RICE UNIVERSITY

High-level execution models for multicore architectures

by

Alina Sbîrlea

A THESIS SUBMITTED
IN PARTIAL FULFILLMENT OF THE
REQUIREMENTS FOR THE DEGREE

Doctor of Philosophy

APPROVED, THESIS COMMITTEE:

[illegible]

Vivek Sarkar
E.D. Butcher Chair in Engineering and
Professor of Computer Science

[illegible]

Keith D. Cooper
Doerr Chair in Computational Engineering
and Professor of Computer Science

[illegible]

Ray Simar
Professor in the Practice of Digital Signal
Processing Architecture Departments of
Electrical and Computer Engineering and
Computer Science

Houston, Texas

July, 2015

ABSTRACT

High-level execution models for multicore architectures

by

Alina Sbîrlea

Hardware design is evolving towards manycore processors that will be used in large clusters to achieve exascale computing, and at the rack level to achieve petascale computing, however, harnessing the full power of the architecture is a challenge that software must tackle to fully realize extreme-scale computing. This challenge is prompting the exploration of new approaches to programming and execution systems.

In this thesis, I argue that a two-level execution model is a relevant answer to the problem of extreme-scale computing. I propose an execution model that decomposes the specification of an application into two parts: defining at a high level *what* the application does, coupled with a low level implementation of *how* that's done. In my model, applications are designed as a set of sequential computational units whose connections are defined in a high- level, easy-to-write dataflow graph. The dataflow graph I propose — *DFGR* — specifies what the computation is rather than the implementation details, and is also designed for ease of programmability.

Second, I propose the use of work-stealing runtimes for coarse-grained dataflow parallelism and doacross runtimes for fine- grained dataflow parallelism, both of which can be expressed uniformly in DFGR. I justify this approach by demonstrating the performance of DFGR on two different runtime systems: Habanero C with coarse-grained task synchronization, and the new proposed OpenMP 4.1 specification with doacross synchronization.

Finally, I introduce a novel primitive for combining SPMD parallelism and task parallelism: the *elastic task*. Elastic tasks allow for internal SPMD parallelism within a computational unit. I provide a new scheduling algorithm for elastic tasks, prove strong theoretical guarantees in both work-sharing and work-stealing environments for this scheduling algorithm, and demonstrate that it also offers performance benefits in practice due to locality and runtime adaptability.

Acknowledgments

I would like to thank first of all my advisor, Professor Vivek Sarkar for his guidance throughout the Ph.D. years, for allowing me the freedom to pursue my ideas and letting me discover my way yet always being there to offer advice, feedback and suggestions for future directions and new research opportunities.

Thank you to my committee members Professor Keith Cooper and Dr. Ray Simar for their feedback on the proposed work, for their encouragements throughout the Ph.D. program and the insightful discussions.

A special thank you to my collaborators: Kunal Agrawal, Zoran Budimlic, Sanjay Chatterjee, Jason Cong, Max Grossman, Louis-Noel Pouchet, Dragos Sbirlea, Jun Shirako, Kyle Wheeler, Yi Zou. It has been a please and an honor to work with you and I am looking forward to the chance of future collaborations.

To the members of the Habanero team, thank you for being an awesome group, for offering the chance to discuss both research problem and getting together for a wide variety of enjoyable activities. I am happy to be able to call you my friends.

Last but not least, I would like to thank my parents and especially my husband Dragos for offering their support in the most difficult moments, for the kind words, encouragements and their love.

Thank you!

Contents

Illustrations

Chapter 1

Introduction

It is now well understood that extreme-scale computing will be faced by key software challenges, including those related to concurrency, energy, resilience [2] and programmability. These challenges are prompting the exploration of new approaches to programming and execution systems. The approaches being explored in the context of extreme scale computing are many-fold and range from dedicated DSLs (domain specific languages), to SPMD parallelism, to general task parallelism.

Domain specific languages aim to give a high-level view of applications and to ease programmability but are generally restricted to particular sets of problems, such as stencil computations [3] or graph processing problems [4].

SPMD parallelism [5] is another paradigm being explored in particular for data parallel applications which can take advantage of emerging heterogeneous architectures and use the full capacities of general purpose graphics processors. It is particularly important for its tight coupling of threads executing the same task, which often leads to great performance due to good memory reuse and efficient synchronization, but may also suffer when branching occurs in the codebase, leading to wasted cycles as the workers are forced to execute in lock-step.

For multicore machines, *task parallelism* has emerged as a dominant paradigm for parallel programming. Many programming languages and libraries such as Cilk [6], Cilk Plus [7], Intel Threading Building Blocks [8], .Net Task Parallel Library [9], Habanero-Java [10], and Habanero C [11] support this programming paradigm. These regularly rely on a runtime scheduler for mapping tasks onto available processors to achieve load balancing and performance.

The architectural roadmap has lead to also re-visiting the dataflow model to find new ways to address the challenges of extreme-scale software. The *macro-dataflow execution model* [12] has begun to gain users through the benefits it offers, which include simplified programmability, intrinsic parallelism, scalability and support for heterogeneous architectures, in particular when being used with a data-driven runtime [13, 14, 15]. In contrast with DSLs, the macro data-flow model can be designed to offer high programmability while preserving generality, and thus not restricting users to particular sets of problems. As a result, a wide variety of programming systems have begun exploring the adoption of dataflow principles, ranging from new programming models such as Concurrent Collections (CnC) [16, 17] to the new task dependency feature in OpenMP 4.0 [18].

In this thesis I show that a two-level execution model can help address the challenges of extreme-scale computing, using a high-level representation for programmability and a low level runtime for performance, portability and theoretical performance guarantees.

The rest of the document is organized as follows. Chapter 3 introduces DFGR (Data-Flow Graph Representation), the macro data-flow programming model I propose and its language specification. Chapter 4 describes its implementation, the translator which enables the use of multiple underlying runtimes and the performance evaluation of my system. Chapter 5 introduces the new elastic task primitive, its theoretical guarantees in a work-sharing and work-stealing environment, the ElastiJ runtime system that implements the elastic primitive and its evaluation. Finally Chapter 6 describes in detail the related work.

1.1 Thesis Statement

My thesis is that a high-level execution model coupled with efficient compiler and run-time implementations can help address the programmability and performance challenges of current and future hardware with strong theoretical guarantees.

Chapter 2

Background

2.1 The Concurrent Collections (CnC) programming model

2.1.1 CnC as a high-level programming abstraction

The Concurrent Collections (CnC) model was originally defined by Intel [16] and further implementations of the model were created as part of the Habanero Research Group at Rice University [17]. The model aims to ease programming for domain experts by defining applications as a set of sequential pieces of computation with implicit dependences between them based on data accesses.

CnC does not define *when* or *where* a computation takes place, but relies on an underlying runtime to ensure that all data read by that computation is available before the computation takes place. A key feature that ensures there are no data-races and that dependences are implicitly determined, is that all data is *dynamic single assignment*. This means that throughout the whole execution of the program each piece of data is only written once.

All implementations of the model offer a separation of concerns: defining the high-level view of the application vs tuning the application. In Intel's approach the structure of the application needs to be written in an API format, using C++, while for tuning a specific *Tuner* interface needs to be implemented and coupled with the original implementation. In the Rice implementations there is a separate textual specification of the structure that is language independent, coupled with a translator that can generate parallel code in different languages (e.g. Java, C). For tuning, a computation can be described to have affinities with different platforms, which translates to placement of that computation to the target architecture. This view of separating the application description and its implementation details makes CnC a great alternative for domain experts.

Our experience with using it in the NSF Expeditions Center for Domain Specific Computing (CDSC) [19, 20, 21] has shown that in practice domain expert find it easy to program with very good outcome in performance [22].

2.1.2 The structure of the CnC model

Programs defined using the CnC model are described using three constructs: steps, items and control tags, and for each such construct there is a collection concept that refers to a group of steps, items or control tags with similar properties.

Step is the name attributed to a sequential piece of computation that receives a set of inputs and outputs a set of results, but has no side effects other than these results. The outputs produced by a step can be items or control tags. Steps are grouped in collections based on their functionality such that steps executing the same computation (for different inputs) will be grouped into the same collection. Identifying a step in a collection is done using a unique tag.

Item refers to a piece of data, with any type, that can be written by a step a single time due to the dynamic single assignment property. Generally, items of the same type that are produced by steps in the same collection are grouped into a single item collection. Just like with steps, items are uniquely identified in a collection using a tag.

The mechanism for creating new steps is by writing a control tag into a control collection. The control tag can be viewed as an item whose value is a step's unique identifier. A control tag can be associated with multiple kinds of steps, which means that writing a single tag with value τ can start steps $S1(\tau)$, $S2(\tau)$, etc.

When creating an application, we say that steps *get* items they read and *put* items and control tags they write. In our previous work [21, 23] we also defined the relations between the step tag and its inputs and outputs as *tag functions*.

In this thesis we propose a high-level programming model, DFGR, that builds on the CnC model. We detail the differences and similarities in Section 3.2.2.

2.1.3 Heterogeneous CnC

In our previous work [21, 23] we explored mapping CnC onto heterogeneous platforms in order to achieve high performance and low energy consumption while preserving the ease of use of data-flow programming. We then designed a software flow for converting high-level CnC programs to the Habanero-C language. We also extended the Habanero-C runtime system to support work-stealing across heterogeneous computing devices and introduce task affinity for these heterogeneous components to allow users to fine tune the runtime scheduling decisions. We demonstrated a working example that maps a pipeline of medical image-processing algorithms onto a prototype heterogeneous platform that includes CPUs, GPUs and FPGAs. For the medical imaging domain, where obtaining fast and accurate results is a critical step in diagnosis and treatment of patients, we showed that our model offers up to 17.72× speedup and an estimated usage of 0.52× of the power used by CPUs alone, when using accelerators (GPUs and FPGAs) and CPUs.

2.2 Parallel runtimes for shared-memory systems

2.2.1 Habanero Programming Model

The Habanero programming model is a task based parallel programming model with C [24, 25] and Java [1] implementations developed at Rice University. The two main features of the Habanero model that are relevant to this thesis are the *async* and *finish* constructs, which define lightweight dynamic task creation and termination and were originally defined in X10 [26].

A new child task is created using statement "**async** ⟨*stmt*⟩". This will run ⟨*stmt*⟩ *asynchronously* (i.e. before, after, or in parallel) with the remainder of the parent task. Figure 2.1 provides a diagram in which the parent task, T_0, uses an `async` construct to create a child task T_1. Thus, `STMT1` in task T_1 can potentially execute in parallel with `STMT2` in task T_0.

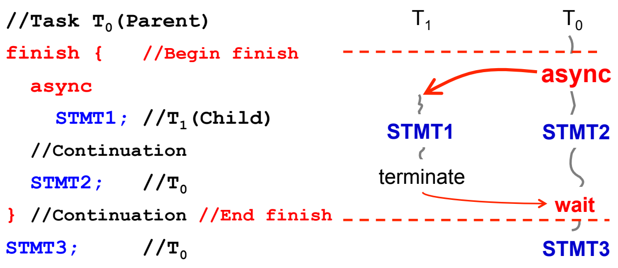


Figure 2.1 : An example code schema with `async` and `finish` [1]

`async` is a powerful primitive because it can be used to enable any statement to execute as a parallel task, including statement blocks, for-loop iterations, and function calls. In this work we use the `async` statement to create dynamic instances of DFGR steps.

`finish` is a generalized join operation. The statement "**finish** ⟨*stmt*⟩" causes the parent task to execute ⟨*stmt*⟩ and then wait until all transitively spawned tasks using the `async` primitive have completed.

Each dynamic instance T_A of an `async` task has a unique *Immediate Enclosing Finish* (IEF) instance F of a `finish` statement during program execution, where F is the innermost `finish` containing T_A [27]. There is an implicit `finish` scope surrounding the body of `main()`, so program execution will only end after all `async` tasks have completed.

For example, the `finish` statement in Figure 2.1 is used by task T_0 to ensure that child task T_1 has completed executing `STMT1` before T_0 executes `STMT3`. If T_1 created a child `async` task, T_2 (a "grandchild" of T_0), T_0 will wait for both T_1 and T_2 to complete in the `finish` scope before executing `STMT3` [21].

In this thesis, we use the Habanero-C language as the foundation for the low-level implementation of our high-level model — DFGR. We also use Habanero-Java library(HJlib) as the foundation for the low-level implementation of elastic tasks, a new construct that unifies task parallelism and SPMD parallelism.

2.2.2 OpenMP Programing Model

The OpenMP programming model [28], has gained users due to its simplicity for porting existing C programs to multicore as well as the performance benefits it offered. Though the model was originally limited to parallel loops, it has since evolved due to the need to express a wider range of applications and offer a broader hardware support.

A major addition to the OpenMP was the introduction of tasks, thus allowing programmers more expressiveness with respect to what can run in parallel. OpenMP's latest specification (OpenMP 4.0) [18], offers a series of additional new features. One extension is extending the OpenMP tasks to allow grouping in order to ease synchronization primitives. In addition, task-to-task synchronization is supported by a means to specify task dependencies. The latter was added based on the research results proposing the DoAcross model with loop dependences [29].

In our work with use the OpenMP 4.0 along with the DoAcross with explicit task dependences extension as one of the underlying runtimes for our high-level programming model DFGR. In Section 4.3 we detail the tool we created to port a DFGR representation to a polyhedral compiler and how we use it to generate the OpenMP code.

Chapter 3

Data-Flow Graph Representation

3.1 Motivation

We are in an era when hardware design has evolved into developing heterogeneous, massively parallel multicore architectures, which are used in large clusters to achieve exascale computing. The challenge faced by the software is how to map applications in order to harness the full power of such architectures. This challenges is prompting the exploration of new approaches to programming and execution systems, and, specifically, re-visiting of the dataflow model to find new ways to address the challenges of extreme-scale software. In the early days of dataflow computing, there was a belief that new programming languages such as VAL [30], Sisal [31], and Id [32] were necessary to obtain the benefits of dataflow execution. However, there is now an increased realization that "macro-dataflow" execution models [12] can be supported on standard multi-core processors by using data-driven runtime systems [13, 14, 15]. There are many benefits that follow from macro-dataflow approaches, including simplified programmability, increased asynchrony, support for heterogeneous parallelism, and scalable approaches to resilience. As a result, a wide variety of programming systems have begun exploring the adoption of dataflow principles [16, 17, 18] and there is now a growing need for compiler and runtime components to support macro-dataflow execution in these new programming systems.

In this thesis we argue that a two-level programming model is a relevant answer to this problem, where the application is first decomposed into (atomic) steps, and the coordination of those steps (which includes data movement, dependence synchronization and creation of new steps) is orchestrated through an easy-to-write and easy-to-generate dataflow graph. To that end, we introduce an intermediate graph representation for macro-dataflow programs called DFGR (Data-Flow Graph Representation), describe its specification and implementation details, and show that DFGR can be used by a wide range of programming systems.

