to target a minimization of the number of super- tasks in the runtime while keeping the number of sub-tasks in every super-task under the order of 10000 sub-tasks.

ACKNOWLEDGEMENTS

I would first like to thank my advisor, Professor Louis-Noël Pouchet, for his support, professionalism and patience. Your dedication to your research and to push your students forward have really inspired me.

Thank you to Heather Sanders, for all her love and support. Thank you to my family and friends back in France for their moral support. I would also like to thanks the friends I made in the department, especially Steve and Matthew, as well as all my friends in Fort Collins for all the time we spent together.

DEDICATION

To my parents.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

Chapter 1

Introduction

Today, compute-intensive applications are run on heterogeneous clusters of machines [1], [2], [3]. The main framework used to distribute and parallelize such computations is Message Passing Interface (MPI) [4], [5], which provides a library interface for message-passing between non-shared memory computational resources. With this library, experienced developers can exploit the performance brought by such clusters, but the development cost of such parallel and distributed programs is huge.

Task-based programming allows the user to conveniently describe the dependencies of a parallel application. Parallelism between tasks is often implicit, like in CnC. This allows a huge reduction of the development cost of such applications. The supervision of the computation as well as the placement of each tasks on the available resources are then the responsibility of a task-based runtime, such as Intel Concurrent Collections [6], [7], [8] or the PARSEC runtime [9], [10].

When implementing a task-based version of a kernel, a programmer needs to be mindful of the computation size of the tasks [11]. Tasks that are too small will lead to a lot of time being lost in communications between each of those tasks, as well as an overhead of the runtime due to the management of a large number of tasks. On another hand, tasks that are too large means that the number of tasks will become small, and that due to this the runtime won't be able to use all the parallelism and concurrency that would be available with smaller tasks. With Concurrent Collections (CnC) [12], [13], the tuning of the task size is manually chosen by the user. The PIPES compiler [14] automatically tile tasks for the Intel Concurrent Collection runtime using polyhedral techniques.

While CnC has been shown to work on heterogeneous platforms [15], the heterogeneity of current supercomputers, potentially composed of CPUs, GPUs and FPGAs, brings the necessity for a hierarchical representation of tasks-based programming, to send tasks of the respective optimal

size for the different type of resources. This has created the need for runtimes that have built-in a hierarchical representation of the task-based application.

This work is part of a broader project which include the implementation of a new hierarchical Concurrent Collections runtime and the implementation of a compiler to automatically tile, tune and generate the code to give to the runtime from a simple representation of the program. This project is motivated by the optimization of compute intensive kernels used in the MADNESS software [16], [17]. The programs to optimize in MADNESS are irregular applications. Different operators are used on polynomial representation of multidimensional functions. Those function approximations are represented by trees with at their leaves the polynomials, with the intervals being recursively divided at with each level of the tree. The operators compute from node values, and the shape of the trees are not known at compile time since the actual function and how the k-d tree is laid out to represent it depends on the actual computation being made. It creates irregular computations, operating on tree nodes.

This thesis is preliminary work to the construction of a full compiler doing automatic tiling for irregular applications. It aims to enable the automatic aggregation of tasks and data into tiles from a single level Concurrent Collection representation of a kernel, into a 2-level hierarchical version to achieve massive performance boost versus a naive implementation. This thesis achieves this on a restricted set of regular kernels that will be described chapter in 4.

The contributions of this thesis are the following. We have prototyped an initial proof-of-concept, made of:

- An Intermediate Representation representing hierarchical programs up to 2-level,
- A code generator creating the corresponding C++ code using the new hierarchical runtime,
- A tiling algorithm, using a graph approach, that allows the user to tile a single-level program to a 2-level program, specifying the target tile size,
- A characterization of the hierarchical runtime overhead on a single node of a cluster to help the user to choose the optimum target tile size for the tiling algorithm.

First, the background of this thesis, composed of the descriptions of distributed runtimes anterior to Concurrent Collections or based on other concept, as well as different compilers doing automatic tiling of task based programs will be presented chapter 2.

Then, the Concurrent Collections programming model will be explained, as the output code of our compiler will be given to a runtime following that programming model. This runtime, implementing a hierarchical version of the Concurrent Collections programming model, as well as the Intel Concurrent Collections runtime will be described in chapter 3.

Chapter 4 defines the notion of tiling in the case of a Control Dataflow Graph (CDFG) and using the notions of the Concurrent Collection programming model. It also presents the tiling algorithm that will be implemented in our compiler, as well as the restrictions of the set of programs that this algorithm accept.

Chapter 5 precisely describes the implementation of the compiler that was built in the context of this thesis. After giving a broad overview of the compiler workflow, the intermediate representation (IR) of the compiler will be described, as well as the main transformations that are realized, such as the different part of the automatic tiling algorithm as well as the code generation.

Finally, the chapter 6 goal is to give the user of the compiler insight on the best way to choose the target tile size to give as input of the compiler tiling algorithm. To give those recommendations, a characterization of the runtime overhead is given in the context of a unique CPU, for a synthetic benchmark representing a classic linear-algebra kernel.

Chapter 2

Background

2.1 Data and computation distribution

Dataflow programming languages [18] need to distribute both data and computation between processors. When working on distributed memory systems, a dataflow runtime usually organizes the available processors so that a master processor gives orders to the others which will execute the computations and transfer data between themselves. Both Intel Concurrent Collections (CnC) and the new hierarchical CnC runtime, described chapter 3 use the Message Passing Interface (MPI) standard [4], [5] to manage all the communications between processors inside the runtime.

MPI is a library interface specification for message-passing between non-shared memory computational resources. It was created in 1993 and defines a communication protocol with the goal of targeting the Fortran and C languages. Today, a lot of different libraries implement this standard as OpenMPI [19], Intel MPI [20], or MPICH [21].

In a nutshell, MPI organizes each process available in communicators, that are groups of processes, defined by the user, in which each process has an ID called rank. When launching a program with MPI, every available process executes the code separately, by default without any synchronization nor data transmission between processes. The user can specify part of the code to be launched only for certain process ranks.

A point-to-point communication between two processes allows the user to transmit the data from a given process to another one (indicated by their ranks). To do this communication, the sending and receiving processes need to call their respective functions (MPI_Send and MPI_Recv) on the same message identifier. Those calls can be blocking or non-blocking. If the call is blocking, the process doing the send or the receive will wait for the communication to be done to continue its execution. In the non-blocking case, the process will continue without waiting. Blocking communications potentially restrain processes from doing anything while waiting for the other side of

the communication to do their job, while non-blocking communications alleviate this constraint, but asks the user to manually manage the synchronization.

A collective communication corresponds to a one to many or a many to one message. An example of one to many is the broadcast operation which distribute information from a specific process to a group of process. As an example of many-to-one is the gather operation which makes a specific process receive data from a group of processes. For those operations, the principle is the same as for the one-to-one communication, i.e. both the sending and the receiving processes need to call the function at their hand of the communication.

The Concurrent Collections runtime, described chapter 3 extensively use non-blocking point to point communications to bring the input data to a task starting its execution, with a synchronization point verifying that all the input has arrived.

MPI implementations come with a set of tools, like a wrapper around the Fortran, C and C++ compilers to include and link to the MPI implementation, as well as an MPI runner.

2.2 Dataflow languages and runtimes

In this thesis, the compiler that will be presented uses a hierarchical version of the Concurrent Collection (CnC) programming model, described chapter 3, which is a dataflow language [18].

CnC uniquely define data and tasks using an identifier (see section 3.1) named tag. In most kernels using the CnC programming model, those tags are represented using tuples that can represent multi-dimensional spaces for data and computations. This tuple indexes shared space at the runtime level and is inspired by the Linda distributed programming language [22]. In Linda, data and computations are also indexed in a global shared space called Tuple Space (TS). In this TS, data for a concrete tuple value are alive between a call to the *out()* function and a call to the *in()* function. This leads to a major differences with CnC, as this feature means that Linda does not follow the Dynamic Single Assignment (DSA) property, which leads to non-deterministic problems depending on the order of execution.

The DSA property of data shared between processes is a property that can be found in the X10 distributed programming language [23], [24]. X10 was created in 2004, is based on the Java language and separates the concerns of placement, computations, and data.

Data in X10 are of 2 different types: data local to a place and shared data between places. Local data can be modified, while shared multidimensional arrays must respect the Dynamic Single Assignment (DSA) property. Arrays indices are called points, and the shared multidimensional arrays are decomposed into regions on points, with each region being located at a specific place. The notion of point in X10 multidimensional-shared arrays is like the notion of tag in CnC for items.

A *place* corresponds to a computation resource, that will contain data and execute tasks. As in CnC, the number of places and its mapping to the architecture are known at runtime. The notion of placement in X10 is similar in the step placement tuner of CnC as described section 3.1.

An *activity* is a dynamic, asynchronous computation. An activity is set to be executed at a specific place and can spawn other activities to run at different places. Activity spawning can be set to wait or not to wait for the termination of the spawned activity. The non-waiting spawning of an activity is similar to the prescription of a step in CnC as described section 3.1.

Habanero is a programming language based on Java that extends X10 and use the Concurrent Collection programming model. The Habanero-Java runtime implements both work-sharing and work-stealing [25] strategies.

Two kinds of task are managed in Habanero. When an *async* task is spawned from a given task, this task continues its execution without waiting for the spawned task to finish. On the contrary, the *finish* construct indicates that the task spawning a new task needs to wait for it to finish before continuing. The prescription relation in the Concurrent Collection programming model corresponds to an asynchronous spawning of a task in Habanero-Java.

Habanero-Java implements different concurrent data-structure. An item collection in the Concurrent Collection programming model corresponds to a concurrent hash map in Habanero-Java.

Overall, Habanero-Java has a lot of structures and notions that have been simplified in CnC.

The Parallel Runtime Scheduling and Execution Controller (PaRSEC) [9], [10], [26], [27] is a distributed runtime, unrelated to Concurrent Collection. It is composed of:

- A compiler that transforms the input algorithm into a dataflow representation and then emit code for the runtime,
- A runtime that executes the program on a given architecture,
- Performance analysis tools.

The PARSEC compiler takes as input a serial affine C loop or a parametrized task graph (PTG) representation of a program [28] and output code for the runtime. A PTG is composed of a set of generic tasks, communications rules between those tasks and cost functions of each task. The PTG representation allows the PARSEC runtime to never have a full view of the full Control Dataflow Graph, which reduces its memory imprint and its overhead.

PARSEC also implements an alternative program representation, Dynamic Task Discovery (DTD) [26]. In DTD, the main notions are tasks, dependencies, and data. Data uses are indicated by the user (read, write or read-write). Dependencies between tasks are determined depending on the type of action done on global data (read, write). The runtime has at the same time different versions of the data being used, which seems to alleviate the Dynamic Single Assignment constraint.

PARSEC also implements a hierarchical DAG to manage execution on hybrid distributed systems [27]. The hierarchical DAG allows choosing different tile sizes depending on the type of resource (full CPU, CPU core, GPU) used to run the tile. The runtime considers two levels: a big tile size at the GPU level and a smaller tile size at the CPU level. The runtime manipulates the top-level tasks, and if a top-level task is given to a GPU, then it is executed. If no GPU are available, the runtime split the task into a sub-DAG composed of tasks with the smaller tile size, before being executed by available CPU resources.

LabView [29] is a graphical dataflow programing language in which the user graphically defines the tasks, called VI, and manually link the data dependencies between them. LabView brings

a set of control tools such as conditional, for and while loops. This set of controls allows the user to build irregular applications, with a control dataflow graph depending on the value of the data going through it. LabView also brings the notion of hierarchy through the concept of sub-VI. A sub-VI is literally a sub-graph of the main execution graph, reduced to a VI (or node) from the main execution graph. As a sub-VI is a sub-graph of the main execution graph, it respects by construction the convexity condition, described section 4.2.2, that will allow us to verify that the tiling of task done in this thesis is legal.

2.3 Compilers for task-based programming

The compiler presented in this thesis is mostly inspired by the PIPES compiler [14], [30]. PIPES provides a full compilation framework which, from a simple CnC graph specification, fully writes a C++ Intel CnC program. Our compiler uses a similar process for the new hierarchical CnC runtime, but lacks a user input language, and of the placement and tuner specification which PIPES has.

PIPES is built with an intermediate representation which extends the DFGR intermediate representation [31] with the tools needed to map the processes on a topology and specify the placement of each task. DFGR is a graph representation that simplifies the CnC model that will be described in Chapter 3. Control collections from the CnC model are removed to directly allow the dependency from a step to another. Specific sets of tags, such as ranges and polyhedra, are available to describe data dependency. DFGR is used to represent regular applications, and the graph of such applications can be completely unfolded without executing any of the steps. Similar unfolding techniques will be used in this thesis to build the execution graph before computing its transitive closure can be computed to verify the tiling legality, described section 4.2.2.