We describe how the DFGR model can be used as an abstraction to map applications for extreme scale systems, while remaining runtime independent. This allows such applications to run anywhere the underlying data-driven runtime systems can run, such as heterogeneous architectures including GPUs and FPGAs or distributed-memory clusters and data centers. DFGR can also help improve productivity by focusing the user on easily expressing task and data parallelism.

The research contributions in this chapter have been published in [33, 34]. The contributions of this chapter are the following.

- We introduce DFGR, a dataflow graph representation language to model full applications and ease their deployment on homogeneous/heterogeneous parallel architectures.

- We create a fully automatic translation system from DFGR to Habanero-C, enabling DFGR graphs to be automatically executed anywhere the HC runtime operates, which includes shared-memory multi-core and heterogeneous nodes containing GPUs and FPGAs.

- We present experimental results comparing a production implementation of the macro-dataflow model (Intel CnC) with DFGR on a shared memory system

- We create a translation of the DFGR model to ScopLib, a popular format used by many polyhedral compilers.

- We demonstrate how the same high-level DFGR representation can be used with an OpenMP backend and be tuned to take advantage of polyhedral optimizations such as tiling, skewing, and `doacross` parallelism with `depends` clauses.

3.2 The DFGR model

3.2.1 Macro-dataflow for high-performance computing

A major objective of the Center for Domain-Specific Computing [19, 20] is to ease the development and deployment of applications on heterogeneous devices. To achieve this goal, we implement a two-level programming principle. Given a complete application, for instance, an end-to-end medical imaging pipeline for CT scan reconstruction and analysis [20, 21], the application is first decomposed into steps, which can be executed non-preemptively on a given device. A key feature to enable both high-performance and programmability on a variety of devices is to allow the computation steps to be implemented in any language of choice, such as C/C++, CUDA, HC, etc. These step implementations form the low-level of the programming model. At the high-level, these steps may be called numerous times on different input data, and are seen as black boxes reading some data elements and producing some other data elements.

The coordination (control and data flow) between these steps is described through a DataFlow Graph Representation (DFGR). This graph is then automatically compiled into a parallel implementation, including the generation of all communications needed between step instances and all the data- and control-flow dependences between them, as described in the DFGR file. We can exploit a dynamic and heterogeneous run-time system for parallel step execution [21] that can be also ported to distributed computing [35, 13], by compiling a DFGR graph to a parallel runtime with good interoperability with low level languages, such as the interoperability of the Habanero-C runtime with the C, CUDA and OpenCL languages. In our framework, *a single DFGR file is used* to model the full application data and control flow between these atomic steps, irrespective of the target platform used to execute the computation. Indeed, a key motivation for DFGR and its associated execution model is to decouple the task of *expressing* the parallelism, which should belong to the domain experts and algorithm designers, from the task of *implementing* the available parallelism on a given hardware. DFGR is meant to ease the expression of data flowing in and out of step instances by letting the user or higher-level tool chain focus exclusively on modeling the application data and control characteristics without any concern about their actual implementation. But for this approach to succeed, it is required to ensure a set of properties of the language to allow for (1) easy debugging and analysis of the graph structure; (2) easy deployment on a variety of hardware, ranging from heterogeneous multi-core/GPUs systems to distributed computing on cluster; and (3) easy modeling of complex applications, including at different parallelism grains.

To address these three problems, we present DFGR, a macro-dataflow graph representation that builds on past work on Intel Concurrent Collections [16] and CnC-HC projects [21]. In a nutshell, DFGR is a macro-dataflow language that supports dynamic task parallelism while enforcing dynamic single assignment for data, thereby greatly simplifying debugging and analysis. DFGR is fully and automatically translatable to the parallel language chosen for the concrete implementation of the model. To demonstrate this, we implemented a compiler from DFGR to the Habanero-C (HC) task-parallel language, which allows the graph to execute on any hardware implementing the HC runtime or equivalent [21, 16, 13], as well as a compiler from DFGR to OpenMP with the recently proposed DoAcross extension in OpenMP 4.1. Finally DFGR uses simple concepts such as unique tags to model the relationship between dynamic instances of steps and the particular data elements they access in a simple text representation, thereby greatly facilitating the description of the application's parallelism and data flow.

3.2.2 Comparison with previous data-flow models

DFGR is meant to serve as a general representation which can rely on a parallel and distributed runtime for performance. Previous models such as Kahn Process Networks (KPN) [36] or the Synchronous Dataflow Model (SDF) [37] target streaming applications, while Concurrent Collections (CnC) primarily targets task parallelism. DFGR can express both the streaming model and task parallelism and it can be further optimized for both through analysis of the graph structure.

DFGR simplifies the CnC model by allowing dependencies to be expressed between steps and items, and also directly between steps and steps. It enhances CnC with a means of expressing precisely what items are read and written by each step. This is achieved through tag functions [21] (the tag identifying an item read by a step is a function of the step's tag) and regions modeling sets of tags. It also preserves the concept of affinities [21] between steps and underlying available hardware, allowing automatic tuning of the application, when multiple code variants for accelerators are made available.

A hierarchy of concepts for modeling sets of tags is provided, from simple ranges (e.g., rectangles) to affine integer tuple sets (e.g., polyhedra) to union of integer sets and finally arbitrary sets. This clear hierarchy allows for static analysis and optimization of the graph using compiler frameworks such as the polyhedral model, when the tag sets and relations between them are affine forms. These concepts are novel and key to the DFGR model; they are further detailed in Section 3.3.3.

While in the streaming model instances of (e.g, calls to) steps run in sequence, in DFGR any instances of steps with no dependence between them are viewed as parallel tasks. Steps in the same collection may run on different cores or even different devices. With tools capable of discovering part of the graph topology statically, DFGR can make use of static scheduling, while the remaining graph topology will be discovered at runtime and it will rely on dynamic scheduling. CnC relies entirely on dynamic scheduling while models such as KPN and SDF expect the full graph topology to be known a priori. DFGR natively encompasses both.

Steps in DFGR are logically functional (e.g., stateless), and inputs and outputs of steps are stored separately from the steps' internal data structures . All resources internal to the step are cleared when the step instance finishes. Data read by a step may come from any source, but any $\langle$key, value$\rangle$ pair is only written once. DFGR items are Dynamic Single Assignment (DSA) and are never collected, i.e. they are persistent. Optimizations such as folding the data space when an ordering is imposed or setting a fixed number of reads per item (“get_counts") can allow items to be collected for better storage management [38]. DFGR also permits arbitrary access to data outside of the graph: such global data can be used to create a state without using an item collection. However using global data limits the outcome of graph analysis and transformations, since this non-graph data is not represented nor analyzed in the macro-dataflow graph.

The creation of steps is achieved directly by one step spawning another step. This is different from CnC, where control (tag) collections are used for spawning steps, but also from streaming models where steps start as soon as data is available without the ability to choose which steps should run. Steps are allowed to make conditional Puts/Prescribes for expressing general applications, while no switch and select nodes are present in SDF. In DFGR, steps may also pass parameters to the steps they spawn.

A step can explicitly request to wait on another step in DFGR, while synchronization in previous models (including CnC) required indirect coordination via items. Inter-step synchronization in DFGR can be used to coordinate/schedule accesses to data that is not modeled in the DFGR graph through item collections. DFGR also allows non-determinism through constructs such as “PutIfAbsent", a method which only writes an item if it was not previously written. Graph algorithms can use such constructs in order to establish the first visitor for a node.

Finally, a DFGR graph is *fully executable* without having to execute any of its actual step implementation. That is, the entire dynamic parallelism and data flow can be uncovered using a simple interpretation of the DFGR program, allowing for subsequent analysis and optimizations such as race detection and storage optimization using techniques such as inspector-executor frameworks [39].

3.3 DFGR language specification

3.3.1 Core features for macro-dataflow modeling

DFGR is a graph representation that contains two main components: *steps*, that represent pieces of computation; and *items*, that represent pieces of data read and written by steps. The user describes an application by writing a graph (in textual form or using an API to create the graph) that captures the relation between data items and steps. In order to model *explicitly* all the dynamic instances of each step as well as all items during the execution of the application modeled, both steps and items are grouped into collections within which they have unique identifiers called *tags*. In order to guarantee the graph is deterministic and free of data races, all data in item collections must follow the dynamic single assignment rule, that is an item in a collection is never written more than once.

An item collection is a group of data items having the same type. Each item in the collection can be uniquely identified by its tag, thus an item collection is a set of (tag, value) pairs. Items can be written to a collection by the environment and also by other steps. Similarly, items can be read by steps and by the environment once the graph execution has finished. Item collections are declared in the textual representation using brackets: `[int* A]` declares a collection of items which are pointers to integers. Using a pass-by-value mechanism, any type, including structures and arrays, can be used for items.

The human-friendly modeling of all data elements being read and/or written by a step instance is achieved by relating the tags of item collections with tags associated to step instances. For instance `[A : i]` models tag `i` of collection `A`. Then `[A : i-1] -> (S : i) -> [A : i]` models the fact that instance `i` of `S` will read element `i-1` of collection `A`, and produce element `i`. In DFGR there are multiple ways to describe tags, as discussed in Sec. 3.3.3. In its most general form the user can write `[A : foo(...)]` to describe a tag value, where `foo` is a call to some pure function possibly requiring run-time evaluation to compute its value.

A step collection is a group of instances of the same step function. The unique identifier (tag) of a step instance can carry semantics used by the step implementation itself, analogous to the “this" pointer in object-oriented programming. Steps can be started by the environment which is in charge of initializing and starting the graph, and also by other steps. Depending on the model’s implementation, it can adhere to the strict preconditions model, where steps will not execute until all its input data is made available; steps can execute eagerly and block or rollback when data is not available; or have a flexible approach through the flexible preconditions model [40]. Steps are written using parentheses: `(S)`. DFGR uses arrows to express reads and writes: `[A]->(S)` and double colon to express the creation of new steps: `(S1)::(S2)`. When using tags, the notation `(S : i)` models instance `i` of step `S`, and `env::(S : {1..42})` models that the environment `env` will create

42 instances of `S` at the start of program execution, with tags ranging from 1 to 42. The modeling of data and control dependences between step instances is achieved through the modeling of the data read/written by a step instance, and also optionally using an explicit step-to-step (e.g., point-to-point) synchronization construct. For instance `(S1 : i) -> (S2 : i)` models the fact that instance `i` of step `S2` will not start until instance `i` of step `S1` completes.

3.3.2 Example: Smith-Waterman in DFGR

In this section we use the Smith-Waterman sequence alignment algorithm to illustrate the steps needed to write an application in DFGR. The DFGR representation can originate from hand-written user code, from tools analyzing dependences in sequential programs or from other graph representations.

Writing a DFGR representation implies that the user must reason about the computation that exists within the graph, the data read and written and how this information flows from one step to another. In Figure 3.1, we give a visual representation of the computation performed on a matrix in the Smith-Waterman algorithm. We identify 4 kind of steps: a single *corner* step (C) computing the top-left matrix corner, and a set of steps computing the *top* row (T), *left* column (L) and the *main* center (M) elements of the matrix. The arrows mark the flow of data, e.g. the information from step (SC is read by three other steps (T),(L) and (M), while each step (T) provides input to another instance of step (T) and 2 instances of step (M). In this example it becomes clear the need to group steps into collections and use unique identifiers to differentiate between instances of the same step. Let us assume that we are using a NH $\times$ NW matrix. Then, there are (NH-1) $\times$ (NW-1) center steps, where each can be identified by a unique tag (i,j), with 1$\leq$i$\leq$NH and 1$\leq$j$\leq$NW. From Figure 3.1 we can also infer data dependences, e.g., all center steps read 3 items and write a single item. Using the tuple (i,j) as the unique tag identifer, we can say that each step (C:i,j) reads items [A:i-1,j-1], [A:i-1,j], [A:i,j-1] and writes [A:i,j].

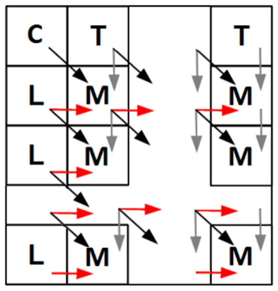


Figure 3.1 : Smith-Waterman: The computation steps are grouped in a matrix structure based on their unique identifiers (i,j) and the items they write [A:i,j]. Arrows show data dependences for each step.

Alternatively, a graph representation can originate from automatic analysis of a sequential code such as in Listing 3.1. In this code snippet we abstracted the actual computation performed by each step with a function call. Note that from this code we can also infer the dependences specified before, in particular what items each step reads and writes and a unique identifier for each step. As it is required to use the dynamic single assignment form for DFGR, if the input code is not in DSA form already, a promotion to DSA must be performed during the translation to DFGR.

Listing 3.1: Sequential Smith-Waterman code.

```
A[0][0] = corner();
for(j=1; j<NW; j++)
   A[0][j] = top(j);
for(i=1; i<NH; i++) {
   A[i][0] = left(i);
   for(j=1; j<NW; j++)
      A[i][j] = main_center(i, j, A[i-1][j-1],
                     A[i-1][j],A[i][j-1];
}
```

The DFGR file for Smith-Waterman is shown in Listing 3.2. The first line of code declares an item collection, where each item is of type int. The next four lines of code specify for each of the 4 steps what items are read and written, using the unique tags for both steps and items. The final four lines specify what the environment needs to produce for the graph to start, and what it needs to emit after completion of the graph (output data). The environment will start all computation steps and it will read one item resulting from the computation (the bottom right corner, the sequence alignment cost in Smith-Waterman).

Listing 3.2: DFGR for Smith-Waterman.

```
[int A];
      //Steps' I/O relations
(corner:i,j) -> [A:i,j];
(top:i,j) -> [A:i,j];
(left:i,j) -> [A:i,j];
[A:i-1,j-1], [A:i-1,j], [A:i,j-1] ->
        -> (main_center:i,j) -> [A:i,j];
     //Steps started by the environment.
env::(corner:0,0);
env::(top:0,{1 .. NW+1});
env::(left:{1..NH+1},0);
env::(main_center:{1..NH+1},{1..NW+1});
[A:NH,NW] -> env;
```

3.3.3 Expressiveness in identifying dependencies

Tag functions explained

We say that I/O and prescription rules are composed of two parts connected by the $->$ and :: operators respectively. One side is the step driver, declared between parentheses and identified by a multi-dimensional tag, abstracted by user-chosen variable names. The other side (left or right for I/O relations and right for prescriptions) is a list of items or steps each identified by a list of functions, where each such function is a function of the driver step's tag components. The diagram in Figure 3.2 gives an example of a step I/O relation and aims to clarify what are tags, tag components and tag functions.

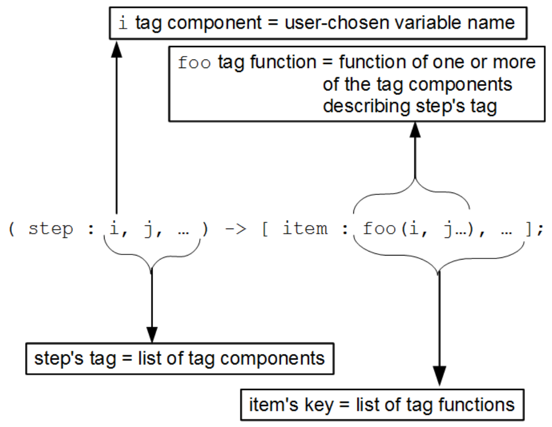

i tag component = user-chosen variable name
foo tag function = function of one or more of the tag components describing step's tag
(step : i, j, …) -> [item : foo(i, j…), …];
step's tag = list of tag components
item's key = list of tag functions

Figure 3.2 : Step tags, tag components and tag functions explained

A hierarchy of concepts for tag sets

A key feature of DFGR is to attempt to reconcile static and dynamic macro-dataflow modeling, especially by allowing the explicit and unique naming of each step instance and each item in a data collection. To achieve this goal we propose a hierarchy of concepts that can be used to represent sets of tags, ranging from the simplest form of integer ranges to the most complex form of arbitrary sets. For instance the example in Listing 3.2 uses *ranges* to describe the step instances prescribed by the environment, i.e., `{1..NW+1}`. But in the general case, more complicated shapes can be required to properly model the set of steps. On the other hand, allowing for abitrary sets without any property allow to model the general case, but may dramatically limit the static analysis that can be performed.

The motivation for introducing several different concepts in a structured way relates to the different kinds of static analysis frameworks that could be used at the graph level. Limiting to particular constructs to model tag sets will automatically enable or disable certain static analysis frameworks. Focusing only on using integer tuples for the tags, for good expressiveness we need to capture the sets of tags to be used in the representation, be it for describing a set of step instances, a set of prescribed steps, or a set of data being read/written by a step. We propose the following hierarchy for these sets: *(1) ranges; (2) simple polyhedron; (3) union of $\mathcal{Z}$-polyhedra; (4) union of arbitrary sets*.

In a nutshell, ranges will model rectangles and are well suited for simple, regular computations. Polyhedra, which are described using affine inequalities, enable powerful static analysis and transformation of the graph based on the affine framework [41, 42]. Unions of $\mathcal{Z}$-polyhedra are a generalization of polyhedra, allowing for more complex geometric shapes to be modeled by using union of convex sets, and regular strides in the set are supported using affine lattices [43]. Modern polyhedral compilation frameworks fully support the complete analysis of programs described using such polyhedra [44, 45, 46]. Finally, arbitrary sets are sets not belonging to any of the previous category, typically used when the set cannot be described using strides and affine inequalities; or when the set is described using functions whose result is not statically known, for example `foo(i,j)` where `foo` is seen as an uninterpreted function at the graph level. In our hierarchy, the less expressive the set type is, the simpler it is to implement static analyses and code generation to a parallel language on such sets.

Ranges and simple polyhedra The role of tag sets is analoguous to the role of iteration spaces for step prescription, and data sets for items. Tag sets can be used to express complex I/O and prescription dependences in tag functions.

In the code snippet below we use a 1-dimensional range to define that step $s1$ may write

$i1$ items into $item1$ with tags ranging from 0 to $i1$.

```
(s1 : i1) -> [item1 : {0 .. i1}];
```

A key benefit of ranges is that they are very simple to analyze and translate. For instance one can scan a set described by a range using a simple `for` loop, using the bounds of the range as the loop bounds. Multidimensional ranges are simply scanned using nested for loops.

The same set can also be defined using a *region*, the generic name in DFGR for tag sets which are not ranges. Polyhedra, union of $\mathcal{Z}$-polyhedra and arbitrary sets are all considered regions in DFGR. A basic syntactic analysis of the region expression is enough to determine which actual type of set is being implemented. Here is an example of defining and using a region $reg1$:

```
<reg1(ub) : r1> { 0 <= r1, r1 <= ub };
(s1 : i1) -> [item1 : r1; reg1(i1)];
```

The first line defines a region named `reg1`, which is a 1-dimensional set. A parameter (that is, an unknown constant) is used in the region definition: `ub`. `r1` is the variable associated to this set, that is, `r1` will take all values defined by the set `reg1`. The region is defined here using a conjunction of affine inequalities each separated by `,` so it is a basic polyhedron. The second line is an I/O relation which uses the region $reg1$. It says that a step instance – identified by $i1$ – from step collection $s1$, will write all items with tag $r1 \in reg1$. We can combine tag functions with regions, such as in:

```
(s1 : i1) -> [item1 : f1(r1); reg1(i1)];
```

which models that instance $i1$ of $s1$ may write into item collection $item1$, for all $r1 \in reg1$, any or all values with tags given by the tag function $f1(r1)$.

Ranges may be used to describe *unnamed, rectangular, contiguous* and *multidimensional* iteration spaces.

```
(s1:i,j)->[item1:{i-1..i+1},{i+j..i*j+1}];
```

In the example above, there are two ranges, one for each of the tag components of the item collection, we also say it defines a 2-dimensional range. First, note that ranges are *unnnamed*. For example, the first region could be translated to "$for(x = i-1; x < i+1; x++)$..." but there is no variable x to name the iteration space.

Secondly, the sets defined by ranges are *rectangular*, because without names each dimension is independent of the others (i.e. the second range cannot refer to an particular item in the first range). As an analogy with for loops, ranges cannot express a triangular loop nest. Thirdly, the ranges can only be used at the outer level when specifying a tag function. To be more precise, it is correct to say "$\{i+1..i+5\}$" but incorrect to say "$i+\{1..5\}$". In this example the two notations are equivalent, because addition (and subtraction) is distributive with a range. However multiplication is not distributive, nor is division. As such, there is no means of specifying that a step writes for example items $i*2, i*3, i*4...i*N$ using a range, but we can easily write this with a region. We say thus that a range is a *contiguous* set of integer points, whereas a region need not be.

Finally, ranges can be *multidimensional*, but only by using a range for each tag component. Thus, a tag component can be only one range; regions will introduce as many dimensions as the region specifies, but the additional dimensions will apply for all tag components, only once. To formalize this, let us name each interval defined using a range as I_index. In the previous example $I\_1 = \{i-1..i+1\}$ and $I\_2 = \{i+j..i*j+1\}$. Using ranges, the total iteration space is $I\_1 \times I\_2$. Let us now assume we have a similar code, which also includes a region:

```
<reg : r> { 1 <= r < 3; };
(s1:i,j)->[item1:{i-1..i+r},{i+j..i*j+r};reg];
```

We name the region interval $\mathcal{R} = \{1..3\}$, and the range intervals $I\_1 = \{i-1..i+r\}$ and

$I\_2 = \{i+j..i*j+r\}$, where $r \in \mathcal{R}$. In this example, the iteration space is $\mathcal{R} \times I\_1 \times I\_2$.

Clearly, we can add additional dimensions using regions, which makes it a great feature. But these will always refer to all tag components. So, the question that arises is: can we express data dependences with ranges that cannot be expressed with regions alone? The answer is yes.

Let us look again at the example we gave for ranges:

```
(s1:i,j)->[item1:{i-1..i+1},{i+j..i*j+1}];
```

If we modify the first range to factor out the *i*, we can use a region *reg*, and obtain the following modified code:

```
<reg : r> { -1 <= r <= 1; };
(s1:i,j)->[item1:i+r,{i+j..i*j+1};reg];
```

However for the second range, there is no means of extracting a region that is independent of i and j, in order to eliminate the range. There is still a means to obtain the same expressiveness with a region, by defining *parametric regions*:

```
<reg2(i,j):r1,r2>{i-1<=r1,r1<=i+1,i+j<=r2,r2<=i*j+1};
```

More region constructs In practice, ranges and simple polyhedra are often enough to express the tag sets needed to model an application. They also come with the benefit of easy compilation to a loop-based language, as scanning such sets is straightforward for ranges (see above) but also for simple polyhedra after normalizing the inequalities in row-echelon form. On the other hand, more complicated patterns may need to be expressed. In our taxonomy for tag sets, we purposely isolated union of $\mathcal{Z}$-polyhedra from arbitrary sets because, similarly to ranges and simple polyhedra, *there exists readily available tools to generate code scanning those sets at compile-time*. For instance CLooG [47] and ISL

[46] both generate automatically code scanning those sets using only `for` loops and `if` conditionals, making the translation of these sets to the target parallel language straightforward. We remark that numerous static analysis (dataflow analysis, scheduling, etc.) are also available for programs described using only this type of set.

In its general form, a region is a union (e.g., disjunction) of conjunction of inequalities. Its inequalities may be a function of parameters and uninterpreted functions. For instance the region below:

```
<reg2(i,j):r1,r2>{i-1<=r1,r1<=i+1,i+j<=r2,r2<=i*j+1};
```

is a conjunction of four inequalities, i and j are parameters, and $r1$ and $r2$ are the two dimensions of this set, e.g., points in this set are tuples $(r1,r2)$. As i and j are parameters, this set is actually a simple polyhedron. The region below:

```
<reg3(i,j):x,y>{i<=x,x<=j,y>=0,y<=fc(x,j)},{x=1,y=3};
```

represents a union of sets (separated by `,`), where in the first part an uninterpreted function `fc(x,j)` is used. Without further information on this function, this set cannot be analyzed statically using polyhedral frameworks, thereby limiting the analysis that can be performed on the whole graph. On the other hand, it allows the modeling of arbitrary sets, but at the expense of a possibly costly run-time scanning code as the function needs to be evaluated for each x value.

3.4 Conclusions

We conclude this chapter with the contributions that we presented. We proposed DFGR, a graph representation for macro-dataflow programs offering high programmability for exascale computing. We outlined the language features that enable DFGR to model complex applications and the potential for graph analysis for DFGR applications. In the next chapter we present our implementation of the DFGR model and how it can map on two different runtimes to obtain good performance.

Chapter 4

Mapping of the DFGR model

In this chapter we present the implementation of the DFGR model and show how the same high-level representation can be mapped onto two different underlying runtimes. We demonstrate the flexibility of the model proposed and its performance benefits for both runtimes used.

4.1 Mapping DFGR to the Habanero C programming model

4.1.1 Language tool chain

The high-level DFGR model is designed to be language independent, and the concrete implementations of the model will make the language choice. We created an implementation of DFGR that relies on a parallel C programming model, in order to achieve both performance and interoperability with languages dedicated to specialized hardware.

Figure 4.1 presents the implementation flow for an application written in DFGR.

The steps for creating an application using DFGR are the following. First, an application needs to be written in the DFGR format, i.e. the algorithms needs to be decomposed into steps, along with the producer-consumer relationships between them. This textual graph can be either written by the user, or it can be generated by tools analyzing and translating from another programming language, or from other dataflow modeling using a similar graph representation. An example of the latter is generating DFGR from CnC, but note that this translation cannot use all the features in DFGR, as CnC's specification is

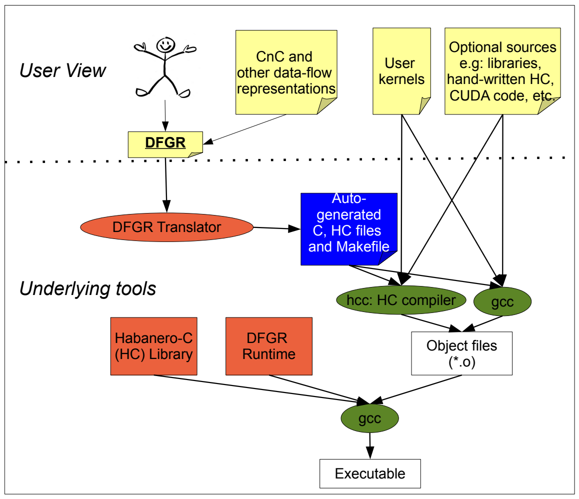

User View
CnC and other data-flow representations
User kernels
Optional sources e.g: libraries, hand-written HC, CUDA code, etc.
DFGR
DFGR Translator
Auto-generated C, HC files and Makefile
Underlying tools
hcc: HC compiler
gcc
Habanero-C (HC) Library
DFGR Runtime
Object files (*.o)
gcc
Executable

Figure 4.1 : DFGR Implementation Flow

more restrictive.

Further, we provide a translator that generates a series of “glue-code" files to enable transparent parallelism for the user, using the inter-step dependences provided in the graph file. The translator also creates code stubs for each of the computation steps, and offers guidelines for creating the user code, i.e., for writing items and spawning new steps. The translator also provides a “full-auto" mode, which assumes all items declared by a step in a graph will be written and all steps prescribed and creates fully compilable steps instead of the step stubs. The user can then simply add his own file containing the implementation of the computation steps, without ever seeing any of the API needed by DFGR or its translation to HC.

Once the user has provided the code for the computation steps, they can use the makefile generated by the translator to build the application. The build system uses the Habanero C compiler and the gcc compiler to generate an executable file. If additional libraries are required, they can be easily added in the provided makefile.

The translator we use in this thesis is based on the same principles presented in the previous MS thesis [23] so we keep its description brief, however we mention that it has been redesigned for the new language specification we propose for DFGR. An additional contribution discussed in Section 4.3 is extending the translator to transform DFGR dependences into a format which can be analyzed by the polyhedral model.

4.1.2 Defining collections

For efficiency, we implement item collections in DFGR using a hash-table for each collection, an approach similar to previous implementations of the CnC model. Indexing in the hash-table is done via the key which uniquely identifies each item in the collection. As items must adhere to the DSA rule, we have created safety nets to warn users an item with the same tag is written multiple times. We do however allow items to be cleared from a collection once they are no longer used. This information can provided by the user when writing to a collection, by specifying a “get-count", which is a number equal to the number of times the item will be read. If this information is not specified, the item will remain in the collection for the duration of the graph execution.

Steps are asynchronous pieces of computation, that we implement using the `async` construct in the Habanero-C [24] task parallel language. A collection of steps is only a theoretical construct, which refers to the fact that these computations perform the same function, and their only side effect are the items they write and the new steps they transitively create.