The two main transformations that the PIPES compiler makes on a CnC program are tasks coarsening and coalescing. Task coarsening correspond to the task tiling that will be implemented in this document. Task coalescing corresponds to the fusion of two tasks instances. To proceed with task coarsening and coalescing, the PIPES compiler uses the polyhedral model [32], [33], that

we will not describe in this thesis, and works on affine programs, which is a subset of the category of programs that CnC allows.

To apply those polyhedral transformations, PIPES starts by trying to translate the dataflow graph into concepts used by the polyhedral model, such as of the iteration domain the data read and written and the dependencies of each task. PIPES does not optimize any program for which a dependency cannot be written as an affine multidimensional function. Task coarsening and task coalescing are then applied as polyhedral tiling and task fusion on the polyhedral regions previously extracted. The polyhedral tiling is done using the PLUTO transformation algorithm [34], that will not be described in this thesis.

A similar approach of polyhedral optimizations applied on a dataflow language have been implemented in the POLYGLOT compiler [35], and is applied to the G language of LABVIEW, and specifically on its high-level intermediate representation, called DFIR [36].

The POLYGLOT passes are inserted to the LABVIEW compilation process before the passes done by LLVM. POLYGLOT first extracts polyhedral concepts from the dataflow language of LABVIEW, but contrary to CnC, the dynamic single assignment property is not ensured, and thus the first step that must be done is to replace all the in-place modification of data by a data copy. POLYGLOT considers every computation node of the dataflow graph as a statement in the polyhedral model in a similar way as PIPES. Both PIPES and POLYGLOT adapt the notion of Static Control Part, defined as a set of consecutive statements such that all loop bounds and conditionals are affine functions of the surrounding loop iterators and parameters [37], [38], to the specificity of the CnC and of the G programming language [30], [35]. Both compilers then use the PLUTO algorithm to apply polyhedral optimizations to the SCoP extracted.

Contrary to PIPES and POLYGLOT which use the polyhedral model to automatically tile tasks of dataflow graph, the approach in this thesis is graph based.

A similar approach is used in the TRACO compiler [39], [40], which has a tiler for arbitrarily nested loops [41]. The tiler of the TRACO compiler has the following process. First, a dependency analysis is done on the statements of the loop nest under study. This dependency analysis [42]

extracts each data dependency relations from each statement of the loop nest, dependency defined between an input and an output in a given domain.

The TRACO tiling algorithm starts from a rectangular target tile size, and build the set of the tile, containing the statement instances in the rectangular tile for each statement. For each of those tiles is built the set containing all the statement instances greater than the statement instances in the tile (following the lexicographic order). By computing the transitive closure of this set, all the statement instances that are dependence target accessible from this set is built. The intersection between this transitive closure and the set of instances in the tile contains all the statement instances that are at the origin of a cycle in the dependency graph. If this set is empty, the target rectangular tile is valid and can be used as is. If there are instances in the intersection, the target tile needs to be shaped into a valid tile. First, all instances origin of a cycle is removed from the tile. Then, the set of instances which are in a tile inferior to the current tile, which are target of dependencies whose source are in the tile and which are not target of dependencies from the set of instances superior to the tile are added.

In this thesis, the intermediate representation presented will be mainly inspired by the PIPES IR, and the tiling algorithm will use the graph transitive closure the same way as the TRACO compiler to check the validity of the tiling found.

Chapter 3

The Concurrent Collection Universe

In this chapter, we will describe all the notions of Concurrent Collection needed to describe the content of this work. The programming model, execution model and some of the existing runtimes will be described.

3.1 Basic notions

Concurrent Collection (CnC) [12], [13] is a programming model, created in 2009, aiming to facilitate the distribution and parallelization of mathematical kernels. CnC introduces a separation of concepts which allows a domain expert to implement the code of each tasks to be executed without having to take care about the parallelism/distribution meanwhile a CnC expert may implement the parts tuning the kernel to the architecture. The CnC programming model uses a set of notions that will be defined below.

Definition 1. *An **item** is a quantum of data from the point of view of the CnC programming model. An item must be serializable to be used by CnC.*

Definition 2. *A **step** is a quantum of computation from the point of view of the CnC programming model. The output of a step must be a pure function of the step inputs.*

Definition 3. *A **tag** is a unique identifier for the different elements of the programming model. A tag needs to be hashable to be used by CnC.*

Definition 4. *A **tag collection** is a type of tag, statically defined by the user. A **tag instance** is a tag that is part of a tag collection. It corresponds to a unique identifier of the type of its tag collection.*

Definition 5. *An **item collection** defines a type of item, and is characterized by the type of data of the items in the collection and by a tag collection. Item collections are statically defined by the user. An **item instance** is uniquely defined by its item collection and by a unique tag instance from*

the tag collection associated with its item collection. Inside an item collection, all items have a unique tag and are of the same type of data. Items respect the Dynamic Single assignment (DSA) property, i.e., an item instance can be inserted in the collection only once. An item instance can be accessed an arbitrary number of times. Item instances have a Boolean value indicating if it has been put.

Definition 6. *A **step collection** defines to a type of step, characterized by its name, its execution code and by the tag collection that controls it. Step collections are statically defined by the user. A **step instance** is uniquely defined by its step collection and by a unique tag instance from the tag collection associated with its step collection. A step instance can be executed once it is **control ready** and **data ready**. A step instance is control ready once it is inserted in the step collection. A step instance is data ready once all its input item instances are inserted in their respective item collections. Step instances have a Boolean value indicating if it has been prescribed. Another Boolean indicates if all the items that the step instance get are available. A last Boolean indicates if the step is done executing.*

All the defined term will allow user to define a Concurrent Collection specification, that will serve as input of the CnC runtimes.

3.1.1 The CnC specification

Definition 7. *The **CnC specification** is the description of the kernel the user needs to gives to the CnC runtime. It is composed of the execution code of the steps, the specification graph and the context of execution.*

In the specification graph, for each step collection, the interaction between a variable step instance and the other elements of the graph. Those interaction are of three types: get, put, and prescribe, with for origin the variable step instance being defined. The inputs, computations and outputs of the kernel are finally defined with the same interaction, with for origin the environment.

Definition 8. *The **environment** represents the outside of the computation kernel. It is from the environment that CnC puts the kernel input, prescribes the kernel computations, and gets the kernel output.*

Definition 9. *A **get** has for origin a step or the environment and for destination an item. If the origin is a step node, then the destination item is an input of the step. If the origin is the environment, then the destination item is an output of the kernel.*

Definition 10. *A **put** edge has for origin a step or the environment and for destination an item. If the origin is a step, then the destination item is an output of the step. If the origin is the environment, then the destination item is an input of the kernel.*

Definition 11. *A **prescribe** edge has for origin a step or the environment and for destination a step. If the origin is a step, then the destination step is dynamically asked to be computed once the origin step is done executing. If the origin is the environment is asked to be computed at the beginning of the execution of the kernel.*

Get, put and prescriptions can be executed conditionally.

Algorithm 1: Example of a CnC specification graph

1 int M;
2 int N;
3 $env \rightarrow [A : 0, 0..N]$;
4 $env \rightarrow (S : 0, 0..N)$;
5 $env \leftarrow [A : M, 0..N]$;
6 $(S : i, j) \leftarrow [A : i, j]$;
7 $(S : i, j) \rightarrow [A : i + 1, j]$;
8 $(S : i, j) \rightarrow (S : i + 1, j)$ if $i + 1 < M$;

An example of a CnC specification is given 8. Syntactically, item instances are represented between brackets and step instances are represented between parentheses. The kernel described has two constants M and N (line 1 and 2), one step collection S and one item collection A. Both the item collection A and the step collection S are prescribed by a 2-dimensional tag collection.

The environment puts the first line of length N of item instances of A, prescribes the first line of length N of step instances of S, and is waiting to get the line M of length N of item instances of A. Then, a variable step instance S of tag i, j is defined to get the item instance A of tag i, j, to put the item instance A of tag $i + 1, j$ and to prescribe the step instance S of tag $i + 1, j$ if $i + 1 < M$.

Now that the CnC specification has been described, the execution model of CnC will be discussed.

3.2 The Concurrent Collection execution model

The CnC execution model proceeds as follow.

First, the environment puts the program inputs items, as well as the steps to be prescribed in the runtime.

The program terminates once all the steps prescribed are done executing and once the items that the environment needs to get are available.

A step instance that has been prescribed is control ready. Based on the CnC specification graph, the concrete tag of the step instance allows the runtime to determine the concrete set of item instances that the step has a consumer relation with. Once all the item instances of this set are available, the step instance become data ready. When the step instance is both control and data ready, the step can now be executed. At the end of its execution, the set of item instances produced become available, and the set of step instances controlled become available.

The computation that CnC does follows a Control Data Flow Graph (CCDFG) that evolves dynamically as the computation goes on. In this CCDFG, steps and items instances as well as the environment are represented as nodes, and the get, put and prescribe relations are represented as edges.

An example of a CCDFG graph, with no tags to simplify the content, is given 3.1. The execution goes as follows. First, the environment puts the items a and b and prescribes the step s_1. The step s_1 is then control ready, and data ready as both the items a and b are available, and then executes. At the end of s_1 execution, it puts the item c and prescribes the step s_2, making it control

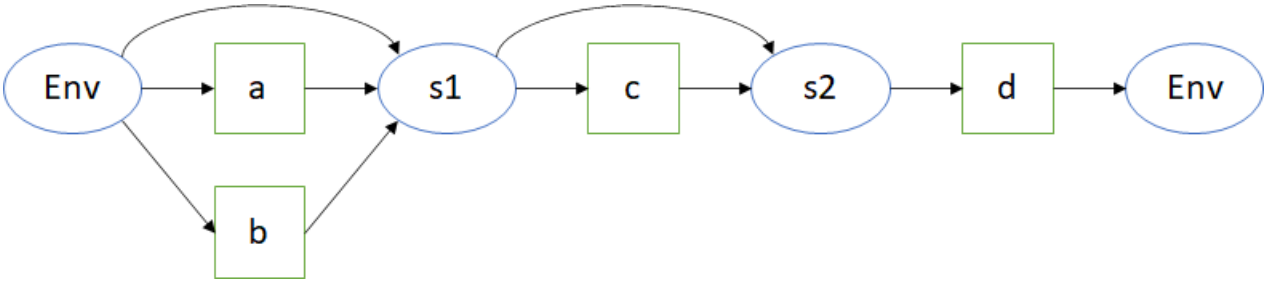

Figure 3.1: Example of CCDFG

ready. As the item c is available, s_2 can execute, and, at the end of execution, s_2 puts the item d. At this point, all the items that the environment needs to get (the item c) are available and the execution terminates normally.

CnC allows one to keep the parallelism implicit, and the user doesn't have to care about it. Any step that is both control ready and data ready can be executed on the next available resource.

This execution model is implemented in different runtimes, and two of those will be described in the next section.

3.3 The Concurrent Collection runtimes

In this section, we will describe two different CnC model: Intel Concurrent Collections and Hierarchical Concurrent Collections.

3.3.1 Intel Concurrent Collection

Intel CnC [7] is a C++ runtime implementing the Concurrent Collection programming model. This runtime can execute a CnC graph in both shared and distributed memory environments.

To describe the CnC kernel, the user starts to define the tag, item, and step types. For each tag type, the user needs to make the type hashable. For each item type, the user needs to make this type hashable and give an equality function. For distributed memory system, item types need to an implementation of the serialization and deserialization. For each step type, the execution function, taking as argument the step tag and the execution context, needs to be implemented. This function

first includes all the gets that the step does, then the implementation of the step computation and finally the puts and prescriptions the step does.

The context class then brings the whole CnC kernel together. It contains as attributes at least each tag collections, item collections and step collections of the program, as well as some potential tuners. The main function of the program finally instantiates the context, put the items input and the steps prescribed by the environment in the context, call a function that wait for the program to terminate and get all the items produced by the program (defined as gets of the environment in the specification).

On top of describing the CnC graph, the user can tune the kernel specifically for the architecture, to improve the performance of the kernel. The tuners in Intel CnC [43] allows the user to add the following:

- Predeclare the data dependencies
- Declare affinity to threads
- Specify tasks distribution to pin each step instance to specific processes
- Declare where each item instance will be produced and used

3.3.2 Hierarchical Concurrent Collection

Hierarchical Concurrent Collection is a new C++ runtime that implements the possibility of giving to the steps and items collections multiple hierarchical levels. In this runtime, the relationship between a super-level and a sub-level is that an element (step/item) of the super-level is associated to a tag, called super-tag, that is associated to a set of tags of the sub-level, called sub-tag.