4.1.3 DFGR runtime

The implementation of the DFGR model we present in this thesis relies on a runtime built on top of the Habanero-C programming language. It uses the `async` construct to create asynchronous running steps and the `finish` construct to wait for the termination of the graph. The DFGR runtime implementation involves ensuring that steps are only started when all the data they need for execution is available. This is achieved by testing the satisfiability of all dependences before each steps’s execution and queuing up the steps on the missing data. In the DFGR runtime, the creator *C* of a data item takes the responsibility of checking for waiting steps. If the current data is the last one in the waiter's dependency list, *C* sends the new steps to the scheduler for execution. Alternatively, it will reenqueue the step on the next missing piece of data it needs. We only use one level of `finish` when executing a DFGR program because we include additional *data-driven* execution constraints on `async` tasks that control when `async` tasks become ready for execution. Note that since DFGR is designed as a high-level representation of data-flow application, its concrete implementation can vary. The implementation we presenting this paper can be used as a reference for future works.

4.2 Experimental comparison with a reference dataflow model implementation

We compare our implementation of the DFGR model with a reference dataflow model implementation: Intel's Concurrent Collection [16]. Our results were obtained on an Intel Westmere node with 12 core Intel Xeon CPU X5660 running at 2.80GHz. The results presented here are an average of 5 runs for tiled implementations of a series of benchmarks (the implementation is C based with a standard deviation below 5%). We use the Cholesky Factorization, Black-Scholes and Matrix Inverse benchmarks provided along with the latest release: 1.0.100, which include the "tuner" interface, thus having already been appropriately tuned. We also create the Smith-Waterman benchmark in Intel's CnC and compare it with our DFGR implementation.

- *Smith-Waterman* is an algorithm from the biology domain, which performs sequence alignment between two strings or nucleotide or protein sequences. We run the alignment algorithm for 2 strings of size 50000 each, with a tile size of 400 in each dimension.

- *Black-Scholes* is a financial application that computes stock values. We use an input size of 1,000,000 and granularity 128.

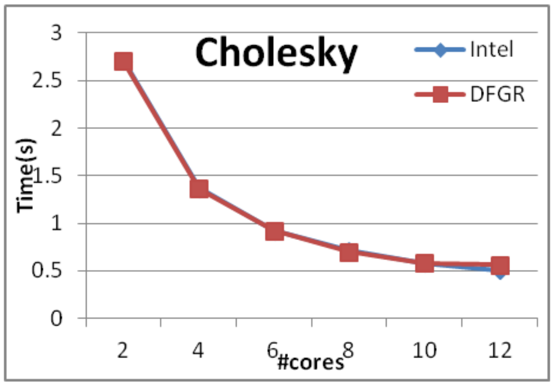
- *Cholesky Factorization* is a linear algebra benchmark that decomposes a symmetric positive definite matrix into a lower triangular matrix and its transpose. The input matrix size is 3000×3000 and the tile size is 150×150.

- *Matrix Inverse* creates the inverse of a symmetric positive definite matrix using an algorithm which decomposes the input matrix into tiles and computes partial matrices before obtaining the inverse. The input matrix size is 4096×4096 and the tile size is 64×64.

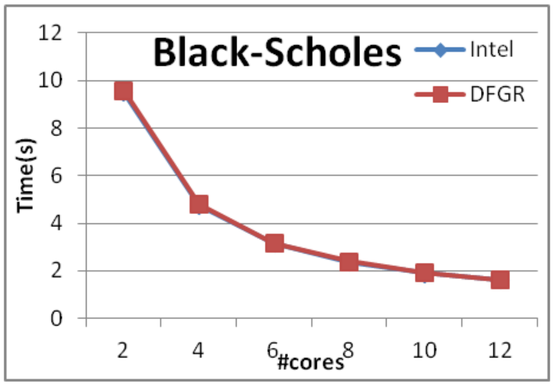
For all benchmarks we use the strict preconditions model, for performance [40], which models applications where all step dependences are known in advance – outlined in a DFGR format in our benchmarks – and steps only start where all inputs are available. We see in Figure 4.2 that all benchmarks described above have good scaling results and are on par or better than the reference implementation. In addition, the DFGR model offers great user productivity due to the automatic generation of the code glueing together a parallel application, relying on the user only to describe the algorithm in DFGR and to provide the computation kernels.

4.3 Mapping DFGR to a polyhedral compiler

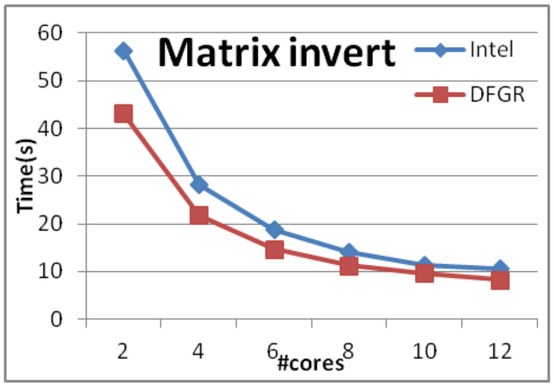
DFGR has been designed with concepts from polyhedral representation and optimization in mind, to allow for graph transformations using the polyhedral model [34]. In a very simplistic manner, DFGR can be seen as a mean to represent a polyhedral dependence



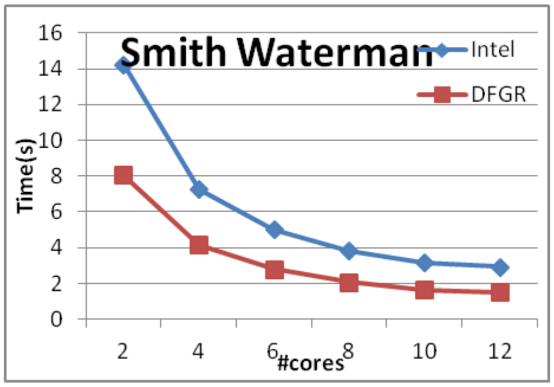
(a) Cholesky



(b) Black-Scholes



(c) Matrix invert



(d) Smith-Waterman

Figure 4.2 : Scalability results for benchmarks implemented using DFGR

graph requiring dynamic single assignment for all arrays. In order to optimize DFGR programs using the polyhedral framework, we need to address the following problems.

- Define the sub-class of DFGR programs that are amenable to polyhedral transformations, and implement the associated recognizer.
- Convert DFGR idioms such as item collections, step prescription, item- and step-to-step dependences, etc. into a polyhedral representation using iteration domains, access functions, and dependence relations.
- Legality analysis of the DFGR graph and detection of deadlocks.

We first give an overview of the polyhedral compitalion framework , then discuss how we address these problems. We then present the transformations performed by our toolchain and give an overview of the user flow for converting DFGR programs to polyhedral.

4.3.1 Polyhedral compilation framework

The polyhedral model is a flexible representation for arbitrarily nested loops.

Loop nests amenable to this algebraic representation are called *Static Control Parts* (SCoPs) and represented in the SCoP format, where each statement contains three elements, namely, iteration domain, access relations, and schedule. SCoPs require their loop bounds, branch conditions, and array subscripts to be affine functions of iterators and global parameters.

Iteration domain, $\mathcal{D}^S \ni \vec{i}$: A statement S enclosed by m loops is represented by an m-dimensional polytope, referred to as an iteration domain of the statement [48]. Each element in the iteration domain of the statement is regarded as a statement instance $\vec{i}$.

Access relation, $\mathcal{A}^S(\vec{i})$: Each array reference in a statement is expressed through an access relation, which maps a statement instance $\vec{i}$ to one or more array elements to be read/written [49]. This mapping is expressed in the affine form of loop iterators and global parameters; a scalar variable is considered as a degenerate case of an array.

Schedule, $\Theta^S(\vec{i})$: The sequential execution order of a program is captured by the schedule, which maps instance $\vec{i}$ to a logical time-stamp. In general, a schedule is expressed as a multidimensional vector, and statement instances are executed according to the increasing lexicographic order of their time-stamps.

Dependence Polyhedra, $\mathcal{D}^{S \rightarrow T} \ni (\vec{i}, \vec{i'})$: The dependences between statements S and T are captured by dependence polyhedra—i.e., the set of n-dimensional polytopes where n is sum of dimensionalities of domains $\mathcal{D}^S$ and $\mathcal{D}^T$. Given two statement instances $\vec{i} \in \mathcal{D}^S$ and $\vec{i'} \in \mathcal{D}^T$, $\vec{i'}$ is said to depend on $\vec{i}$ if 1) they access the same array location, 2) at least one of them is a write and 3) $\vec{i}$ has lexicographically smaller time-stamp than $\vec{i'}$—i.e., $\Theta^S(\vec{i}) \prec \Theta^T(\vec{i'})$.

4.3.2 DFGR restrictions for enabling polyhedral optimizations

To facilitate the conversion to a polyhedral representation, we focus on a restricted subset of DFGR that can be summarized as follows: item keys are described as affine expressions of step tags, and all steps are started by the environment. item collection before the conversion to the polyhedral representation. Note that a step-to-step dependence is converted into step-to-item and item-to-step dependences using a new item collection. In addition to rectangles by ranges model [33], DFGR supports simple polyhedra shaped by affine inequalities of step tags when starting steps and accessing items. In this work we use $\mathcal{Z}$-polyhedra to capture sets of integer vectors, and multidimensional affine functions to represent relations between these vectors. We remark that DFGR can represent arbitrary (macro-)dataflow programs, therefore not all DFGR graphs can be represented within the polyhedral framework we propose. The set of DFGR graphs that can be optimized is driven by the restrictions we impose on each DFGR constructs, and is summarized following sections

In practice, ranges and simple polyhedra are often enough to express the tag sets needed to model an application. They also come with the benefit of easy compilation to a loop-based language, which we will use to generate parallel OpenMP code.

The implementation we propose relies on generation of C code due to the benefits of high performance given by a low level language and the ease of programming provided by DFGR, which abstracts applications using a high-level representation. This approach is also appropriate for using DFGR as an embedded DSL, since the OpenMP code that our toolchain generates can be integrated into larger code bases (in effect, an OpenMP parallel region is generated for each DFGR parallel region), while the user steps, which the generated code calls, can themselves be optimized routines or library calls (possibly with non-affine data accesses, since only the DFGR program is processed by the polyhedral framework, not the internal step code).

4.3.3 Polyhedral representations of DFGR program

This section introduces our approach for creating a polyhedral representation of a DFGR program, in which each step is modeled as a “black box" statement, and dependences are extracted from the dataflow model.

The first stage of applying polyhedral optimization to DFGR is to translate a DFGR into polyhedral concepts, that is (1) iteration domains for each step, properly modeling the dynamic set of executions of step instances; (2) access functions for each step to capture the data accessed by each step instance; and (3) dependence relations between step instances, to enable correct semantics-preserving transformations of the program.

Converting step prescription

The first DFGR construct to translate is step prescription, that is the set of all dynamic instances of each step. The purpose is to capture, for each step, its polyhedral iteration domain. We leverage the fact that tag functions are restricted to producing integer tuples, which are amenable to representation as $\mathcal{Z}$-polyhedra, that is union of parametric polyhedra of integer tuples intersected with multidimensional affine lattices.

The prescribe operation `::` can be between arbitrary step instances, that is the graph may contain prescriptions like `(foo: 3*i,j)::(bar: 2*j,i)`; where a particular step instance (i, j) prescribes another step instance identified by the tag function $f(3*i, j) = (2*j, i)$. Sets of instances may be produced, using ranges or regions [33]. To correctly model the iteration domain of `bar` in this example we need to analyze the tag functions of both operands of the prescribe operation, as well as being able to model the iteration domain of the left-hand side operand.

We propose an algorithm to compute iteration domains for steps as follows. It iterates over all prescribe operations, propagating the left hand side iteration domain to the right hand side if and only if the relationship between tag functions is a multidimensional affine function and if the iteration domain of the left hand side is computable and known. In addition, when regions (or ranges as `1..M` above) [33] are used to model sets of tags we require the region to be itself describing a union of $\mathcal{Z}$-polyhedra, and the prescriber to have a single iteration. Affineness of a region is determined by simply ensuring all inequalities in the region are affine expressions and free of uninterpreted functions. We implement a simplified version of the algorithm that takes into account all step prescriptions from the environment, analysis the right hand side of the prescription and generates the correct iteration domains for all steps.

Converting data and control dependences

Data and control dependences need to be captured using a polyhedral representation to enable optimization. First, we remark that control dependences in DFGR (e.g., `(S:i) -> (T:i)`) can be seamlessly converted to data dependences by introducing fake item collections for analysis purpose (e.g., `(S:i) -> [fk:i] -> (T:i)`), therefore we only focus below on data dependence modeling. To properly model data dependences we need to express the data being accessed by each step instance.

Modeling item collections in the polyhedral representation does not need any form of treatment. We here rely on the properties that each item in a collection is uniquely identified by an integer tuple. That is, item collections can be seen as multidimensional arrays for polyhedral analysis purpose. When generating the information regarding item collections, we ensure using a unique numeric identifier for each collection, as expected by the polyhedral compiler.

Access functions allow to associate to a particular step instance the data which is being read and written. To determine if an access function can be built from a DFGR read/written item collection, we simply analyze its tag function and ensure it is an affine multi-dimensional function. Similarly to prescription, ranges and regions can be used to describe a set of data being accessed by a step instance, or a set of step instances. This is why we actually use *access relations* instead of simple access functions. If for instance a range is used to describe the set of data accessed, such as in:

```
[A : {i-1..i+1},{j-1..j+1}] -> (S : i,j);
```

its access relation is written:

$$C : \{(i,j) \rightarrow (d1,d2) \mid i-1 \leq d1 \leq i+1,\ j-1 \leq d2 \leq j+1\}$$

With the information from step prescriptions, and data and control dependences we are ready to convert the DFGR specification to a format that the polyhedral compiler understands. In practice we generate a file in the ScopLib [50] format, which is then read in by the polyhedral compiler.

SCoP for DFGR model

As shown in Section 4.3.1, the original SCoP format consists of three components: iteration domain, access relation, and schedule. The restricted DFGR model defined in Section 4.3.2 allows a step collection's iteration domain to be represented as a convex polytope, and the I/O relations of each step instance to be modeled as read/write access relations. Examples of DFGR code fragments and their SCoP representations are shown below.

$$\texttt{[A:i-1,j+1]->(S:i,j)->[B:i,j]} \Leftrightarrow \mathcal{A}^{S_R}(i,j) = (A, i-1, j+1),\ \mathcal{A}^{S_W}(i,j) = (B,i,j)$$

$$\texttt{env::(S:\{1 .. N\},\{i .. M\})} \Leftrightarrow \mathcal{D}^S = \{(i,j) \in \mathbb{Z}^2 \mid\ 1 \leq i \leq \texttt{N} \wedge i \leq j < \texttt{M}\}$$

Instead of specifying the sequential execution order (total order) among all step instances, the DFGR model enforces ordering constraints via dataflow: a step instance is ready to execute only once all of its input items (data elements) are available. Therefore, the SCoP format specialized for DFGR contains iteration domains and access functions, but no explicit schedule.

Dependence computations

To compute dependences between any two step instances, we need to determine their *Happens-Before* (HB) relation — i.e., which instance must happen before another [51]. Instead of the lexicographic comparison of schedules $\Theta^S(\vec{i}) \prec \Theta^T(\vec{i'})$ in Section 4.3.1, we define the HB relation between instance $\vec{i}$ of step S and $\vec{i'}$ of step T as:

$$\mathcal{HB}^{S\rightarrow T}(\vec{i},\vec{i'}) \;\equiv\; \mathcal{A}^{S_W}(\vec{i}) = \mathcal{A}^{T_R}(\vec{i'}) \;\wedge\; (S \neq T \vee \vec{i} \neq \vec{i'})$$

This simply captures the ordering constraints of the DFGR model: step instance $\vec{i'}$ reading an item cannot start before step instance $\vec{i}$ writing that item, even if step instance $\vec{i'}$ of T appears lexically before instance $\vec{i}$ of step S in the DFGR program. According to the definition in Section 4.3.1, dependence polyhedra between steps S and T are expressed as:

$$\mathcal{D}^{S\rightarrow T} \;\equiv\; \{(\vec{i},\vec{i'}) \mid \vec{i} \in \mathcal{D}^S \;\wedge\; \vec{i'} \in \mathcal{D}^T \;\wedge\; \mathcal{A}^{S_W}(\vec{i}) = \mathcal{A}^{T_R}(\vec{i'}) \;\wedge\; \mathcal{HB}^{S\rightarrow T}(\vec{i},\vec{i'})\}$$

, which captures that $\vec{i}/\vec{i'}$ is an instance of step S/T, $\vec{i}$ writes the item read by $\vec{i'}$ (access equation), and $\vec{i}$ happens before $\vec{i'}$ (HB relation). Because of the dynamic single assignment rule, the DFGR model disallows Write-After-Write dependence and Write-After-Read dependences. The next section outlines how polyhedral analysis can be used to check of these error cases.

4.3.4 Legality analysis

This section introduces the compile-time analyses to verify the legality of a DFGR program. Enforcing the DFGR semantics, it detects the violation of the dynamic-single-assignment rule, plus three types of deadlock scenarios.

Violation of the single-assignment rule is equivalent to the existence of Write-After-Write dependences, and is represented by the following condition, which indicates that instances $\vec{i}$ and $\vec{i'}$ write an identical item (data element):

$$\exists\vec{i} \in \mathcal{D}^S,\ \exists\vec{i'} \in \mathcal{D}^T\ :\ \mathcal{A}^{S_W}(\vec{i}) = \mathcal{A}^{T_W}(\vec{i'})\ \wedge\ (S \neq T \vee \vec{i} \neq \vec{i'})$$

Self deadlock cycle is the simplest case of deadlock. An instance $\vec{i}$ needs to read an item which is written by $\vec{i}$ itself, thereby resulting in indefinite blocking.

$$\exists\vec{i} \in \mathcal{D}^S\ :\ \mathcal{A}^{S_W}(\vec{i}) = \mathcal{A}^{S_R}(\vec{i})$$

General deadlock cycle is the second of deadlock scenarios, where the dependence chain among multiple step instances creates a cycle. Any instance on the cycle waits for its predecessor to complete and transitively depends on itself. As discussed in Section 4.3.5, transformations in the polyhedral model are equivalent to finding new legal schedules (total orders among step instances) according to the dependence polyhedra. Therefore our optimizer is not permitted to find any schedules for dependence polyhedra that exhibit a deadlock cycle.

$$Transform(\{\mathcal{D}^{*\rightarrow *}\}) = \emptyset$$

Deadlock due to absence of producer instance is the third deadlock scenario. Even without a cycle in the dependence chain, it can be possible that a step instance $\vec{i'}$ needs to read an item that any other step instance does not write. Detecting this scenario is represented by the following condition, which means there is no step instance $\vec{i}$ that writes an item to be read by $\vec{i'}$. Note that the items written/read by the environment `env` are also expressed as domains and access relations [*].

$$\exists \vec{i'} \in \mathcal{D}^{T} \;:\; \neg\,(\exists \vec{i} \in \mathcal{D}^{S} \;:\; \mathcal{A}^{S_W}(\vec{i}) = \mathcal{A}^{T_R}(\vec{i'}))$$

4.3.5 Transformations

Given a set of dependence polyhedra $\{\mathcal{D}^{*\to *}\}$ that captures all program dependences, the constraints on valid schedules are:

$$\Theta^{S}(\vec{i}) \prec \Theta^{T}(\vec{i'}),\;\; (\vec{i},\vec{i'}) \in \mathcal{D}^{S\to T},\;\; \mathcal{D}^{S\to T} \in \{\mathcal{D}^{*\to *}\}$$

For any dependence source instance $\vec{i}$ of step S and target instance $\vec{i'}$ of step T, $\vec{i}$ is given a lexicographically smaller time-stamp than $\vec{i'}$. The transformation phase selects a valid schedule per step based on performance heuristics — maximizing objective functions. There have been a variety of polyhedral optimizers in past work with different strategies and objective functions e.g., [52, 53, 54].

We used the PolyAST [54] framework to perform loop optimizations, where the dependence information provided by the proposed approach is passed as input. PolyAST employs a hybrid approach of polyhedral and AST-based compilations; it detects reduction and doacross parallelism [55] in addition to regular doall parallelism. In the code generation stage(CLooG [47]), doacross parallelism can be efficiently expressed using the proposed doacross pragmas in OpenMP 4.1 [56, 57]. These pragmas allow for fine-grained synchronization in multidimensional loop nests, using an efficient synchronization library [58].

[*]In future work, we may consider the possibility of not treating this case as an error condition by assuming that each data item that is not performed in the DFGR region has a initializing write that is instead performed by the environment.

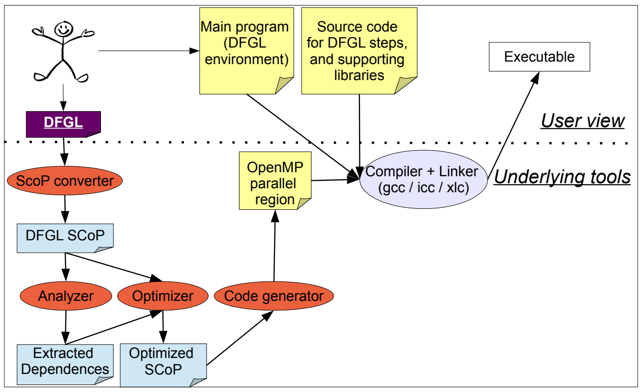


Figure 4.3 : Optimization and build flow for a DFGR parallel region.

4.3.6 Flow for converting DFGR to polyhedral

The flow for compiling a DFGR parallel region is shown in Figure 4.3. The user creates the DFGR description and provides the main program (DFGR environment) and codes for the compute steps. Then, they use our toolchain, which couples an extended translator [19] that we created for conversion to SCoP, and an extension to ROSE Compiler framework [59, 54], to obtain an executable for running the application.

The first component of the toolchain is the *SCoP converter* that transforms the DFGR representation into a simplified SCoP format, providing only the domain and access information as described in Section 4.3.1. Next, we use the *Analyzer* to report errors in the input DFGR program and obtain the dependences. The dependences, along with the information from the DFGR SCoP, are then fed into the *Optimizer*. The final stage is the generation of the optimized OpenMP code, which is built together with the user-provided main program, kernels and libraries to obtain the executable.

In section 4.4 we present motivating experimental results showing that using the doacross OpenMP 4.1 backend for DFGR gives a very good performance.

4.4 Experimental results with an OpenMP backend

While in Section 4.2 we have compared DFGR with a backend which relies on the Habanero C language, in this section we test the performance of the model when using an OpenMP backend. To this end, we use the transformation to the polyhedral compiler detailed in Section 4.3. Our results were obtained first on an the same Intel Westmere node with 12 core Intel Xeon CPU X5660 running at 2.80GHz. Then we provide additional results on an IBM POWER7 node containing four eight-core POWER7 chips running at 3.86GHz (we disable SMT and use a maximum of 32 threads).

For benchmarks, we use nine benchmarks: LULESH, Smith-Waterman, Cholesky Factorization, and six stencil kernels from PolyBench [60]. In the graphs in this section we refer to the toolchain DFGR-Polyhedral-OpenMP as DFGL, a notation used for this worked in the published document [61]. We will also include a comparison with the Habanero C runtime for the benchmarks presented in the previous section; we will refer to this as DFGR-HC.

The first example is *Lulesh*, a benchmark needed for modeling hydrodynamics [62]. It approximates the hydrodynamics equations discretely by partitioning the spatial problem domain into a collection of volumetric elements defined by a mesh. In this implementation each element is defined as a cube, while each node on the mesh is a point where mesh lines intersect and a corner to 4 neighboring cubes. The mesh is modeled as a 3D space with N^3 elements and $(N+1)^3$ nodes. The benchmark uses a iterative approach to converge to a stable state. We pretested the application and saw a convergence after 47 iterations, thus in our results we use a fixed number of 50 iterations for simplicity.

The tuning parameters we investigate for Lulesh are: (1) the time tile size for the iteration space, (2) the space tile size used for the 3D mesh and (3) the ordered clause we explain below. The OpenMP backend uses an implementation of the DoAcross model [29], which allows the specification of dependences between iterations, as well as the nesting level where the synchronization should occur. In Listing 4.1 we give two examples using the *ordered* clause for different nesting depth. The src clause identifies uniquely a loop nest using the loop iterators (e.g. i, j), while the sink clause defines the iteration on which the iteration that is about to start has to wait on, effectively specifying loop-carried dependences. The numerical value used in the ordered clause defines the nesting level where the src and sink depends clauses are inserted, thus also specifying the granularity. Provided there are no loop carried dependences in the inner-most loops, a smaller value in the ordered clause will increase the granularity of the computation.

Figure 4.4 gives the results for a 100^3 space domain and our toolchain tiled both the time loop and the 3D loop nest corresponding to the space. In practice we observe the best results for the doacross synchronization level $ordered(4)$, which is what we use in our results. In Lulesh we find that there is an inter-iteration dependence (a loop-carried dependence on the outermost loop) and no dependences on the 3D space loop nest. We also notice little data reuse between iterations, which makes a larger tile size give less performance than a smaller one, even though the input data set is large. We see that even with a time tile size of 2, this leaves only 25 parallel iterations at the outermost doacross loop, which for the POWER7 in particular leads to a smaller speedup. The best results are obtained with no time tiling and a space tile of 8^3, on both Westmere and POWER7. We have run additional results for larger tile sizes on both the time (iteration) dimension and the 3D space dimension and obtained lower speedups. Nevertheless, our results give significantly better scaling results than the reference implementation we use, which is a C++ code parallelized manually using OpenMP [63].

The next benchmark used is *Smith-Waterman*, an algorithm from the computational biology domain, which performs sequence alignment between two strings representing nu-

Listing 4.1: Examples using the ordered clause.

```
#pragma omp parallel for ordered(2)
for (i = ...) {
  for (j = ...) {
    #pragma omp ordered depend(sink: i-1, j)
     for (k = ...) {
       foo();
     }
    #pragma omp ordered depend(src: i, j)
  }
}
```

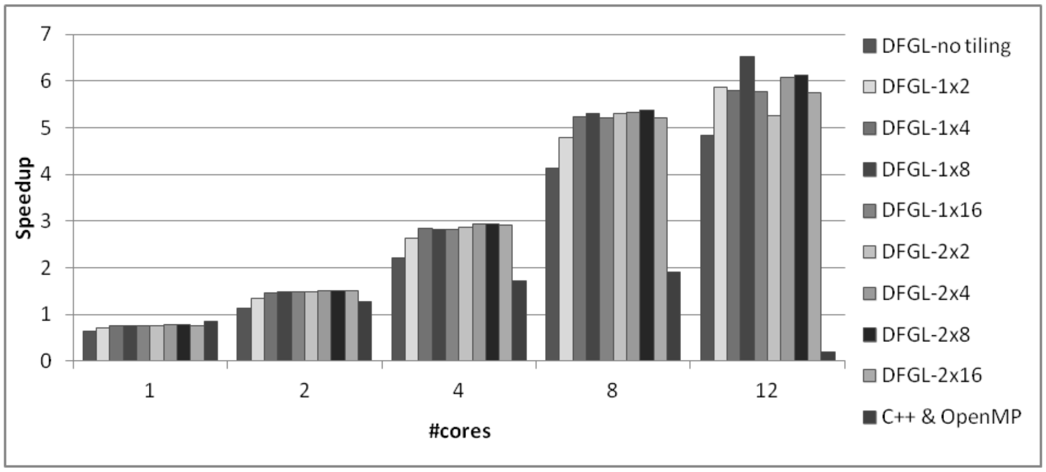
(a) Example of synchronization at depth = 2

```
#pragma omp parallel for ordered(3)
for (i = ...) {
  for (j = ...) {
    for (k = ...) {
    #pragma omp ordered depend(sink: i-1, j, k)
     foo();
    #pragma omp ordered depend(src: i, j, k)
    }
  }
}
```

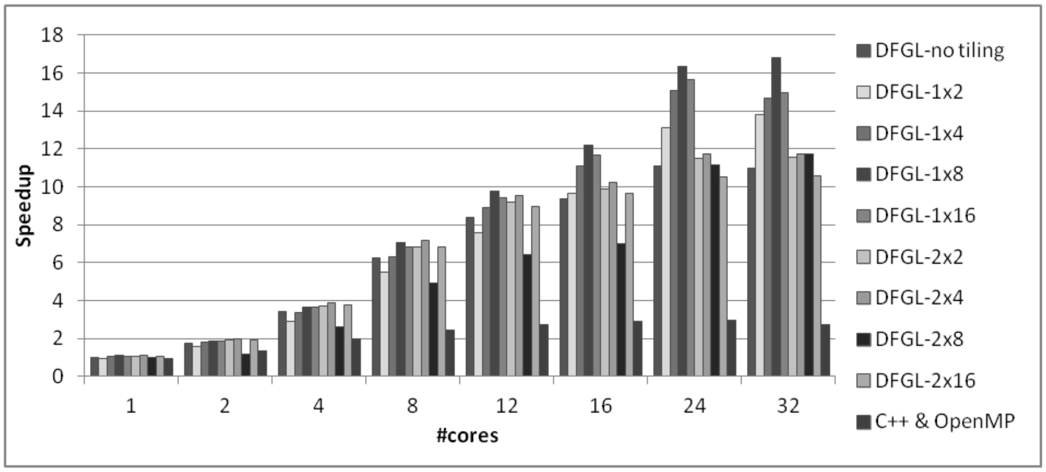
(b) Example of synchronization at depth = 3

cleotide or protein sequences. We used our translator to obtain the polyhedral format for the Smith-Waterman benchmark presented in the Section 4.2. We run the alignment algorithm for 2 strings of size 100,000 each, with a tile size varying between 16 and 1024 in each dimension. As the baseline OpenMP implementation, we manually provided a wavefront doall version via loop skewing.