To add a super-level to a single level kernel implementation, the user needs to add specific information describing the tiling of steps and the tiling of items.

Tiling of items

To tile an item-collection, the user needs to choose or implement the data structure to store a tile of items, and to add a specific tuner that give the following information:

- The super-tag corresponding to a sub-tag given as input.
- The sub-item from the super-item and a sub-tag given as input.
- How to store the sub-item of tag sub-tag in a super-item.

Tiling of steps

To tile a step-collection, the user needs to add a specific tuner that give the following information:

- The super-tag corresponding to a sub-tag given as input.
- Indicate when a tile is full when adding to it the different sub-tags (usually a counter).
- Indicate the set of sub-tags that are prescribed from outside the tile (from the environment or from outside steps.

If a step tile both read and write in a same item tile, then the user needs to pre-declare the item super-tag in another tuner to give that information to the runtime.

The runtime allows the two kinds of execution of a tiled step. The user can specify a specific code to run for the whole step tile. In this case, the runtime considers a tile of steps as a step. The user can also ask for the step to be run in a micro-runtime. In this case, the runtime sends the step tile to be run on a specific resource and executes it as its own specific CnC graph on this shared system.

Chapter 4

Tiling of Control Data Flow Graphs

In this chapter, the tiling of both tasks and data in a control data flow graph (CDFG) will be discussed. If in this thesis the same word of tiling is used for tasks and data as the notions are similar, their meaning and interest are very different.

Before describing tasks and data tiling in the case of the CDFG of CnC applications, the generic definitions given to the CnC notions will be specialized to the specific class of programs of interest. Then, the tiling of tasks will be discussed, and its validity conditions will be explicated. Finally, the tiling of data and its application in our context will be explained.

4.1 Graph representation

To reason on items and steps, the previous definitions of a tag collection and of a tag instance are restricted to the following:

Definition 12. *A tag instance of a given tag collection is defined as a tuple of integers. A tag collection is then characterized by the number of dimensions of the multidimensional space in which its tag instances will evolve.*

This definition allows to consider step collections to have their own iteration space and to consider item collections as multidimensional arrays. Get and put of items can then be seen as read and write at specific indices of those multidimensional arrays.

Before reasoning on tiling, the following restrictions will be put on the iteration's spaces and on the read/write relations between steps and items.

Restriction 1. *The iteration space of step collections is dense. The same restriction applies to item collections.*

Restriction 2. *Gets, puts and prescribes can be executed conditionally, but that condition is only allowed to depend on the value of the tag of the step.*

The density property allows us to have a very simple definition of neighbor for step instances.

Definition 13. *Two step instances are neighbors if they are from the same step collection, their tag is distinct, and in each dimension of the iteration space their distance is inferior or equal to 1.*

The set of neighbors of an arbitrary step instance in the step iteration space (not at its border) is then a known set.

The restriction on the conditionals excludes irregular applications from the scope of the tiling algorithm. This allows us to know exactly the set of items read and written for a given step instance.

4.2 Task tiling

4.2.1 Definitions

In compilation, tiling [44], [45], [46] is a loop transformation [47] which aggregates a group of loop iterations together, a group that can be multi-dimensional in case of a loop nest.

Definition 14. *In the case of a CDFG, a tile is an aggregation of tasks, which are the nodes of the graph, that is replaced as a single node (or a single macro-task) in the graph.*

With this definition, the directed edges going to and starting from the tiled node is the set of all edges that goes from/to a node outside the tile to/from a node inside the tile.

The full execution CDFG graph that terminates correctly using the CnC programming model cannot have any dependency cycle. If there was a dependency cycle in such a graph, then the first prescribed step of this cycle would never be able to execute as it would be waiting for the execution of the step before this cycle. This violate the condition that all the prescribed steps executed correctly, which is one of the conditions for a correct termination of CnC.

The figure 4.1 shows an example of a valid and an invalid tile on a simplified representation of a CDFG, in which only the tasks are kept as nodes and dependencies as directed edges (independently of if the dependency is a data dependency or a control dependency). In both examples, the nodes identified 1, 2 and 4 are replaced by a node identified A. All the directed edges crossing the

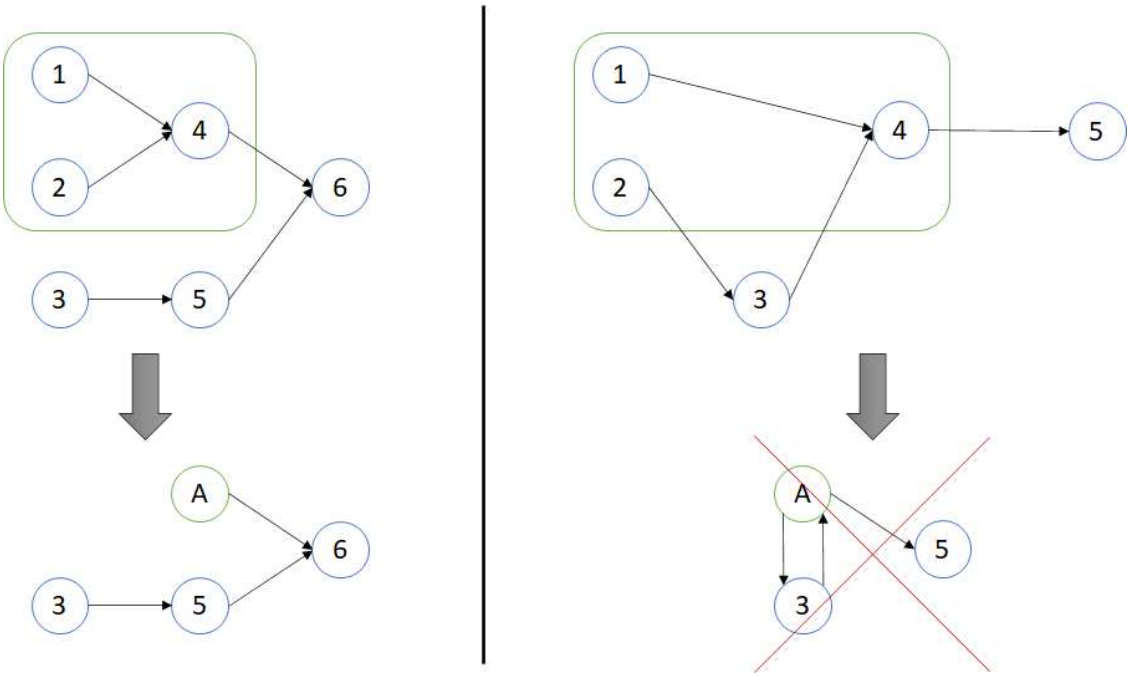

Figure 4.1: Example of a valid (on the left) an invalid (on the right) tile in a simplified CDFG

border of the tile are kept in the new graph. The example on the left is a valid tile as the resulting CDFG would terminate correctly using the CnC execution model. The example on the right is an invalid tile as the resulting CDFG has a cycle.

This is the only condition that tiles need to respect. By construction, the DSA property of the CnC guarantees that the execution is deterministic, without any data race [12].

Now that a general definition of a task tile has been given, its specification to the CnC programming model will be developed.

Definition 15. *A step tile is defined as a set of step instances that are grouped together and replaced in the CCDFG graph by a single atomic step. The edges interacting with this new node is the set of the edges of all the steps in the groups.*

In this work, we will restrict the tile notion to only contain steps from the same step collection. A step tile is characterized by:

- a step collection
- the tile tag, called **super-tag**

- the set of steps in the tile. Each of those steps tag is called **sub-tag**.
- the set of items that is got by the tile
- the set of items that is put by the tile
- the set of steps that is prescribed by the tile

A set of restrictions is applied on steps and on step tiles.

Restriction 3. *The iteration space of a step-collection is dense and includes the origin point of the space.*

Restriction 4. *A step tile forms a dense domain in the iteration domain of the step collection.*

Restriction 5. *Both the iteration space of a step collection and the step tile are of rectangular shape.*

Those restrictions allow, for a given step collection, to fully define the tiling of its iteration domain based on:

- The tile size in each dimension of the step collection iteration space $size$.
- The tiling direction in each dimension of the step collection iteration space $direction$.

Based on that information, for a given sub-tag of dimension n named $subtag_i$, a tile size $size_i$ and for a positive direction in each dimension $0 \leq i < n$, the super-tag $supertag_i$ is given by:

$$supertag_i = subtag_i \ \mathbf{div} \ size_i \tag{4.1}$$

In reverse, from a given super-tag $supertag_i$, all the sub-tags in the tile are expressed as:

$$size_i \times supertag_i \leq subtag_i < size_i \times (supertag_i + 1) \tag{4.2}$$

To adapt the tiles function to the iteration space of the step collection, the only function that needs to be changed is the one that output the sub-tags of a tile from its super-tag. If the iteration

domain is defined as the rectangle between the min value m_i and the max value M_i, then the previous equation becomes:

$$max(size_i \times supertag_i, m_i) \leq subtag_i \tag{4.3}$$

$$subtag_i < min(size_i \times (supertag_i + 1), M_i) \tag{4.4}$$

From those definitions, the set of items both read and written by the tile and the set of steps instances prescribed from outside the tile can be inferred. The set of items both read and written by the tile is the intersection of the set of items that the tile gets and of the set of items that the tile put. The set of step instances in the tile that are prescribed from outside the tile is the set of step instances in the tile that are not in the set of step instances prescribed by the tile.

Example: 1D accumulation in a 2D space

Algorithm 2: Accumulation 1D in a 2D iteration space

```
1 int M;
2 int N;
3 env → [A : 0, 0..N];
4 env → (S : 0, 0..N);
5 env ← [A : M, 0..N];
6 (S : i, j) ← [A : i, j];
7 (S : i, j) → [A : i + 1, j];
8 (S : i, j) → (S : i + 1, j) if i + 1 < M;
```

Let's consider the kernel of a 1D accumulation in a 2D space, as described in the algorithm 8, that will be tiled with a tiling of dimension $(4, 3)$ for the step S, as shown in the figure 4.2. Each point represents a step instance having the tag corresponding to its coordinates. The red arrows represent step dependencies (in this case both control and data dependency). The set of sub-tags in the tile, represented by the green rectangle for a non-border case in the figure 4.2, is:

$$subtags = \{(x, y) : max(i, 0) \leq x < min(i + 4, M), max(j, 0) \leq y < min(j + 3, N)\} \tag{4.5}$$

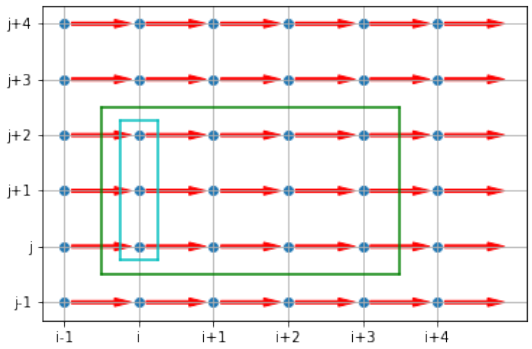

Figure 4.2: General tile shape of the step S of the accumulation 1D in a 2D iteration space

Considering we tile starting from the origin, the super-tag of this tile is:

$$supertag = (i \textbf{ div } 4, j \textbf{ div } 3) \tag{4.6}$$

The sets of items of type A get and put by the tile and the set of steps S prescribed by the tile are respectively:

$$get = \{(i,j) : max(i,0) \leq x < min(i+4, M), max(j,0) \leq y < min(j+3, N)\} \tag{4.7}$$

$$put = \{(i,j) : max(i,0) + 1 \leq x < min(i+4, M) + 1, max(j,0) \leq y < min(j+3, N)\} \tag{4.8}$$

$$presc = \{(i,j) : max(i,0) + 1 \leq x < min(i+4, M) + 1, max(j,0) \leq y < min(j+3, N)\} \tag{4.9}$$

Finally, the set of sub-tags in the tile that are prescribed from outside the tile, represented in the cyan rectangle in the figure 4.2, is:

$$border = \{(max(i,0), y) : max(j,0) \leq y < min(j+3, N) \tag{4.10}$$

Through those definitions and the previous example, the tiling of tasks has been explicated, and its legality can now be discussed.

4.2.2 Legality

This section will reason on a regular CnC graph, i.e., any conditional edge can only be depending on the tag concrete value of the corresponding step and on the program constants.

Definition 16. *A step tile is convex if for any two steps in the tile, all steps in all paths from/to the first step to/from the second step are also in the tile.*

In the figure 4.1, an example of valid and invalid tiles are given. Knowing that the DSA property of the CnC programming model and that the steps are pure functions, the program execution will be deterministic.