Figure 4.5 shows the speedup results on our two test platforms, relative to the sequential implementation. We observe that the performance varies depending on the tile size chosen: for Westmere the best tile size is 1024, while for POWER7 the best tile size is 64. However our approach gives a big performance improvement compared with the skewed



(c) Intel Westmere



(d) IBM POWER7

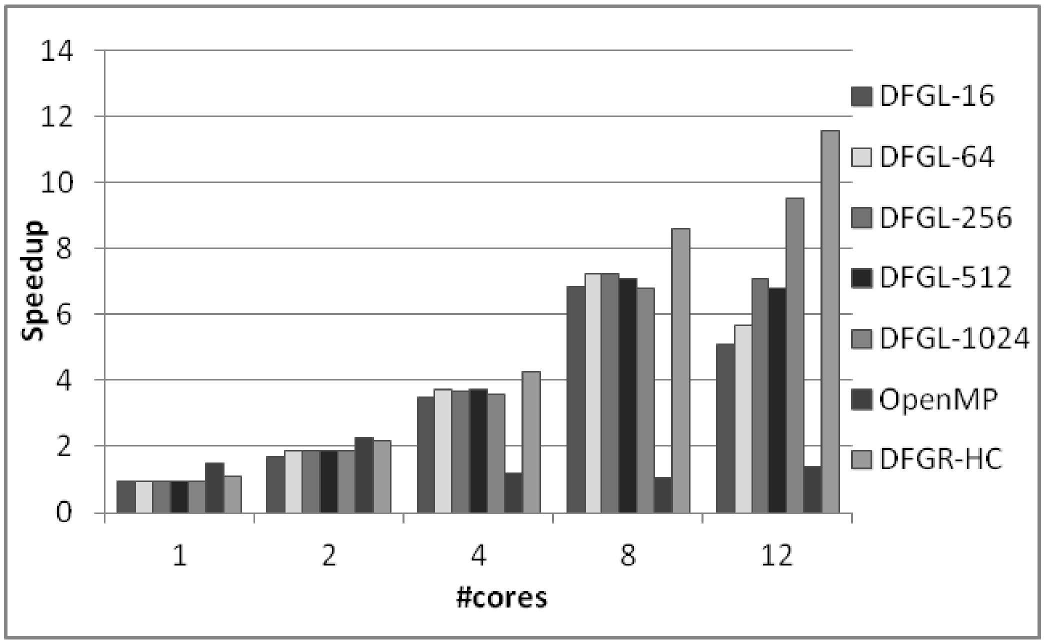
Figure 4.4 : Scalability results for Lulesh benchmark on (a) Westmere and (b) POWER7, with 50 iterations, and 100^3 elements. Comparing OpenMP+DoAcross backend (DFGL) with reference C++ implementation using OpenMP. The X axis the number of cores used. Each set of results shows the performance for a different tile setting using the OpenMP + DoAcross backend and the baseline we compare against.

wavefront OpenMP implementation: up to 6.9× on Westmere and up to 2.3× on POWER7 for the maximum number of cores, due to cache locality enhancement via tiling and efficient doacross synchronizations.

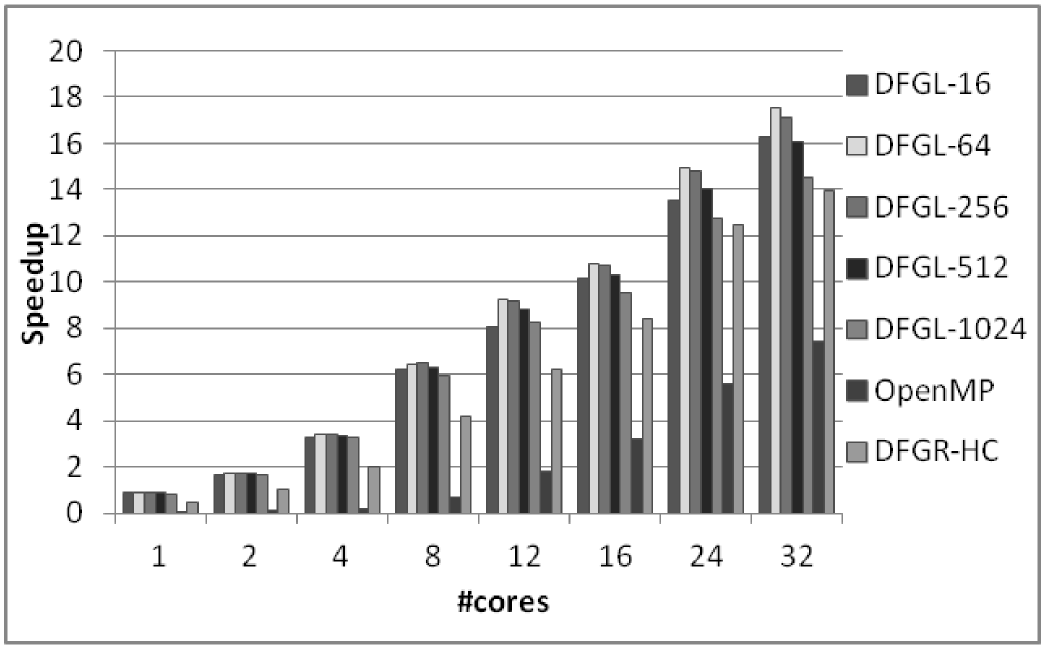
Compared with the Habanero C runtime, we observe that the speedup is lower for DFGL.s We believed the reason to be that the OpenMP backend cannot do the complex tiling that we performed manually in DFGR. More precisely, the computations on the first top row of the matrix and the first left column of the matrix (top and left steps as explained in Section 3.3.2) are computed sequentially before the tiled computation on the rest of the matrix. We manually added an additional OpenMP parallel pragma to ensure this part of the work is also done in parallel, but obtained the same overall results. Further investigation revealed that the underlying OpenMP runtime was originally tuned for the POWER 7 machine, a fact that can be seen in our results.When running on the POWER 7, the tables turn and the Habanero C backend with the best tilesize obtains less performance than the best setting using the OpenMP+DoAcross backend. We argue that this result further motivates the use of DFGR as a high-level representation, as it offers an easy means for expressing applications and automatic portability to a variety of backends, which can used for performance tuning on different platforms.

Cholesky Factorization is a linear algebra benchmark that decomposes a symmetric positive definite matrix into a lower triangular matrix and its transpose. The input matrix size is 2000 × 2000 and the generated code has 2D loop tiling with tile size varying between 4 and 32. Figure 4.6 shows that even though this approach does not yield a large speedup, it still gives improvement compared to the OpenMP implementation: 1.4× on Westmere and 3.0× on POWER7.

Previous work has shown that Cholesky Factorization can yield very good speedup [64]. We use the approach described in [64] to create tiled steps. Note that our toolchain remains

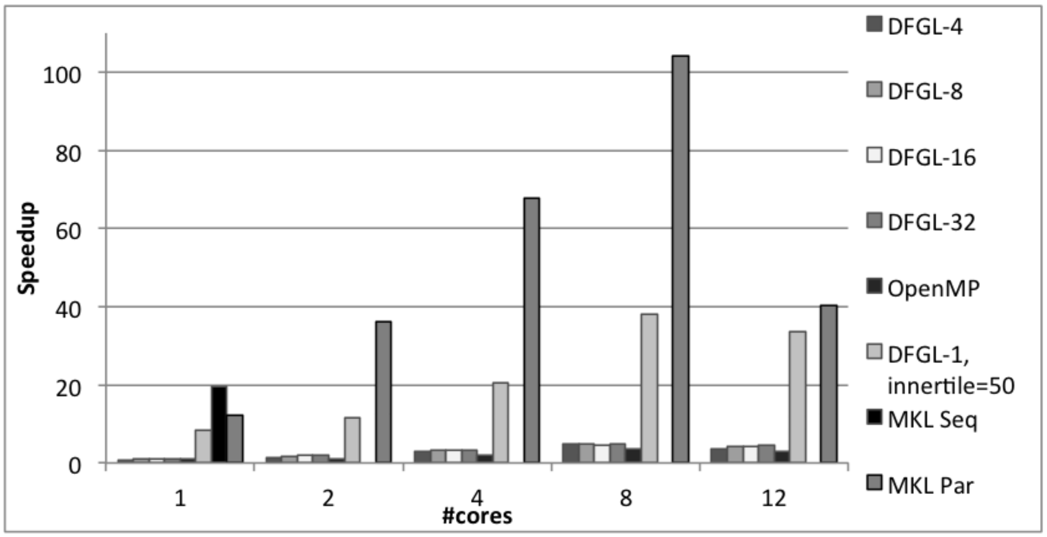


(a) Intel Westmere

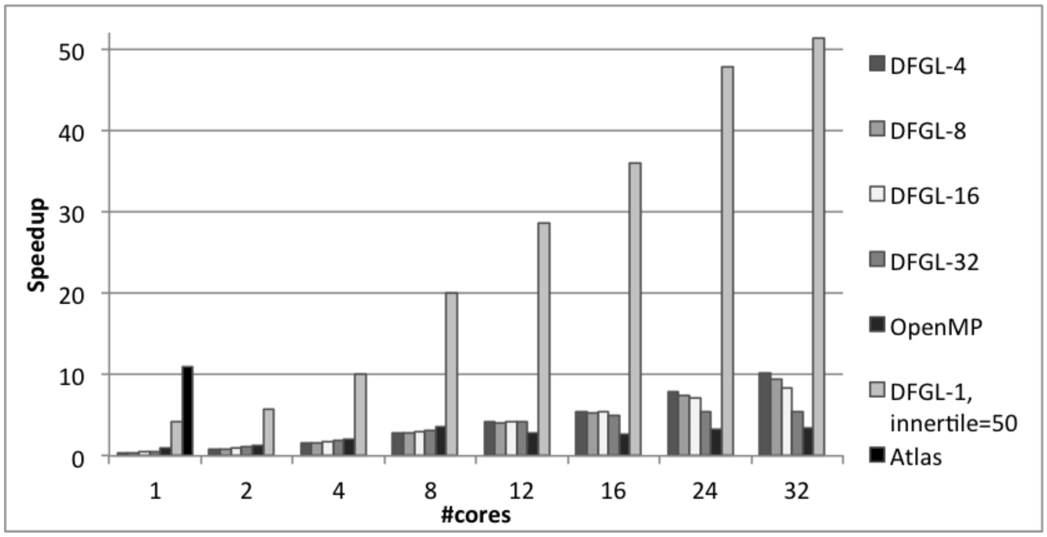


(b) IBM POWER7

Figure 4.5 : Scalability results for Smith-Waterman benchmark on (a) Westmere, (b) POWER7, with 100,000 elements. Comparing OpenMP+DoAcross backend (DFGL) with parallel OpenMP implementation and with DFGR manual tuning with best tilesize. The X axis the number of cores used. Each set of results shows the performance for a different tile setting using the OpenMP + DoAcross backend and the two baselines we compare against.



(a) Intel Westmere



(b) IBM POWER7

Figure 4.6 : Cholesky Factorization using 2000x2000matrix on (a) Westmere, (b) POWER7. Results are for loop tiling using tile sizes between 4 and 32, OpenMP parallelism, data tiling resulting of the inner steps and reference MKL/Atlas implementations.

the same however: the input DFGL representation does not change and the generated code is the same. The only thing that changes are the user provided steps, which instead of processing a single element of the matrix now process a tile of size 50×50. We observe a great improvement in speedup, which becomes superlinear due to the additional improvement attributes to cache locality. This improvement is up to 15× on the Westmere and up to 10.8× on the POWER7 when compared with the standard OpenMP implementation. We include results for the MKL/ATLAS libraries. MKL is the best tuned library for Intel platforms and we include results for both sequential and parallel implementations of MKL. The parallelism offered by DFGR outperforms the sequential library when using more than 4 cores, but the parallel library is better tuned than our approach. However, due to the expressiveness of our model, we can use MKL library calls inside the DFGR steps, which we expect to improve the performance further, as shown in previous work ([64]). On POWER7 we use ATLAS - the sequential library -, as MKL cannot run on POWER7 and no parallel libraries were available. As on the Westmere, we see that DFGR outperforms the reference library when using more than 4-way parallelism.

Intuitively, the approach of enabling tiling of the data inside the user provided steps leads to better performance due to the optimized cache locality. We tested this hypothesis by measuring cache misses on the POWER7. Figure 4.7 shows that the fact the cache misses measured are correlated with the performance obtained, i.e. the best execution is obtained by the approach with the least cache misses.