Theorem 1. *A tiling of a valid single level CnC graph verifying the previous restrictions is valid iff there is no cycle in the super-level graph.*

A valid tiling of a valid single level CnC graph executes correctly following the CnC execution model. One of the correct termination conditions is that every prescribed super-step finished executing. If there was a cycle in the super-level graph, it means that the first step of this cycle to be prescribed cannot be executed as one of its dependencies is not ready, which mean that the graph cannot terminate correctly. So, if there is a valid tiling of a valid single level CnC graph, then there is no cycle in its super-level graph.

Now, let's consider a valid single level CnC graph for which there is a tiling that has no cycle in the super-level graph. As there are no cycle at the super-level, it means that there is no cycle that involve both the super-level graph and the sub-level graph from a tile as it would mean that there would be a cycle in the super-level graph. As the single level CnC graph is valid, there cannot be a cycle purely inside a sub-level graph of the tiling. Finally, as every tile is a correct CnC graph, it can be considered as a pure function, and as the super-level graph has no cycle, the program can terminate correctly, and the tiling is valid.

4.2.3 Verification

Definition 17. *The adjacency matrix of a directed graph is a Boolean square matrix representation of such graph, with as size the number of nodes in the graph. Then, each directed edge between*

the nodes numbered m and n is represented by a true value in the adjacency matrix at the line m, column n.

Definition 18. *The transitive closure matrix of a directed graph is a Boolean square matrix representation of such graph, with as size the number of nodes in the graph. There is a true value in the matrix at the line m, column n if there is a path between the node numbered m and the node numbered n.*

Theorem 2. *A directed graph has no cycle iff there are no true value on its transitive closure matrix diagonal.*

From the adjacency matrix representation of the CCDFG graph, the transitive closure of the graph is computed following the algorithm 3. This is the algorithm that have been implemented in the compiler, which is of complexity of $O(V^3)$, where V is the number of vertices.

In [48], it is shown that computing the transitive closure of a directed graph is a specific case of the all pairs shortest path problem. So there are ways to change the current algorithm to reduce the complexity. Johnson's algorithm is mentioned in [48] as being $O(V^2log(V)+VE)$ with E the number of edges. In [49], in chapter 7, Tarjan mentions an algorithm based on a call to Dijkstra's algorithm for every sources of the graph. In our case, a CnC graph is single source, which is the environment node. Dijkstra's algorithm is $O(Elog_{2+m/n}(V))$ [49].

4.3 Data tiling

What will be called in this section data tiling is fundamentally different from the task tiling defined in the previous section. A single level item collection can be viewed as a single level mapping between tags and items. The data tiling in our context is the aggregation of items to transform the single level mapping into a two level mapping with the first level mapping a super-tag with a super-item, which is itself a mapping associating sub-tags to sub-items.

The steps can then access and write items at the super and the sub levels. The definition of an item tile, or super-item, is given below.

Algorithm 3: Floyd Warshall transitive closure algorithm

Input:
int[][] adjmat: Adjacency matrix of the graph.
int N: Number of nodes in the graph.
Output:
int[][] out: Transitive closure of the graph.

```
1 begin
2     for i ← 0 to N do
3         for j ← 0 to N do
4             out[i][j] ← adjmat[i][j]
5     for k ← 0 to N do
6         for i ← 0 to N do
7             for j ← 0 to N do
8                 out[i][j] ← out[i][j] or (out[i][k] and out[k][j]);
9     return out;
```

Definition 19. *An item tile is defined as a set of item instances that are grouped together and replaced in the CCDFG graph by a single atomic step. The edges interacting this new node is the sets of the edges of all the items in the groups.*

An item tile is characterized by the following information:

- an item collection,
- the tile tag, called **super-tag**,
- the set of items in the tile. Each of those items tag is called **sub-tag**.

The same restrictions as the step tiles are applied to the item tiles.

Restriction 6. *The iteration space of an item-collection is dense.*

Restriction 7. *An item tile forms a dense domain in the iteration domain of the step collection.*

Restriction 8. *Both the iteration space of an item collection and the item tile are of rectangular shape.*

With this definition of an item tile, a tile of item doesn't necessarily respect the Dynamic Single Assignment anymore, as each separate sub-tags can come from different steps. In order to maintain the DSA property on the coarse-level graph, the following restriction is added:

Restriction 9. *An item tile must be fully put by a unique step tile.*

Based on those information, for a given sub-tag of dimension n named $subtag_i$, a tile size $size_i$ and for a positive direction in each dimension $0 \leq i < n$, the super-tag $supertag_i$ is given by:

$$supertag_i = subtag_i \ \mathbf{div} \ size_i \tag{4.11}$$

In reverse, from a given super-tag $supertag_i$, all the sub-tags in the tile are expressed as:

$$size_i \times supertag_i \leq subtag_i < size_i \times (supertag_i + 1) \tag{4.12}$$

Coming back on the 1D accumulation on a 2D space 8, the tiling of A would be exactly the set of items produced by the tile of the step S.

The restriction applied to keep the DSA condition at the super-level is needed to consider the super-level graph with only super level steps and items to be a valid CnC graph. It then allow to consider a super-step as a specific implementation function instead of considering it as a CnC sub-graph to improve performance, even if this will not be implemented in this thesis.

Now that the notion of tiling have been developed for steps and items and that the legality condition for step tiling have been explained, the compiler implemented for this thesis will be described.

Chapter 5

The compiler

In this chapter, we will extensively describe the compiler that was implemented to automatically tile kernels and generate C++ code using the hierarchical CnC runtime. After describing the general workflow of the compiler, its intermediate representation (IR) and the core algorithms applied on it will be described.

5.1 Overview

The compiler takes as input a single-level or a multi-level kernel description. As neither a grammar nor a parser were implemented for this compiler, the input is given using the user-interface of the IR, here a set of methods of the root node of the IR, populating the AST. The figure 5.1 gives a high-level view of the compiler.

From the single-level IR, the single-level CnC C++ code can directly be generated. The compiler let the runtime deal with the possibility that the Given tiling targets for each step-collections, the single-level IR is transformed to a multi-level IR. This multi-level IR can also be directly created from the user-interface. The legality of the tiling is then verified before generating the C++ CnC code if the tiling found is legal. Adding external code (kernel constants, input data and main function), the binary can finally be generated. The steps execution code is provided using the IR user interface.

The compiler user also provides external tools such as a CCDFG builder and tracer that output a text representation of the CCDFG inferred by the compiler, a tracer option for the CnC C++ code generation that output in the same text format and a trace comparator that verifies that two traces are identical once sorted in order.

5.2 User Input

The user input corresponds to C++ code composed of the following parts:

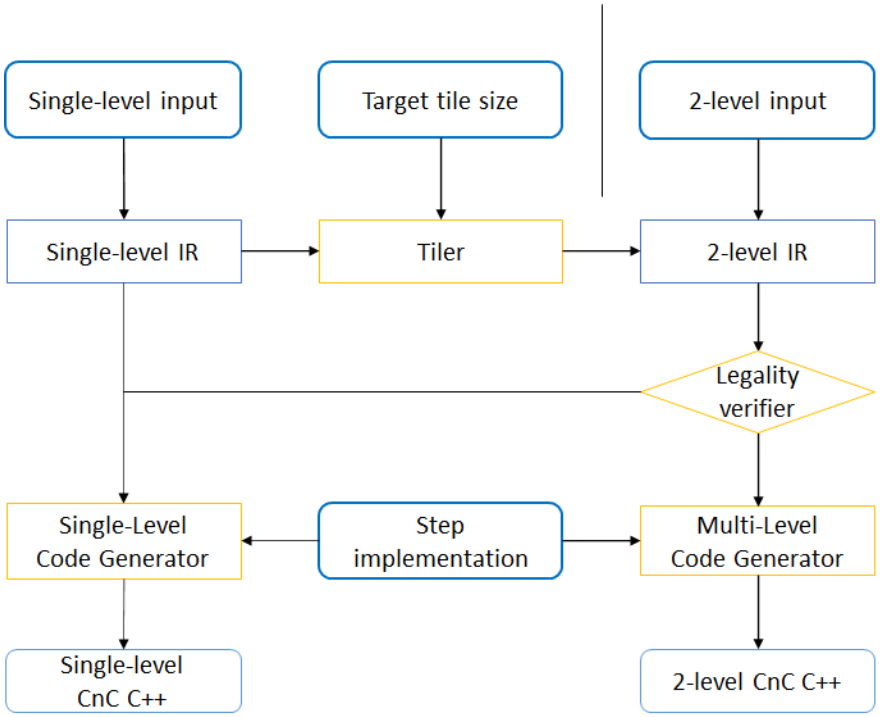

Figure 5.1: Compiler workflow

- Constants definitions
- Collections definitions
- Environment edges
- Steps edges

Algorithm 4: Example of a CnC specification graph

```
1 int N;
2 env → [A : 0];
3 env → (S : 0);
4 env ← [A : N];
5 (S : i) ← [A : i];
6 (S : i) → [A : i + 1];
7 (S : i) → (S : i + 1) if i + 1 < N;
```

Each constant is defined by its string name. A tag collection is defined by its name and its tag dimension. An item collection is defined by its name, its C++ data type, and its tag collection name. A step collection is defined by its name, its tag collection, and its execution code. Then, each edge of the corresponding CnC specification is defined, for the environment and each step collection, for which the tag variables first need to be defined.

As an example both the CnC graph specification and the corresponding user input of the compiler are given for a mono-dimensional kernel counting the number of steps executed 7. The steps code can use the items that the step will get and put using a name code based on the item collection name, on if the data is get or put by the step and on the index of the get/put edge the user chose.

Once the user input is done being specified, the compiler starts with a first pass to verify each name in the program, and the fact that each constants and variables have been defined.

```
#include "hcnc_ir.hh"

using namespace hcnc_ir;

Program program;

/* Context */
program.add_constant("N");
program.add_tag_type("Tag0", 1);
program.add_item_type("A", "float", "Tag0");
program.add_step_type("S", "Tag0", "A_out_0_ = A_in_0_ + 1;\n");

/* Env */
Int N = program.get_constant("N");
program.add_env_put("A", 0);
program.add_env_prescribe("S", 0);
program.add_env_get("A", N);

/* (S: i) */
Int i("i");
Boolean cond = lt(i + 1, N);
program.add_step_def("S", i);
program.add_step_get("S", "A", i);
program.add_step_put("S", "A", i + 1);
program.add_step_prescribe("S", "S", i + 1, cond);
```

5.3 Intermediate Representation

The IR is built as an Abstract Syntax Tree with as root the program node. At a high-level point of view, the program node contains the list of constants, collection types and functions defined, as well as a representation of the CnC specification graph. This architecture is shown figure 5.2.

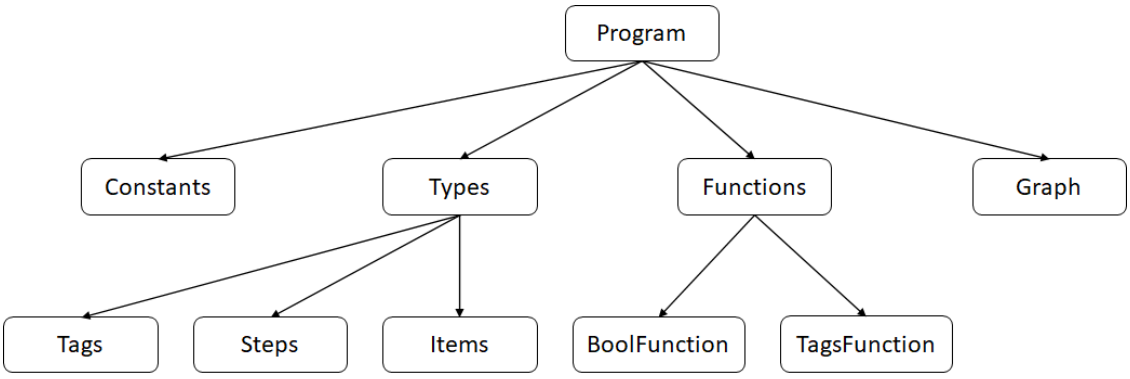

Figure 5.2: High level of the IR tree

The list of defined constants is used mainly to verify that every integer defined by the user in the functions section and in the specification graph exist. The collection types of section is used to attach the tag, item and step type to each elements of the specification graph. In the functions section, every function that are called in the specification graph must be defined.

Integer expressions

Integer expressions are managed in the IR as trees, with variables and concrete values as leaves and binary operators as inside nodes. This tree structure comes with a set of functionalities implemented like the replacement of a variable by an integer expression, the basic expression simplification for the tiling functionality, computation of an integer without variable...

Boolean expressions

Boolean expressions are managed in the IR the same way as integer expressions. The tree leaves can be a Boolean value, a condition or a function call. The tree internal nodes can be a binary operator or a unary operator. This tree structure comes with a set of functionalities like the replacement of integer variables in the tree or the evaluation of the Boolean concrete value.

Tags

Tags were defined previously as a tuple of integer. In order to model the fact that the environment or that step collections can have edges that contains a destination with multiple tags, the notion of Tags is implemented to accept the following regions:

- A unique tag represented as a tuple of integer expressions
- A rectangular region having for each dimension either a range either an integer expression
- A polyhedron defined by a set of variables, conditions for each variable and the output tag form.
- A union of polyhedrons
- A generic union of any of the above.

This implementation is based on the implementation created in the PIPES compiler [14], [30]. The implementation is based on a tree with leaves being the unique tag, the rectangular region and the polyhedron, and internal nodes being the polyhedra union and the generic union.

Functions

The user can define two types of functions: Boolean and tags function, taking as input a tag and returning respectively a Boolean or a tags object.

Graph

In the IR, the Graph node stores all the information of the CnC specification graph. We separate origin and destination of the edges of the graphs into different type. Origin of edges are the environment and a step being defined, which is associated to a tag of variables.

Destination of edges can be steps or items, that in the IR will be associated to a Tags instance, that can only depend on the program constants and on the variables in the tag of the step being defined.

Get, put and prescribe edges possess a pointer to its origin and its destination, as well as an execution condition that is a Boolean that can only depend on the program constants and on the variables in the tag of the step definition.

Hierarchy

The data associated to the hierarchical feature of the steps and items are attached to their definition. Those data are the rectangular size of the tile, an offset for each dimension. Steps also have as information the list of border tags to consider and the border of the iteration domain in each dimension.

5.4 Automatic tiling

The core algorithm that justifies the creation of this compiler is the automatized transformation from a single-level description of an irregular application to a two-levels implementation fully leveraging the performance potential of the Hierarchical Concurrent Collections runtime.

In order to take into account, the characteristics of irregular applications, a graph based approach is chosen.

5.4.1 Restrictions

- On the input program
  - The program is a Concurrent Collection program that terminates correctly
  - The number of steps in the program is finite both statically (upper-bound of the iteration space size in each dimension) and at run-time (real size of the problem iteration space)
  - The program is single-level
  - The application must be regular: no data dependent edges shall exist in the graph
  - The iteration space of both step and item collections are dense
  - A step can only be prescribed from the environment
  - The set of item and step tags put by the environment is limited to non-parametric range
  - The set of item tags put and got from a step is limited to a unique tag expressed as an affine function of the origin tag

  - Outside of the environment, only one step collection can write in a specific item collection
  - The problem size in each dimension must be a multiple of the target tile size in that dimension

- On the tiling
  - Working only on rectangular tiling

The restriction which imposes that only the environment can prescribe steps doesn't restrict the class of kernels that can be tiled but only their implementation, as the program is regular.

The restriction which imposes that the tiling is rectangular is not due to the algorithm, which could generate a non-rectangular tiling, but to the IR and the code generator, which only work with a super-level containing rectangular tiles.

5.4.2 Overview of the algorithm

The tiling algorithm transforms a single level program description on a two-levels program description containing the information needed to tile each step and item collections. A rectangular target tile size is specified for each step collection to tile.

After choosing the direction of tiling in each dimension, a generic tile shape is going to be iteratively grown for the step collection to be tiled. This generic tile will then be adapted to consider the border of the iteration space of the step. Then, each item collection in produced by the tiled step is tiled so that the set of items of the same collection is gathered in a tile. Finally, the validity of the tiling is verified by computing the transitive closure of the tiled graph and verifying the absence of cycles.

To completely specify the tiling in the IR, we need enough information to be able to produce the hierarchical tuners described section 3.3.2. we need to bring it the following information:

- Step tiling
  - A function associating a super-tag of variables to the polyhedron of sub-tags in the tile.

  - The number of elements in the tile at the border and inside the iteration domain.
  - A function associating a super-tag of variables to the set of sub-tags prescribed from outside the tile.
  - A function associating a super-tag of variables with, for each concerned item collections, the set tags both read and written in the step tile.

- Item tiling
  - A function associating a super-tag of variables to the polyhedron of sub-tags in the tile.

The algorithms main steps are, for a specific item collection:

1. Build the list of neighbors of a generic points to consider in the iterative growth.
2. Grow a generic tile from a point made of variables. It returns the function associating a super-tag to the polyhedron of sub-tags in the tile for the step collection, as well as the function associating a super-tag to the set of sub-tags prescribed outside of the tile.
3. Apply the tiling to the iteration domain (treat the case of the border of the iteration domain). It returns a function associating the super-tag to the number of elements in the tile.
4. Tile every item collection that are put by the step collection, as well as every item collection that are only put by the environment. It returns the function associating the super-tag to the number of elements in the tile for every item collection in the program.
5. Verify the validity of the tiling by computing the graph transitive closure and checking for the absence of cycle.

5.4.3 Building the list of neighbors

Blocking hypothesis

- The application must be regular: no data dependent edges in the graph

- The iteration space of both step and item collections are dense
- The tiling searched for is rectangular

This algorithm 5 describes how the neighbors of a variable point in the step space are computed. After finding the step definition to tile, the tiling direction is determined from the relation in each dimension (algorithm 6. Line 5 and 6 build for each dimension a pair $(0, 1)$ or $(0, -1)$ for each dimension. Then, each neighbor corresponds to an increment from the origin point following the cross product line 7.

Algorithm 5: Build neighbors list

Input:
Program p: program representation, root node of the IR
String name: name of the step to tile
Output:
$pair < Tag, set < Tag >>$: Pair associating to a variable tag the set of its neighbors

```
1  Tag origin = p.findStepDefinition(name).tag();
2  pair < Tag, set < Tag >> neighbors = (origin, set());
3  vector < int > direction = findTilingDirection(p, name);
4  vector < pair < int >> pointsPerDimension;
5  for dim = 0; dim < dimension(step.tagType); dim++ do
6      pointsPerDimension.pushBack({0, direction[i]});
7  for increment ∈ crossProduct(pointsPerDimension) do
8      Tag incrementTag(origin.type(), increment);
9      Tag neighbor = origin + incrementTag;
10     if neighbor == origin then
11         continue;
12     else
13         neighbors.second.insert(neighbor);
14 return neighbors;
```

5.4.4 Step tile growth

Blocking hypothesis

- The application must be regular: no data dependent edges in the graph

Algorithm 6: Find Tiling Direction

Input:
Program p: program representation, root node of the IR
String name: name of the step to tile
Output:
vector<int> direction: Tiling direction in each dimension ($+1$ or -1))

```
1  vector < int > direction;
2  step = p.findStepDefinition(name);
3  for dim = 0; dim < dimension(step.tagType); dim++ do
4      if hasDependency(step, dim) then
5          if dependencyIsBidirectional(step, dim) then
6              throw Error;
7          else
8              direction.pushBack(dependencyDirection(step, dim))
9      else
10         direction.pushBack(1);
11 return direction;
```

- The iteration space of both step and item collections are dense
- The set of item tags put and got from a step is limited to a unique tag expressed as an affine function of the origin tag.
- Only one step collection is allowed in the program.
- The tiling searched for is rectangular

The tile growing algorithm is given algorithm 15. Line 1 to 4 initialize the algorithm variables. The $Tile$ class contains the information relative to a tile. It maintains the set of tags in the tile, as well as the set of items required and produced by the tile and the list of steps prescribed by the tile. When a tag is added to the tile, all those sets are updated by applying the dependency relations defined in the program to the tag. The $Tile$ class is also able to generate the set of tags at the border of the tile in the tiling direction. Line 5 adds the point of origin to the tile. The growing process of the tile iteratively go through the current neighbors of the tile in the tiling direction and add them to the tile if it can be executed until the target tile size is reached (line 6) or if the algorithm cannot

Algorithm 7: Grow tile

Input:
Program p: program representation, root node of the IR
String step: name of the step to tile
vector<int> direction: Tiling direction in each dimension ($+1$ or -1))
$pair < Tag, set < Tag >>$ neighborsTemplate: Pair associating to a variable tag the set of its neighbors volume: maximum volume of the tile
Output:
StepTile tile: object containing the information on the tile: super tag, set of subtags, of required and produced items.

```
1  begin
2      StepTile tile(p, step, direction);
3      tag = v_0...v_dim
4      hasChanged = false;
5      tile.add(tag)
6      while tile.size() < maxVolume do
7          hasChanged = false;
8          for point ∈ tile.border() do
9              for neighbor ∈ buildNeighborList(point, neighborTemplate) do
10                 if tile.canBeExec(neighbor) then
11                     tile.add(neighbor);
12                     hasChanged = true
13         if not(hasChanged) then
14             break;
15     return tile;
```

add any step to the tile (line 14 to 15). The current neighbors of the tile are enumerated by the loops line 8 and 9, with redundancy. A step can be executed (line 10) if the step producing each data dependence is before the tile or inside the tile. A step is said to be before the tile if it is the environment, if it is a step type different from the step being tile or if the tag of the step is inferior to the steps in the tile regarding to the data and prescription dependencies.

5.4.5 Application of the tiling to the iteration space

For now, and until the hypothesis on the problem size in each dimension being a multiple of the target tile size on that dimension, is lifted, this part of the algorithm is empty.

5.4.6 Tiling of the item collections

Blocking hypothesis

- Outside of the environment, only one step collection can write in a specific item collection

This part of the algorithm tiles all the item collections in the program. To optimize performance, we try to make it so that each step tile creates exactly a tile of each item it produces. So, the algorithm loop for each step collection and starts by determining which step collection produces which item collection, and the list of tags created by the step tile for this item collection. The corresponding rectangle is then computed for this item collection as well as the offset from the origin for the item tile.

5.4.7 Verification of the tiling correctness

To verify the tiling legality, we implement the algorithm introduced section 4.2.2. This algorithm has a complexity of $\Theta(n^3)$ on the number of item and steps in the program, and is thus extremely time consuming. To reduce the algorithm complexity, and to simplify the research of cycles, all items are removed of the graph on which the transitive closure is computed. Once the transitive closure is computed, the tiling is correct if all the values in the matrix diagonal are equal to zero.

5.4.8 Code generation

This section will describe the code generation for a tiled version of matrix multiply, defined by the algorithm 12.

The code generation starts with the declaration of the different classes needed by the runtime, then implement the tags and items methods needed to work in distributed, and finally implement each method of the tuners and the steps and a run function that instantiate the context, give the loop-nests needed to for the user to put, prescribe and get from the environment.

Algorithm 8: Matrix multiply kernel

```
1  int M;
2  int N;
3  int P;
4  env → [A : 0..M, 0..P];
5  env → [B : 0..P, 0..N];
6  env → [C : 0..M, 0..N, 0];
7  env → (S : 0..M, 0..N, 0..P);
8  env ← [C : 0..M, 0..N, P];
9  (S : i, j, k) ← [A : i, k];
10 (S : i, j, k) ← [B : k, j];
11 (S : i, j, k) ← [C : i, j, k];
12 (S : i, j, k) → [C : i, j, k + 1];
```

This whole process is done in three main traversals of the IR. The first traversal does all the declarations but the context, the second traversal declares the context, and the last traversal implement all the methods.