These results further motivate our work, since the application tuning can be accomplished both by the polyhedral transformations and the user by replacing the steps with optimized versions. For example, in the case of Cholesky, it is possible to call optimized MKL/ATLAS kernels inside the user steps. In our results, these steps are regular sequential steps. Further, since DFGR can be used as an embedded DSL, the OpenMP code being

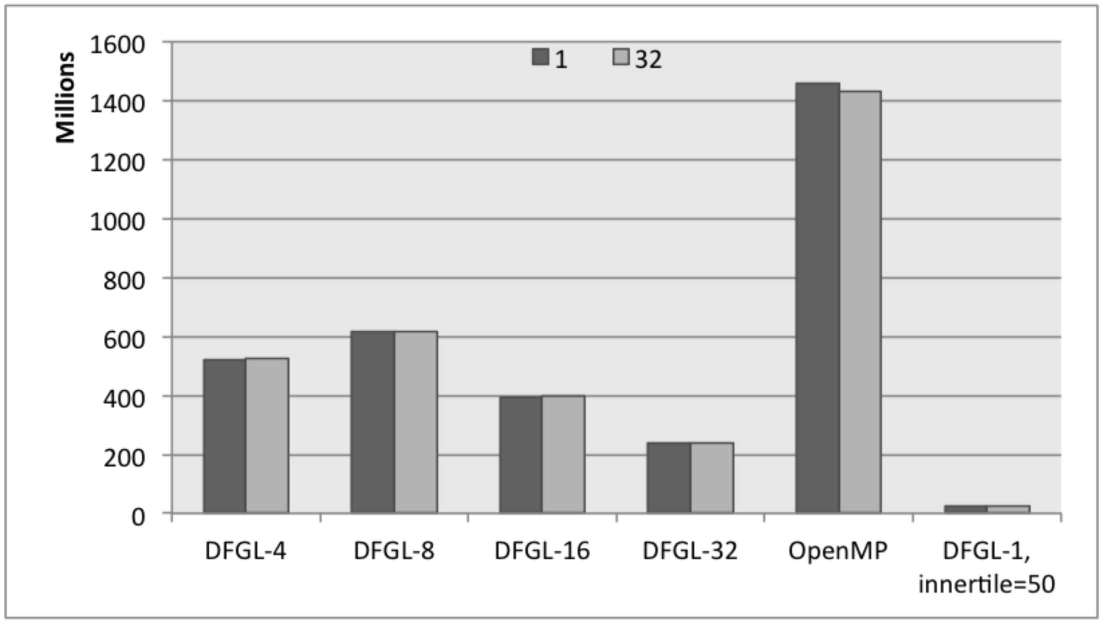
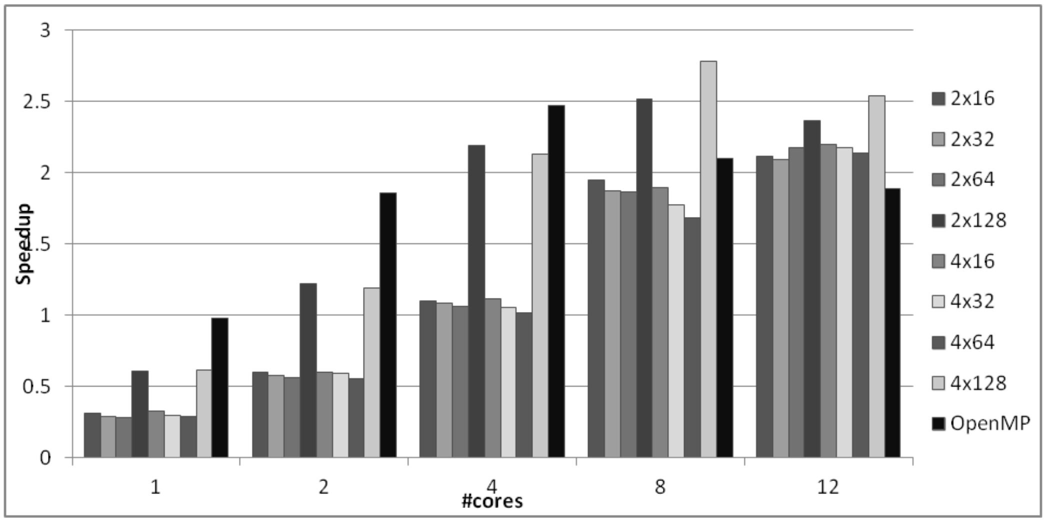


Figure 4.7 : Cache misses for Cholesky benchmark on POWER7, using 1 and 32 cores. Comparing OpenMP+DoAcross backend (DFGL) with parallel OpenMP implementation and with DFGR manual tuning with best tilesize.

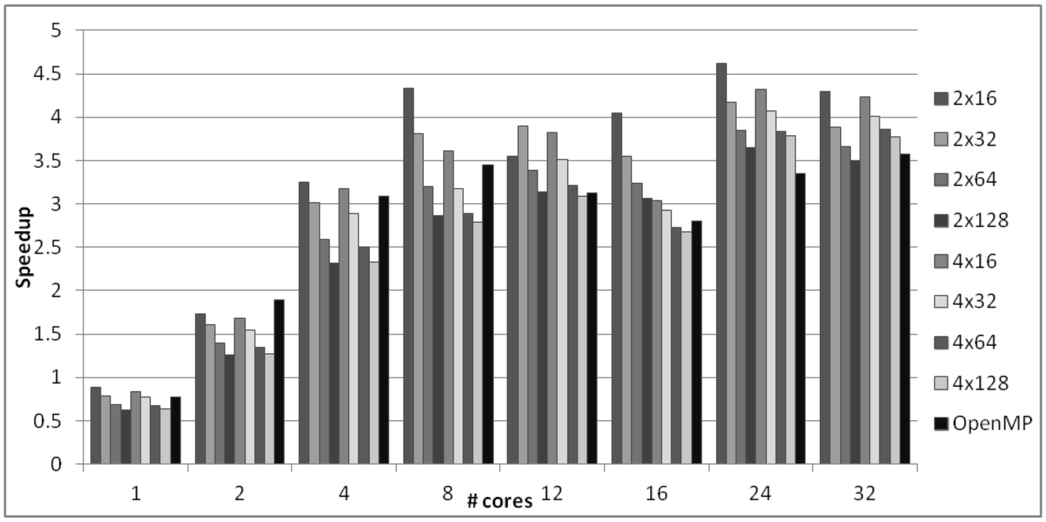
generated can be incorporated in larger applications and coupled with optimized user steps.

Finally, we give the scaling results for the stencil benchmarks from the Polybench suite [60]: *Jacobi-2D* (Figure 4.8), *Jacobi-1D* (Figure 4.9), *Seidel-2D* (Figure 4.10), *FDTD* (Finite Different Time Domain) (Figure 4.11), *FDTD-APML* (FDTD using Anisotropic Perfectly Matched Layer) (Figure 4.12) and *ADI* (Alternating Direction Implicit solver) (Figure 4.13). For each benchmark we created the baseline OpenMP implementations in a standard manner: parallelism added at the outer most loop for fully parallel loops and after skewing for loops with loop-carried dependences.

The results show that the best tile sizes vary between platforms: on the Westmere the best results are generally for the larger time tile (4) and the largest space tile size (128), while for the POWER7 the best results are for the smaller time tile (2) and the smallest space tile (16). We also note that the results obtained using the DFGR toolchain outperform the OpenMP implementations for most cases, with up to 1.8× speedups.

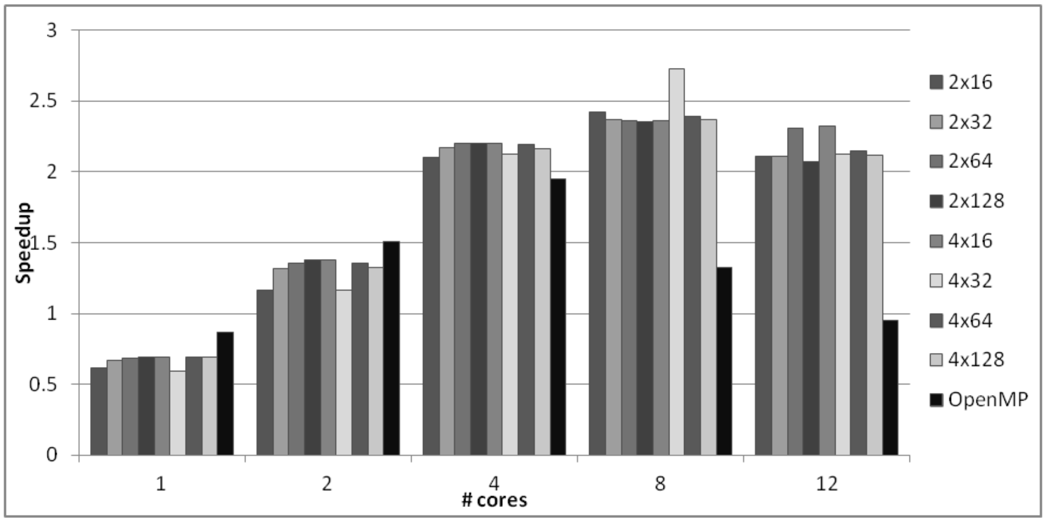


(a) Intel Westmere

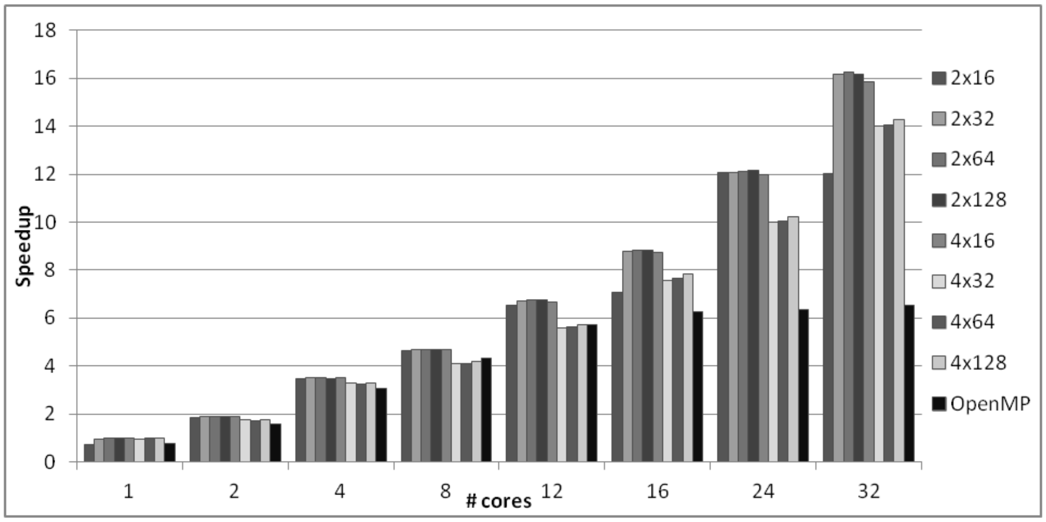


(b) IBM POWER7

Figure 4.8 : Jacobi 2D benchmark from the Polybench suite. Results compare DFGL tiling with standard OpenMP parallel version.

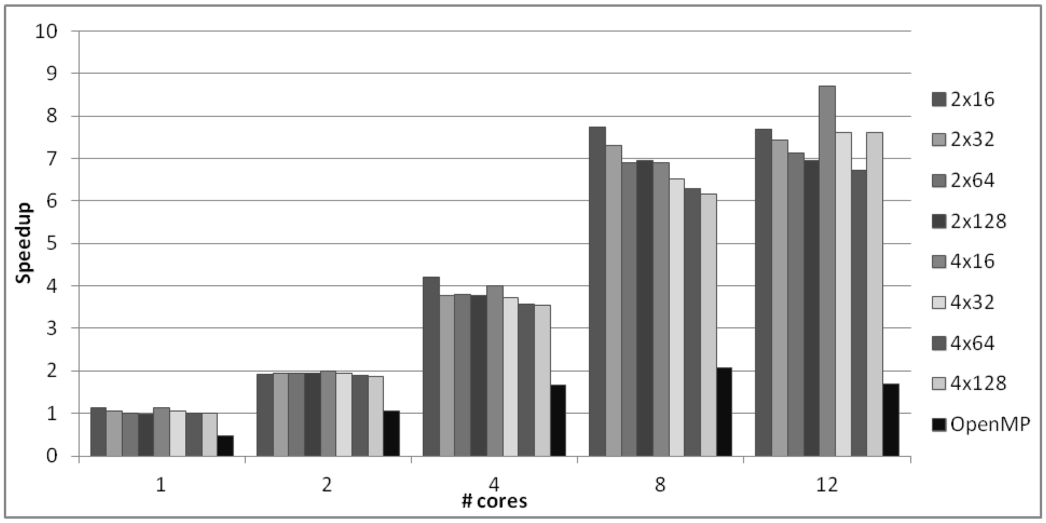


(a) Intel Westmere

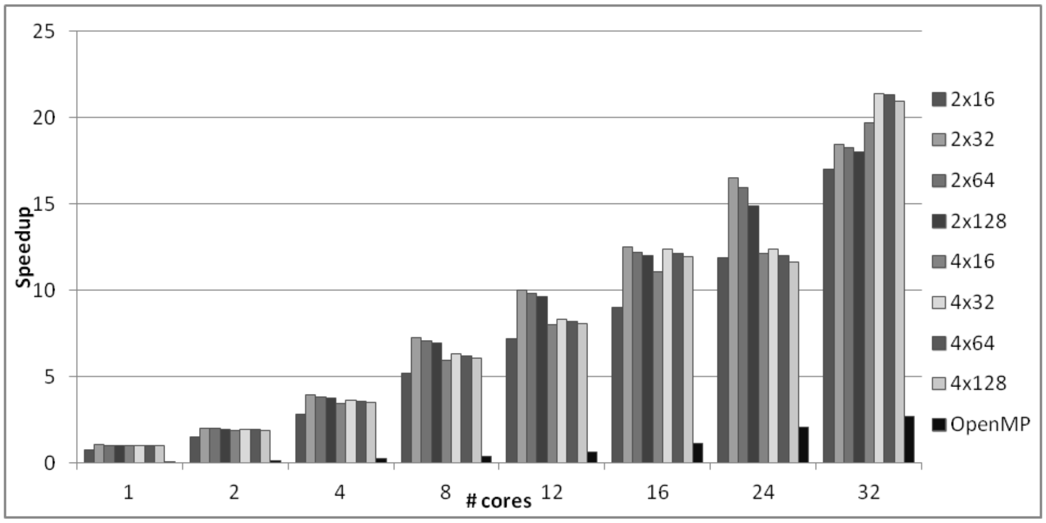


(b) IBM POWER7

Figure 4.9 : Jacobi 1D benchmark from the Polybench suite. Results compare DFGL tiling with standard OpenMP parallel version.

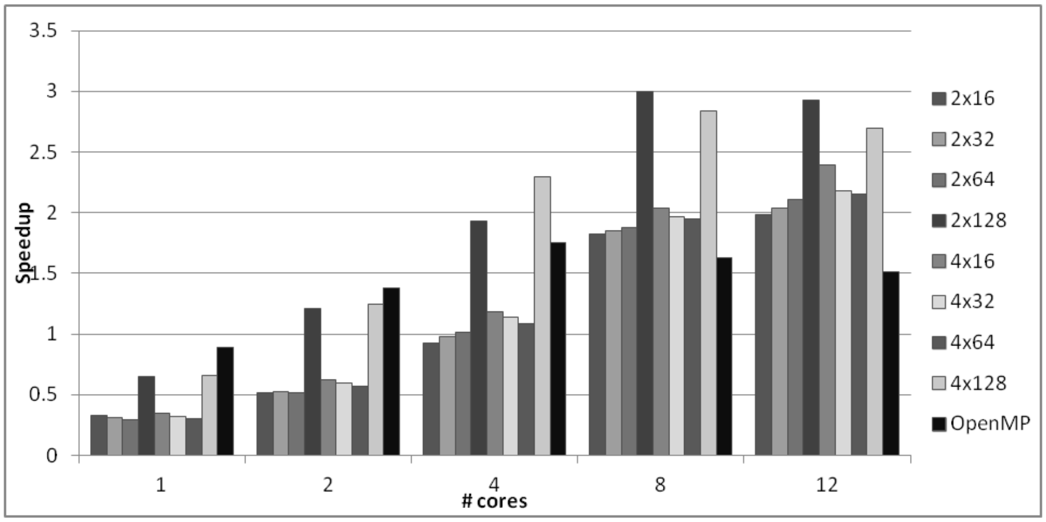


(a) Intel Westmere

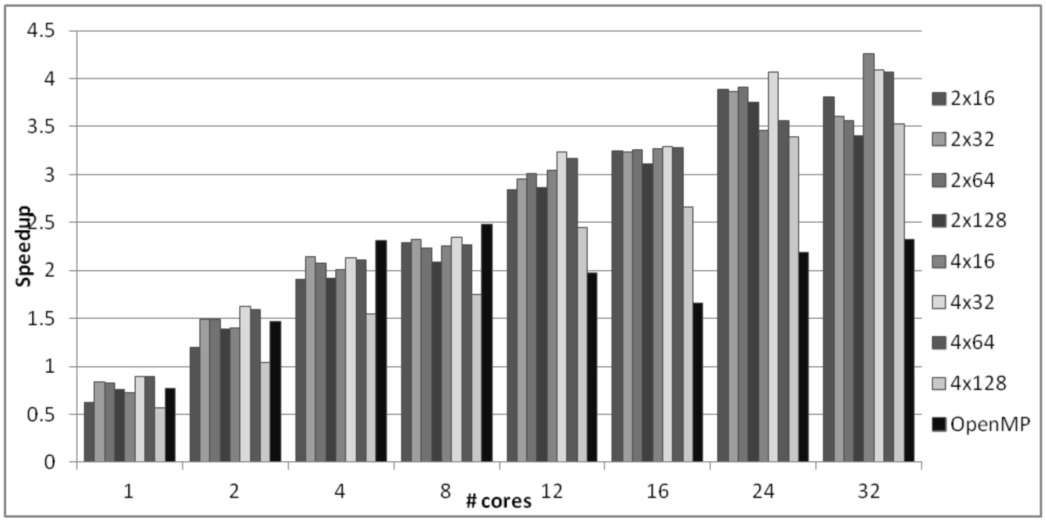


(b) IBM POWER7

Figure 4.10 : Seidel 2D benchmark from the Polybench suite. Results compare DFGL tiling with standard OpenMP parallel version.