Collections declarations

The first traversal starts by looping over every tag type to define its type and its hashing function. This first code part is represented in the listing 5.1. The hashing algorithm used is a variant of [50].

```
namespace std {
    unsigned int _compute_hash(unsigned int h, unsigned int ki) {
        unsigned int highorder;
        highorder = h & 0xf8000000;
        h = h << 5;
        h = h ^ (highorder >> 27);
        return h ^ ki;
    }
}
typedef std::tuple<unsigned int, unsigned int> Tag2D;
namespace std {
    template<>
    struct hash<Tag2D> {
        size_t operator()(const Tag2D &k) const {
            unsigned int h = 0;
            h = _compute_hash(h, std::get<0>(k));
            h = _compute_hash(h, std::get<1>(k));
            return h;
        }
    };
}
```

Listing 5.1: Tag type declaration

Then, for each multi-level item types, the item tile type and the item hierarchy tuner are defined. The item tile contains an array of the size of the number of fine level items in the tile as well as a counter to be used by the item hierarchy tuner. Then the item hierarchy tuner is declared with the tile size as attribute and the *getSuperTag*, *getFromSuperItem* and *putInSuperItem* methods. The corresponding code is given in the listing 5.2 for a 2D step collection named A with a tile of 6 elements.

```
struct Atile {
    int cnt{0};
    float values[6];
};
struct AHierarchyTuner : public cnc::ItemCollectionHierarchyTuner<Tag2D, Tag2D
    , float, Atile> {
    bool getSuperTag(const Tag2D &tag, Tag2D &superTag) const;
    void getFromSuperItem(const Tag2D &tag, float *item, Atile const *
    superItem) const;
    bool putInSuperItem(const Tag2D &tag, float *item, Atile *superItem) const
    ;
};
```

Listing 5.2: Item tile and hierarchy tuner declaration

Then, for each multi-level step types, its hierarchy tuner is declared with the tile size as attribute and the *getSuperTag*, *subTagPut* and *getBorderTags* methods. If there are any items that are both read and written by a tile, an additional tuner is added to pre-declare this item. Finally, for each step collections defined in the specification graph, the step structure is defined with the execute method as well as container attributes for each get and put edge in interaction with this step. The corresponding code is given in the listing 5.3, for matrix multiply.

```
struct SHierarchyTuner : public cnc::StepCollectionHierarchyTuner<Tag3D, Tag3D
    > {
    int cnt{24};
    static bool getSuperTag(const Tag3D &tag, Tag3D &superTag);
    bool subTagPut(const Tag3D &tag);
    std::array<Tag3D, 24> getBorderTags(const Tag3D superTag) const;
};

struct SMicroRuntimeTuner {
    static void preExecute(const Tag3D &tag, Context &ctx);
};

struct S : public cnc::StepBase {
    void execute(Tag3D &tag, Context &ctx);
    float A_in_0_;
    float B_in_1_;
    float C_in_2_;
    float C_out_0_;
};
```

Listing 5.3: Step, hierarchy tuner and pre-execution declaration

Context declaration

The second traversal declares the context class. First, the context constructor is defined to have as arguments every program constant. Then all the constants are defined as attributes of the class, followed by each item and step collections. The corresponding code is given in the listing 5.4 for matrix multiply. The long lines were split to enhance readability.

```
1  struct Context {
2      Context(unsigned int M, unsigned int N, unsigned int P) :
3          M_{M}, N_{N}, P_{P}, A_(this, "A"),
4          B_(this, "B"), C_(this, "C"), S_(this, "S") {}
5
6      unsigned int M_;
7      unsigned int N_;
8      unsigned int P_;
9      cnc::ItemCollection<
10         Context,
11         cnc::ItemCollectionHierarchyLevel<Tag2D, float>,
12         AHierarchyTuner,
13         cnc::ItemCollectionHierarchyLevel<Tag2D, Atile>
14     > A_;
15     // ... Idem for item collections B and C
16     cnc::StepCollection<
17         Context,
18         cnc::StepCollectionHierarchyLevel<Tag3D, S>,
19         SHierarchyTuner,
20         cnc::StepCollectionHierarchyLevel<
21             Tag3D,
22             cnc::DefaultMicroRuntime<
23                 Context,
24                 S,
25                 SHierarchyTuner,
26                 SMicroRuntimeTuner>>
27     > S_;
28 };
```

Listing 5.4: Context declaration

Lines 9 to 14 define the item collection A with two levels. The collection is associated to the context (line 10), has a fine level associating the tag collection *Tag2D* to a float (line 11) and a coarse level associating the tag collection *Tag2D* to the type *Atile* declared in the listing 5.2 (line 13). The tiling is managed by the corresponding hierarchy tuner (line 12).

Lines 16 to 27 define the step collection S with two levels. The collection is associated to the context (line 17) and has a fine level associating the tag collection *Tag2D* to the step type defined in the listing 5.3 (line 18). Here, the coarse level is managed as a CnC micro-runtime, i.e., the tile is considered as a sub-graph that can be executed. This micro-runtime is associated to the context (line 23), execute the step collection S (line 24) and is managed by the corresponding hierarchy tuner and pre-executor (lines 25 and 26). In the main graph, the hierarchy is managed by the corresponding hierarchy tuner (line 19).

Steps, tuner and run function implementation

The third traversal implements every method as well as the main run function. For each multi-level item and step collections, the hierarchy tuners methods are implemented using the data stored in the IR. The detail of how those data are generated are given in 4.2.1, The corresponding C++ code won't be described as it basically translate what is described in this section. Then, each user-defined functions associating a tag to a set of tags and associating a tag to a Boolean are implemented using the code given in the IR.

For each step collections defined in the graph specification, the execute method is implemented using the following process:

1. For each get edge, the items accessed by the edge are put in the corresponding step public attribute.

2. The user-defined step code is then added. This code needs to write the output in the class attributes corresponding to the put edge defined for those outputs.

3. For each put edge, the corresponding attribute is given to item collection in the context.

4. For each prescribed steps, the corresponding tag is given to the step collection in the context.

```
void S::execute(Tag3D &tag, Context &ctx) {
    /* Tag variables declaration */
    unsigned int i = std::get<0>(tag);
    unsigned int j = std::get<1>(tag);
    unsigned int k = std::get<2>(tag);
    /* Gets */
    gets([&]{
        {
            Tag2D new_tag = std::make_tuple(i, k);
            ctx.A_.get(new_tag, A_in_0_);
        }
        {
            Tag2D new_tag = std::make_tuple(k, j);
            ctx.B_.get(new_tag, B_in_1_);
        }
        {
            Tag3D new_tag = std::make_tuple(i, j, k);
            ctx.C_.get(new_tag, C_in_2_);
        }
    });
    /* Implementation of the step goes there */
    /* Puts */
    {
        Tag3D new_tag = std::make_tuple(i, j, (k + 1));
        ctx.C_.put(new_tag, C_out_0_);
    }
    /* Prescribes */
}
```

Listing 5.5: Context declaration

Finally, the run method is implemented. This method instantiates the context, starts the runtime, put every item, and prescribe every step in the context, call the *wait* function of the runtime before getting in the context all the output of the kernel. To deal with the multiprocessor, all the gets, puts, and prescribes call are executed only by the root process. The corresponding code generated is given in the listing 5.6.

The user of the compiler can then use the run function to call the kernel. This function takes as arguments a Boolean indicating if the thread is the runtime root thread, the kernel constants as well as a reference to a map associating a tag to its item instance for every item collection interacting with the environment.

```
typedef std::map<Tag2D, float> Item2D;
typedef std::map<Tag3D, float> Item3D;
void run(bool isRoot, unsigned int M, unsigned int N, unsigned int P, const
    Item2D& A, const Item2D& B, Item3D& C) {
    Context ctx(M, N, P);

    cnc::start();

    if (isRoot) {
        for (unsigned int u0 = 0; u0 < M; u0 += 1)
            for (unsigned int u1 = 0; u1 < P; u1 += 1) {
                Tag2D new_tag = std::make_tuple(u0, u1);
                ctx.A_.put(new_tag, A[new_tag]);
            }
            for (unsigned int u0 = 0; u0 < P; u0 += 1)
                for (unsigned int u1 = 0; u1 < N; u1 += 1) {
                    Tag2D new_tag = std::make_tuple(u0, u1);
                    ctx.B_.put(new_tag, B[new_tag]);
                }
            for (unsigned int u0 = 0; u0 < M; u0 += 1)
                for (unsigned int u1 = 0; u1 < N; u1 += 1) {
                    Tag3D new_tag = std::make_tuple(u0, u1, 0);
                    ctx.C_.put(new_tag, C[new_tag]);
                }
            for (unsigned int u0 = 0; u0 < M; u0 += 1)
                for (unsigned int u1 = 0; u1 < N; u1 += 1)
                    for (unsigned int u2 = 0; u2 < P; u2 += 1) {
                        Tag3D new_tag = std::make_tuple(u0, u1, u2);
                        ctx.S_.put(new_tag);
                    }
    }

    cnc::wait();

    if (isRoot) {
        for (unsigned int u0 = 0; u0 < M; u0 += 1)
            for (unsigned int u1 = 0; u1 < N; u1 += 1) {
                Tag3D new_tag = std::make_tuple(u0, u1, P);
                float tmp7;
                ctx.C_.get(new_tag, tmp7);
                C[new_tag] = tmp7;
            }
    }
}
```

Listing 5.6: Context declaration

Chapter 6

Experimental results

The goal of this chapter is to give the user recommendation for the value to give to the target tile size for the tiling algorithm. The hierarchical runtime overhead will be characterized depending on the number of coarse-grain steps in the program, as well as the number of sub-steps in one super-step. The experiments will run on synthetic benchmarks representing classic linear-algebra dependency pattern.

6.1 Experimental setup

6.1.1 Cluster

This work was done on the Summit supercomputer [51], [52], which is supported by the National Science Foundation (awards ACI-1532235 and ACI-1532236), the University of Colorado Boulder, and Colorado State University. The Summit supercomputer is a joint effort of the University of Colorado Boulder and Colorado State University. The experiments are run on SkyLake nodes, which contains two Intel® Xeon® Gold 6126 CPU [53]. The specification of this CPU is given table 6.1.

Table 6.1: CPU specification

CPU name	Intel® Xeon® Gold 6126
Frequency	2.60GHz
Logical cores	12
Threads per core	1
Cache line size	64B
L1 Data cache	32KB
L1 Instriction cache	32KB
L2 cache	1024KB
L3 cache	19712KB

6.1.2 Kernels

The kernel studied is a 3D synthetic benchmark representing a classic linear-algebra dependency pattern, described algorithm 12. Each item is chosen to be of the C++ data type $float$. No dependencies exist in the first two dimension, and a step depends on the previous one along the third dimension. The single and multilevel implementations are generated from the compiler described chapter 5. The code was instrumented with Polybench [54] to produce the experimental results.

Algorithm 9: Synthetic Linear Algebra kernel under study

```
1  int M;
2  int N;
3  int P;
4  env → [A : 0..M, 0..P];
5  env → [B : 0..P, 0..N];
6  env → [C : 0..M, 0..N, 0];
7  env → (S : 0..M, 0..N, 0..P);
8  env ← [C : 0..M, 0..N, P];
9  (S : i, j, k) ← [A : i, k];
10 (S : i, j, k) ← [B : k, j];
11 (S : i, j, k) ← [C : i, j, k];
12 (S : i, j, k) → [C : i, j, k + 1];
```

6.2 Experimental results

To evaluate the runtime performance, we define the execution time of the kernel as the execution time between the start of the runtime execution to the end of the extraction of the kernel output once the runtime has terminated. For each configuration, the kernel is run three times, and the minimum execution time is kept. We define the throughput as the number of tasks in the program divided by the execution time. For every experiment presented below, the throughput values will be adapted to get a value range that improves readability.

In this section, we will evaluate the strong and the weak scaling of the kernel presented previously for both the single-level implementation and for different tile sizes.

The strong scaling characterizes how the throughput evolves as, for a fixed total amount of computation, the number of computational resources is increased [55]. Analyzing the strong scaling allows the analysis of how much of the program can be parallelized and how much of the program needs to be executed sequentially for a specific kernel size. As the number of computational resources increases, the throughput is expected to stabilize at some point as the sequential part of the program becomes prominent. For the upcoming experiments, strong scaling will be studied on square problem sizes ($M = N = P$) with the values of 100, 200 and 400.

The weak scaling characterizes how the throughput evolves as, for a fixed amount of computation per computational resource, the number of computational resources is increased. [55]. Studying the weak scaling of an application allows the analysis of the evolution of the sequential part of the program as the problem size and the number of resources evolve, as the workload per resource stays constant.

6.2.1 Strong scaling of the single-level implementation

First, the strong scaling of the runtime overhead will be characterized on the single-level implementation as given by the user as a CnC specification graph. All of the parameters of this experiment have been specified previously: square problem sizes of 100, 200 and 400, and a number of threads between 1 and 12. The results of the strong scaling of the single-level implementation are given figure 6.1.

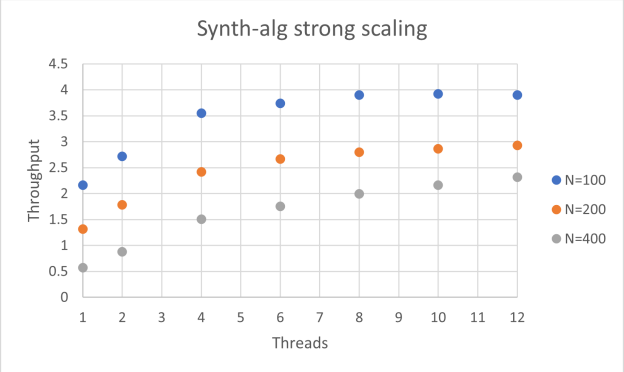

Figure 6.1: Fine grain synthetic linear algebra kernel strong scaling

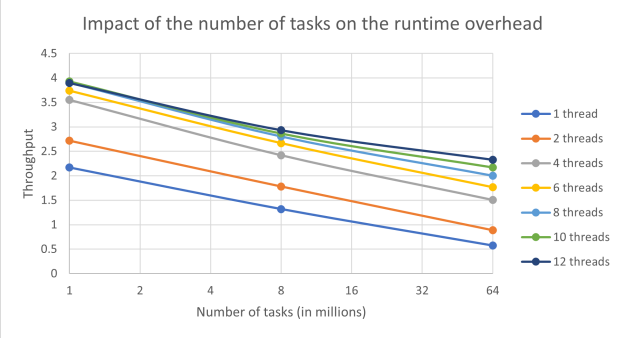

Figure 6.2: Impact of the number of tasks on the runtime overhead

First, the strong scaling is evolving similarly for the three problem sizes presented. The kernel scales linearly between 1 and 4 threads, slowly increases between 4 and 8 threads and finally stabilizes between 8 and 12 threads.

The figure 6.1 also shows that the throughput decreases significantly as the problem sizes increase. For 12 threads, the throughput goes from 4 to 3 as the problem size increases from 100 to 200 in each dimension, and then goes from 3 to 2.4 as the problem size increases from 200 to 400.

The figure 6.2 presents another view the same experimental results, view presenting the throughput depending on the number of tasks in the kernel.

This plot shows that the tendency is almost linear (on a linear-log base 2) representation for all the number of threads studies, but that the throughput isn't as linear for higher numbers of threads. This plot allows us to infer that, for this problem size range and data dependency pattern, the runtime throughput decreases logarithmically as the number of tasks increases.

6.2.2 Strong scaling of the tiled implementation

Now that the strong scaling of the runtime overhead has been characterized for the fine grain implementation of the synthetic linear algebra kernel, the next step is to evaluate the impact of the tiling and of the tile size chosen on the runtime overhead.

For those experiments, the tile sizes will be chosen as square sizes $B_M = B_N = B_P$ with B_M, B_N and B_P the tile size chosen in their respective dimensions. The kernel will be evaluated with

the following square tile sizes of values 1, 2, 5, 10, 25, 50 and 100, and we will evaluate it for fixed square problem size of 100, 200 and 400.

Table 6.2: Number of super-steps in the runtime and of sub-steps per micro-runtime for each problem and tile sizes.

Tile size	1	2	5	10	25	50	100
Problem size 100							
Tasks in runtime	1000000	125000	8000	1000	64	8	1
Tasks in micro-runtime	1	8	125	1000	15625	125000	1000000
Problem size 200							
Tasks in runtime	8000000	1000000	64000	8000	512	64	8
Tasks in micro-runtime	1	8	125	1000	15625	125000	1000000
Problem size 400							
Tasks in runtime	64000000	8000000	512000	64000	4096	512	64
Tasks in micro-runtime	1	8	125	1000	15625	125000	1000000

This experiment allows the analyses of the impact of the tile size on the for tiles going from containing only one step (tile size of 1) to for a problem size of 100 a unique tile containing the full kernel (tile size of 100). The number of super-steps in the runtime and the number of sub-steps in each micro-runtimes are presented table 6.2. We are expecting the kernel throughput to increase as the tile size increase as it reduces the number of tasks that the runtime is managing, until a point where the number of tasks in each micro-runtime increases so much that the micro-runtimes overhead mitigates the decrease of the main runtime overhead.

The points described in this part with a tile size of 1 correspond to the fine-grain implementation as given by the user, and not by an actual implementation with call to micro-runtimes containing a unique sub-step. It means that the throughput for those parameters are the same as the ones presented figure 6.2.

The strong-scaling and the impact of the tile size on the runtime overhead are presented figure 6.3. The strong scaling for tile sizes shows that tile sizes that implies a huge number of super-steps implies a low throughput (here tile sizes of 1 and 2) for all the presented problem sizes.

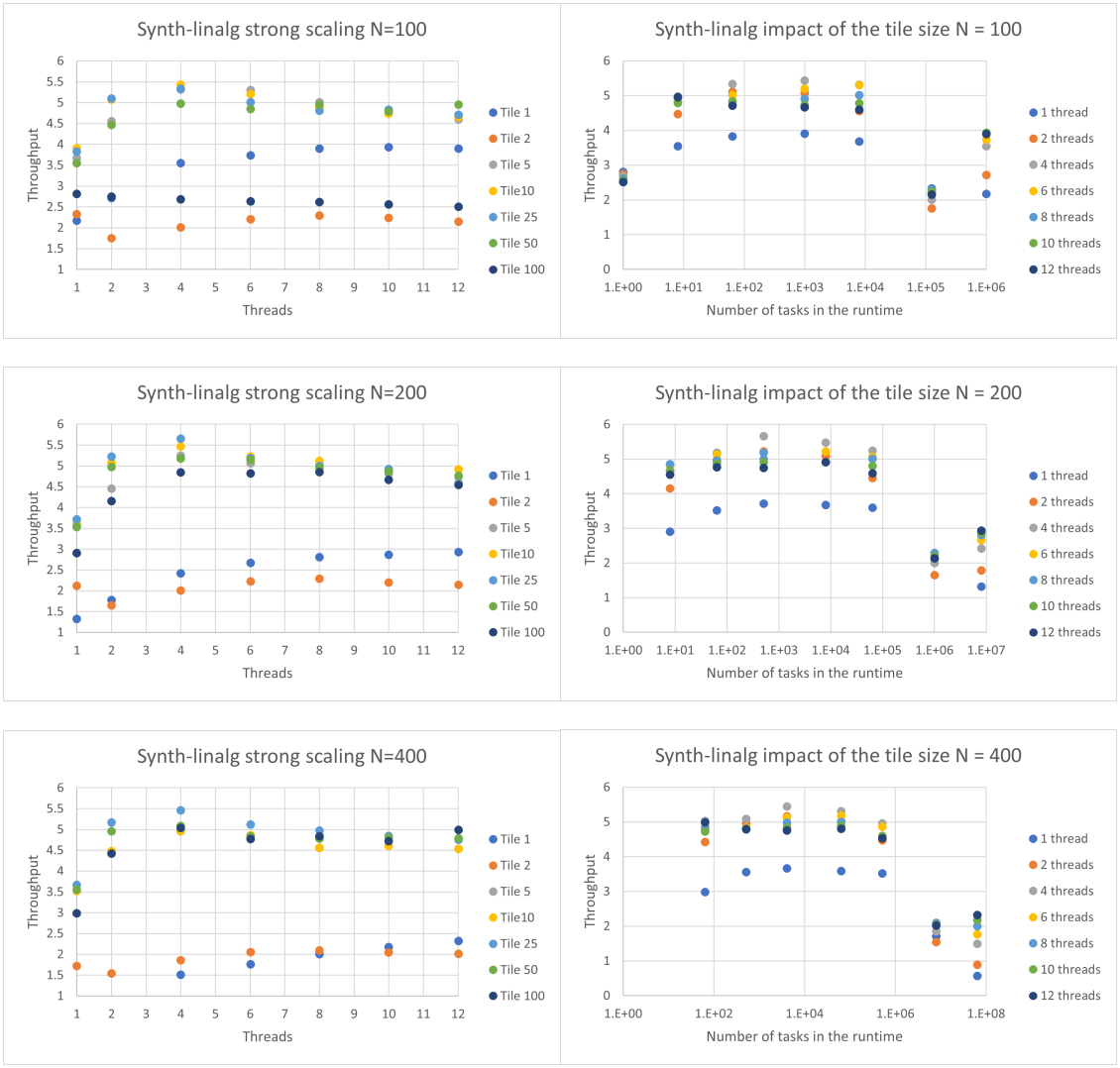

Figure 6.3: Synthetic linear algebra kernel strong scaling and impact of the tile size

The tile size of 100 for a problem size of 100 is specific as it represents the case where the runtime has only one super-step containing the whole kernel. As there is only one super-step and that each super step is executed sequentially, the throughput is slightly decreasing as the number of threads is increased.

For tile size superior to 2, the throughput increases significantly compared to the other cases, and the throughput increases from 3.5 to up to 5.5 as we increase the number of threads from 1 to 4, before slowly decreasing as the number of threads increases from 4 to 12. This shows that in the

runtime overhead by itself the parallelized part becomes small in front of the sequential part of the overhead for more than 4 threads.

The graphs presenting the impact of the tile size in the figure 6.3 confirm that there is a balance to find between the number of super-steps in the runtime and the number of sub-steps in every micro-runtimes. For the studied problem sizes for the synthetic linear algebra kernel with number of tasks between 1 and 64 million, the optimal results are obtained for tiles containing 1000 and 15625 sub-steps.

This specific experiment shows that, for this specific kernel, the best way to minimize the runtime overhead is to try to reduce the number of super-steps in the runtime while keeping the number of sub-steps in every micro-runtime under the order of 10000 sub-steps.

6.2.3 Weak scaling of the runtime overhead

For this experiment, we consider the synthetic linear algebra benchmark where each super-step contains 8000 sub-steps (square tile size of 20) and where each core executes 4 of those super-steps, such that each of those super-steps are independent. The scaling is studied between 1 and 8 cores.

As the workload of the synthetic linear algebra kernel under study evolves following $M \times N \times P$, a square problem size cannot be kept as it would mean that the number of resources would need to evolve in a cubic way, which is very limited for a CPU of 12 threads. In order to study the best-case scenario parallelism wise, the problem size will be increased on the first two dimensions as there is a data dependency along the third dimension. The problem size chosen are given table 6.3.

Table 6.3: Weak scaling problem dimensions

Threads	1	2	4	6	8	10	12
M	40	80	80	80	160	80	160
N	40	40	80	120	80	200	120
P	20	20	20	20	20	20	20
Super-steps	4	8	16	24	32	40	48

The results are represented figure 6.4. The kernel scales linearly between 1 and 4 threads, and then stabilizes between 4 and 12 cores at a throughput around 4.25.

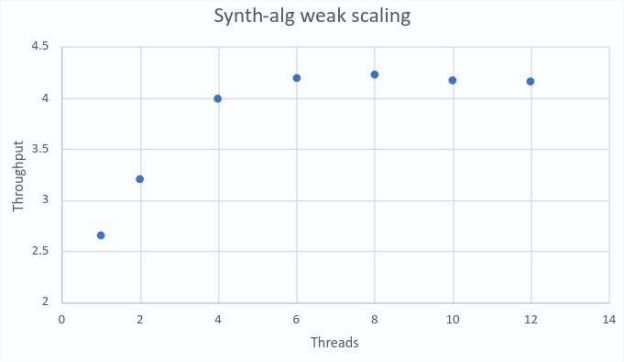

Figure 6.4: Synthetic linear algebra kernel weak scaling

As the workload for each thread is kept constant, the increase of the throughput between 1 and 4 threads shows that the part of the overhead that can be parallelized (which contains actions like the get, put and prescribe of each step, etc...) is prominent enough compared to the sequential part so that throwing more resources reduces the overhead. After 4 threads, the sequential part of the overhead has become prominent enough, due to the reduction of the parallelized part, so that the kernel stops scaling.

Now, with more computationally intensive sub-steps in the synthetic linear algebra kernel under study, it would be expected to observe a weak-scaling that would continue to scale after 4 threads.

Chapter 7

Conclusion

Context

Implementing a computation intensive kernel on a distributed memory system is usually done using the MPI library interface to pass messages between non-shared memory computational resources but comes at a high development time cost. Task based programming can alleviate similar computational performance while significantly reducing the development cost. The separation of concern between the domain expert, who implements the different tasks in the kernel and the tuning expert, who oversees implementing its distribution, also helps reducing the development cost by reducing the technical knowledge needed to implement such kernels. The execution of the distributed kernel is the responsibility of a software called a distributed runtime, such as Intel Concurrent Collections.

The heterogeneity of current supercomputers brings the necessity for a hierarchical representation of task-based programming. As such, a new hierarchical runtime implementing the Concurrent Collections programming model have been implemented, and this thesis implements parts of a compiler that generate code for this runtime from a simplified representation of this kernel.

Contributions

The work in this thesis was mainly focused on the automatic tiling of regular mathematical kernel using a graph representation of the program. This thesis main work was to design an algorithm to tile the step collections and the item collections used in a Concurrent Collection graph specification, to implement this algorithm in a compiler that generate C++ code using a hierarchical CnC runtime, and to characterize the overhead of this runtime to give recommendations on the best way to find the target tile size for the algorithm.

This thesis achieves the automation of step and item tiling using a graph approach for a regular single level Concurrent Collection program that terminates correctly, with a finite number of steps and items statically and at runtime, in which step and item collections iteration space are dense.

Those programs are restricted by only allowing steps to be prescribed from the environment, set of item and step tags put by the environment limited to non-parametric range. Finally, the set of item and step tags put and got from a step is limited to a unique tag expressed as an affine function of the origin tag.

The approach of the tiling algorithm implemented in this thesis is based on a graph approach in which neighbors' steps from the same collection are aggregated while maintaining the sets of items put and get by the aggregate. Steps are aggregated one by one in their dense iteration space, if they can be executed once the current tile is done executing, until a tile of shape as close as possible from the target tile is reached. The tiling found is then adapted to the iteration space borders, and the tiled CDFG is then built, its transitive closure computed, and a check is done to verify that there is no cycle in it, guaranteeing that the tiling found is legal. Finally, item collections are tiled so that each step tile generates exactly one item tile per collection.

The algorithm is implemented for an IR that was built for this thesis, which can both represent a single- and two-level Concurrent Collection program, and has a C++ code generator calling the hierarchical CnC runtime.

The tiling algorithm requires the user to specify a target tile size for each step collections, and the chapter 6 characterize the overhead of the hierarchical runtime on a shared memory system, on a 3D synthetic benchmark representing a classic linear-algebra dependency pattern. The main recommendation is to choose the tiles volume so that it minimizes the number of super-steps in the runtime while keeping the number of sub-steps in every micro-runtime under the order of 10000 sub-steps.

Future work

From this point on, the IR and the algorithm can be improved to reduce the number of restrictions on the input program.

The restriction on the prescriptions that can only come from the environment can be removed by considering it as a data dependency. The algorithm 15 use only the data dependency, thus not considering the prescription dependency. By considering the prescription relation as a data

dependency, i.e., that the prescribing step put an item in a dedicated fake item collection, and that the prescribed step gets this item. By adding those new dependencies, the growing algorithm gets all the information needed to run correctly.

The restriction on the shape of the tiles can be lifted by replacing the rectangular representation of a tile by a polyhedron representation. Currently, the algorithm 15 is able to generate a correct non-rectangular tiling shape, such as a triangle or a trapeze for the Jacobi 2D kernel. The parts that need modification are the application of the tiling to the iteration space by considering its borders, to adapt the tile representation in the IR from a rectangle representation to the existing polyhedron representation, and to modify the code generator to manage such cases.

The main restriction on the input program is that it needs to be a regular application. In an irregular application, the graph edges are conditionally used in the graph depending on the kernel data in the item collections. A first approach to extend the tiling algorithm would be to approximate the irregular application as a regular application and to try to find a valid tiling on this approximation. A brute force approach would be to consider all the possibilities for the conditionals and to call the tiling algorithm until a valid tiling is found.

To build a more complete compiler, the IR also need to let the user specify a particular implementation for super-level steps implementation, so that the super-step is executed with it instead as using a micro-runtime calling the sub-step implementation on every sub-steps. The compiler is also lacking a parser for the user to define the CnC graph with a user-language. Finally, the IR should be extended to accept placement tuning of the kernel steps and to pre-declare where each item are produced and consumed.

Bibliography

[1] Is it time for heterogeneous supercomputing? https://www.hpcwire.com/2006/07/14/is_it_time_for_heterogeneous_supercomputing/.

[2] Intel bets on heterogeneous future for supercomputing. https://www.zdnet.com/article/intel-bets-on-heterogeneous-future-for-supercomputing/.

[3] Top 500 supercomputers november 2021. https://www.top500.org/lists/top500/2021/11/.

[4] Lyndon Clarke, Ian Glendinning, and Rolf Hempel. The mpi message passing interface standard. In *Programming environments for massively parallel distributed systems*, pages 213–218. Springer, 1994.

[5] Wikipedia mpi. https://en.wikipedia.org/wiki/Message_Passing_Interface.

[6] Richard Adjogah, Randal Mckissack, Ekene Sibeudu, Andrew M Raim, Matthias K Gobbert, and Loring Craymer. Intel concurrent collections as a method for parallel programming. *UMBC Mathematics and Statistics Department*, 2011.

[7] Intel cnc github page. https://icnc.github.io/index.html.

[8] Randal Mckissack. Cluster computing using intel concurrent collections. *UMBC Student Collection*, 2012.

[9] George Bosilca, Aurelien Bouteiller, Anthony Danalis, Mathieu Faverge, Thomas Hérault, and Jack J Dongarra. Parsec: Exploiting heterogeneity to enhance scalability. *Computing in Science & Engineering*, 15(6):36–45, 2013.

[10] Parsec. https://icl.utk.edu/parsec/software/index.html.

[11] Chenyang Liu and Milind Kulkarni. Evaluating performance of task and data coarsening in concurrent collections. In *International Workshop on Languages and Compilers for Parallel Computing*, pages 331–345. Springer, 2016.

[12] Zoran Budimlić, Michael Burke, Vincent Cavé, Kathleen Knobe, Geoff Lowney, Ryan Newton, Jens Palsberg, David Peixotto, Vivek Sarkar, Frank Schlimbach, et al. Concurrent collections. *Scientific Programming*, 18(3-4):203–217, 2010.

[13] Michael G Burke, Kathleen Knobe, Ryan Newton, and Vivek Sarkar. The concurrent collections programming model. Technical report, 2010.

[14] Martin Kong, Louis-Noël Pouchet, P Sadayappan, and Vivek Sarkar. Pipes: a language and compiler for task-based programming on distributed-memory clusters. In *SC'16: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 456–467. IEEE, 2016.

[15] Alina Sbîrlea, Yi Zou, Zoran Budimlic, Jason Cong, and Vivek Sarkar. Mapping a data-flow programming model onto heterogeneous platforms. *ACM SIGPLAN Notices*, 47(5):61–70, 2012.

[16] Madness. https://github.com/m-a-d-n-e-s-s/madness.

[17] Robert J Harrison, Gregory Beylkin, Florian A Bischoff, Justus A Calvin, George I Fann, Jacob Fosso-Tande, Diego Galindo, Jeff R Hammond, Rebecca Hartman-Baker, Judith C Hill, et al. Madness: A multiresolution, adaptive numerical environment for scientific simulation. *SIAM Journal on Scientific Computing*, 38(5):S123–S142, 2016.

[18] Jack B Dennis. First version of a data flow procedure language. In *Programming Symposium*, pages 362–376. Springer, 1974.

[19] Edgar Gabriel, Graham E. Fagg, George Bosilca, Thara Angskun, Jack J. Dongarra, Jeffrey M. Squyres, Vishal Sahay, Prabhanjan Kambadur, Brian Barrett, Andrew Lumsdaine, Ralph H. Castain, David J. Daniel, Richard L. Graham, and Timothy S. Woodall. Open MPI: Goals, concept, and design of a next generation MPI implementation. In *Proceedings, 11th European PVM/MPI Users' Group Meeting*, pages 97–104, Budapest, Hungary, September 2004.

[20] Intel message passing interface. https://www.intel.com/content/www/us/en/develop/documentation/mpi-developer-guide-linux/top.html.

[21] High-performance portable mpi. https://www.mpich.org/.

[22] David Gelernter. Generative communication in linda. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 7(1):80–112, 1985.

[23] Philippe Charles, Christian Grothoff, Vijay Saraswat, Christopher Donawa, Allan Kielstra, Kemal Ebcioglu, Christoph Von Praun, and Vivek Sarkar. X10: an object-oriented approach to non-uniform cluster computing. *Acm Sigplan Notices*, 40(10):519–538, 2005.

[24] X10 language documentation. http://x10-lang.org/documentation/intro/latest/html/node3.html.

[25] Yi Guo, Jisheng Zhao, Vincent Cave, and Vivek Sarkar. Slaw: A scalable locality-aware adaptive work-stealing scheduler. In *2010 IEEE International Symposium on Parallel & Distributed Processing (IPDPS)*, pages 1–12. IEEE, 2010.

[26] Reazul Hoque, Thomas Herault, George Bosilca, and Jack Dongarra. Dynamic task discovery in parsec: A data-flow task-based runtime. In *Proceedings of the 8th Workshop on Latest Advances in Scalable Algorithms for Large-Scale Systems*, pages 1–8, 2017.

[27] Wei Wu, Aurelien Bouteiller, George Bosilca, Mathieu Faverge, and Jack Dongarra. Hierarchical dag scheduling for hybrid distributed systems. In *2015 IEEE International Parallel and Distributed Processing Symposium*, pages 156–165. IEEE, 2015.

[28] Michel Cosnard, Emmanuel Jeannot, and Tao Yang. Symbolic partitioning and scheduling of parameterized task graphs. In *Proceedings 1998 International Conference on Parallel and Distributed Systems (Cat. No. 98TB100250)*, pages 428–434. IEEE, 1998.

[29] Jeffrey Kodosky. Labview. *Proceedings of the ACM on Programming Languages*, 4(HOPL):1–54, 2020.

[30] Martin Richard Kong. *Enabling Task Parallelism on Hardware/Software Layers using the Polyhedral Model*. PhD thesis, The Ohio State University, 2016.

[31] Alina Sbirlea, Louis-Noel Pouchet, and Vivek Sarkar. Dfgr an intermediate graph representation for macro-dataflow programs. In *2014 Fourth Workshop on Data-Flow Execution Models for Extreme Scale Computing*, pages 38–45. IEEE, 2014.

[32] Paul Feautrier. Parametric integer programming. *RAIRO-Operations Research*, 22(3):243–268, 1988.

[33] Paul Feautrier. Some efficient solutions to the affine scheduling problem. i. one-dimensional time. *International journal of parallel programming*, 21(5):313–347, 1992.

[34] Uday Bondhugula, Albert Hartono, Jagannathan Ramanujam, and Ponnuswamy Sadayappan. A practical automatic polyhedral parallelizer and locality optimizer. In *Proceedings of the 29th ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 101–113, 2008.

[35] Somashekaracharya G Bhaskaracharya and Uday Bondhugula. Polyglot: a polyhedral loop transformation framework for a graphical dataflow language. In *International Conference on Compiler Construction*, pages 123–143. Springer, 2013.

[36] Ni labview compiler: under the hood. https://www.ni.com/en-us/support/documentation/supplemental/10/ni-labview-compiler--under-the-hood.html.

[37] Louis-Noel Pouchet, Peng Zhang, Ponnuswamy Sadayappan, and Jason Cong. Polyhedral-based data reuse optimization for configurable computing. In *Proceedings of the ACM/SIGDA international symposium on Field programmable gate arrays*, pages 29–38, 2013.

[38] Polyopt. http://web.cs.ucla.edu/~pouchet/software/polyopt/doc/htmltexinfo/index.html.

[39] Marek Palkowski and Wlodzimierz Bielecki. Traco parallelizing compiler. In *Soft Computing in Computer and Information Science*, pages 409–421. Springer, 2015.

[40] Marek Palkowski and Wlodzimierz Bielecki. Traco: source-to-source parallelizing compiler. *Computing and Informatics*, 35(6):1277–1306, 2016.

[41] Włodzimierz Bielecki and Marek Pałkowski. Tiling arbitrarily nested loops by means of the transitive closure of dependence graphs. *International Journal of Applied Mathematics and Computer Science*, 26(4):919–939, 2016.

[42] William Pugh and David Wonnacott. An exact method for analysis of value-based array data dependences. In *International Workshop on Languages and Compilers for Parallel Computing*, pages 546–566. Springer, 1993.

[43] Kathleen Knobe and Michael G Burke. The tuning language for concurrent collections. In *16th Workshop on Compilers for Parallel Computing*, 2012.

[44] François Irigoin and Rémi Triolet. Supernode partitioning. In *Proceedings of the 15th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 319–329, 1988.

[45] Guillaume Iooss, Sanjay Rajopadhye, Christophe Alias, and Yun Zou. *Mono-parametric tiling is a polyhedral transformation*. PhD thesis, INRIA Grenoble-Rhône-Alpes; CNRS, 2015.

[46] Jagannathan Ramanujam and Ponnuswamy Sadayappan. Tiling multidimensional iteration spaces for multicomputers. *Journal of Parallel and Distributed Computing*, 16(2):108–120, 1992.

[47] Michael E Wolf and Monica S Lam. A loop transformation theory and an algorithm to maximize parallelism. *IEEE Transactions on Parallel & Distributed Systems*, 2(04):452–471, 1991.

[48] Thomas H Cormen, Charles E Leiserson, Ronald L Rivest, and Clifford Stein. *Introduction to algorithms*. MIT press, 2022.

[49] Robert Endre Tarjan. *Data structures and network algorithms*. SIAM, 1983.

[50] Hash functions. https://www.cs.hmc.edu/~geoff/classes/hmc.cs070.200101/homework10/hashfuncs.html.

[51] The summit cluster. https://www.colorado.edu/rc/resources/summit.

[52] The summit cluster documentation. https://curc.readthedocs.io/en/latest/index.html.

[53] Intel xeon gold 6126. https://www.intel.com/content/www/us/en/products/sku/120483/intel-xeon-gold-6126-processor-19-25m-cache-2-60-ghz/specifications.html.

[54] Louis-Noël Pouchet et al. Polybench: The polyhedral benchmark suite. *URL: http://www.cs. ucla. edu/pouchet/software/polybench*, 437, 2012.

[55] Strong and weak scaling. https://www.kth.se/blogs/pdc/2018/11/scalability-strong-and-weak-scaling/.