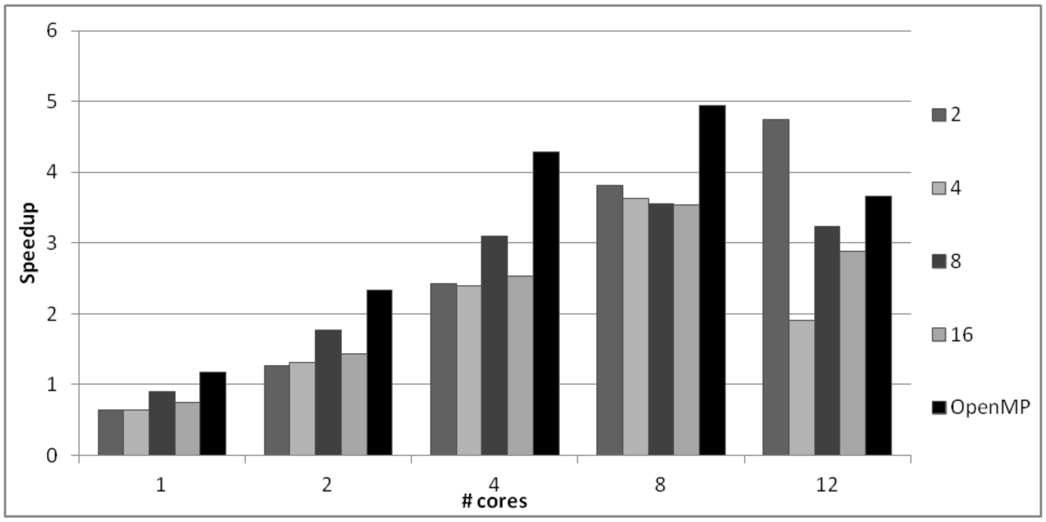


(a) Intel Westmere

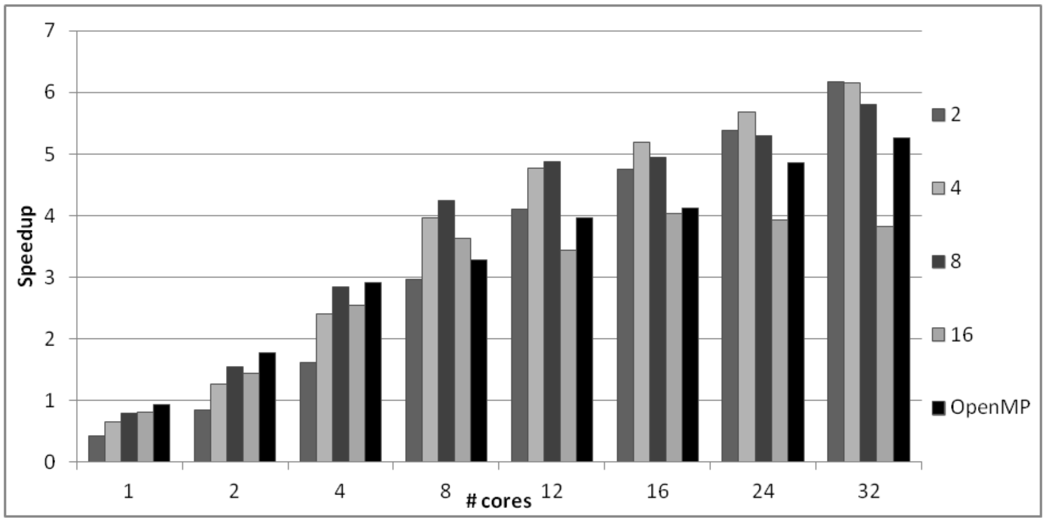


(b) IBM POWER7

Figure 4.11 : FDTD benchmark from the Polybench suite. Results compare DFGL tiling with standard OpenMP parallel version.

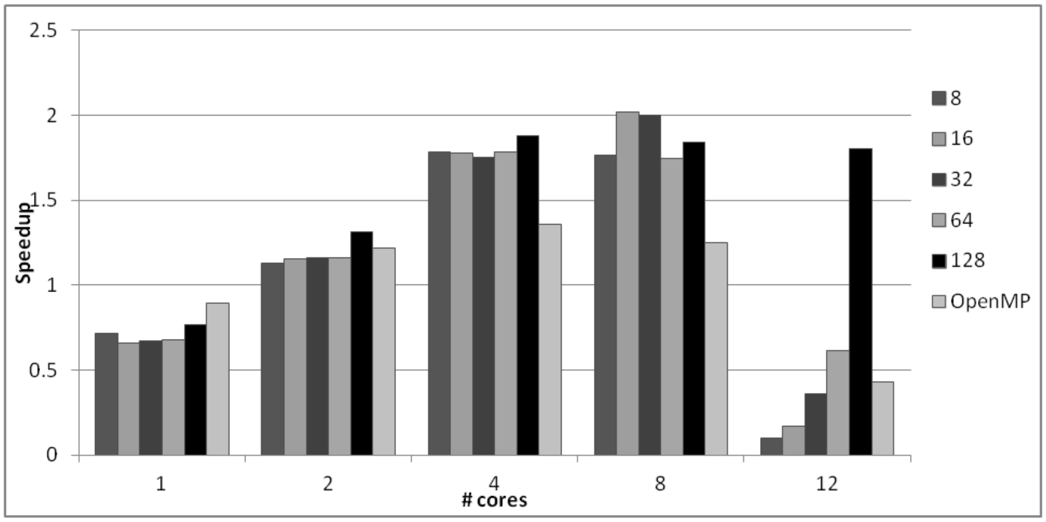


(a) Intel Westmere

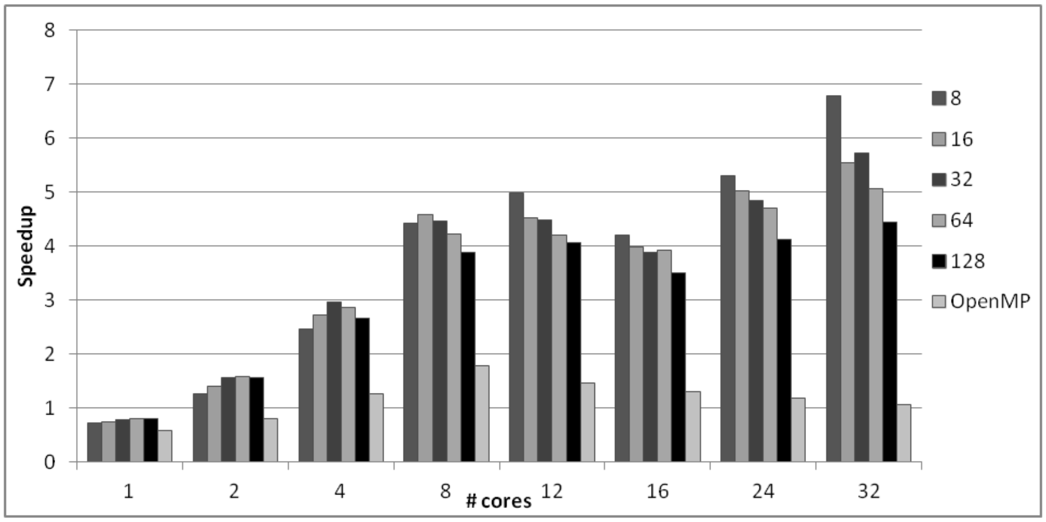


(b) IBM POWER7

Figure 4.12 : FDTD-APML benchmark from the Polybench suite. Results compare DFGL tiling with standard OpenMP parallel version.



(a) Intel Westmere



(b) IBM POWER7

Figure 4.13 : ADI benchmark from the Polybench suite. Results compare DFGL tiling with standard OpenMP parallel version.

4.5 Conclusions

We conclude this chapter with the contributions presented and results obtained. We have shown how DFGR can be mapped onto two runtimes, making use of the programmability feature while obtaining high performance. We have presented our implementation of the DFGR model composed of the translator which reads in the DFGR specification and generates Habanero C code or the ScopLib format for polyhedral transformations, and the DFGR runtime that ensures enforcement of the data-dependencies. We have shown experimental results of our implementation of the DFGR model, which offers good scalability for complex graphs, and competitive performance when compared with the production task parallel language Intel CnC and the OpenMP-DoAcross runtime.

Chapter 5

Elastic tasks

5.1 Motivation

As multicore machines become ubiquitous, *task parallelism* has emerged as a dominant paradigm for parallel programming. Many programming languages and libraries such as Cilk [6], Cilk Plus [7], Intel Threading Building Blocks [8], .Net Task Parallel Library [9], and Habanero-Java [10] support this programming paradigm. In this model, the programmer expresses the logical parallelism by specifying sequential tasks and the dependences between them, and a runtime scheduler is responsible for mapping these tasks on the available processors (workers). Fork-join parallelism is a popular special case of task parallelism with nested divide-and-conquer task structures.

SPMD parallelism [5] is an alternate paradigm for exploiting multicore parallelism that dates back to the earliest parallel computing systems. In this model, a fixed number of worker threads execute a single SPMD region in which certain statements may be redundantly executed by all workers and others are distributed across workers. The tight coupling arising from simultaneous execution of worker threads can be used to efficiently support synchronization primitives such as barriers, which are not supported by the standard task parallel paradigm. There is general agreement that SPMD parallelism can outperform fork-join task parallelism in certain cases, but fork-join parallelism is more general than SPMD parallelism. There has even been work on compiler optimizations to automatically transform fork-join regions of code to SPMD regions for improved performance [65, 66].

However, to the best of our knowledge, there has been no prior work that combines task parallelism and SPMD parallelism in a single adaptive runtime framework.

The goal of a runtime scheduling strategy is to deliver speedups that are as close to optimal as possible. The common scheduling strategies used by these systems are ***work-sharing*** and ***work-stealing*** [67]. A work-sharing scheduler maintains a global pool of tasks that are ready to execute and all processors enqueue and deque ready tasks from this queue. In work-stealing, each processor maintains its own local queue and when the local queue is empty, the worker randomly steals work from victim worker queues. Given a computation with *work* — the running time of the program on 1 worker — W and *span* — the running time of the program on an infinite number of workers — S, both these strategies guarantee a completion time of $O(W/P+S)$ on P workers. This bound implies that they provide linear speedup (speedup of $\Theta(P)$ on P workers) as long as the programs have sufficient parallelism.

In this chapter, we propose *elastic tasks*, a new primitive that helps bridge the gap between task parallelism and SPMD parallelism, can deliver close to optimal speedups is a work-sharing or work-stealing environment, and in practice can take advantage of the application structure to obtain locality benefits or to adapt the parallelism based on the size of the task. Elastic tasks are tasks that are internally parallel and can run on a single worker or expand to take over multiple workers, if available. Elastic tasks need not be data parallel, but they are assumed to have linear scaling with a defined number of processors and to exhibit locality benefits from co-scheduling of their internal parallelism. Workers must be assigned to the task before the task starts executing and must work only on this task until the task finishes executing. Each elastic task u can be assigned anywhere between 1 and $c(u)$ workers, where $c(u)$ is the *capacity* of the task. An application that uses elastic tasks is similar to a task-parallel computation, except that some tasks may be sequential tasks while others may be elastic. Just as in normal task-parallel computations, the programmer specifies the dependences between tasks and the runtime scheduler is responsible for scheduling the computation efficiently on a fixed number of processors.

In this thesis, we extend both the work-sharing and work-stealing strategies to handle task-parallel computation with elastic tasks. In particular, these schedulers must decide how many workers (between 1 and $c(u)$) to assign to an elastic task u before starting its execution. We prove that the work-sharing scheduler completes a computation with work W and span S in $O(W/P+S+E)$ time, where E is the total number of elastic tasks. Similarly, the work-stealing scheduler completes the computation in $O(W/P+S+E\lg P)$ expected time. If the number of elastic tasks is small, then these bounds are comparable to the completion time of computations without elastic tasks. Note that work stealing is a randomized scheduling strategy, therefore our completion time bounds (like all bounds for work stealing schedulers) are probabilistic.

We have implemented two runtime schedulers, using work-sharing and work-stealing, respectively, for elastic tasks, and include experimental results obtained from the work-stealing implementation. Our preliminary results show that elasticity can be beneficial for tasks that are medium or coarse grained, and exhibit internal parallelism with locality. Additionally, when the amount of parallelism varies at runtime for the same section of code, expressing this section as an elastic task enables the parallelism used by this task to adapt at runtime and thus adapt with the granularity changes.

The research contribution in this chapter have been published in [68]. The contributions of this chapter are as follows:

- The elastic task primitive, its definition, properties and requirements.

- Theoretical proof that the elastic task primitive guarantees a completion time of

$O(W/P+S+E)$ time on P workers in a work-sharing runtime and expected time of $O(W/P+S+E\lg P)$ in a work-stealing runtime. The formal proofs were done in collaboration with Kunal Agrawal, which co-authored the published work [68].

- An implementation of a runtime (ElastiJ) with 2 policies (work-sharing and work-stealing) which executes computations with both sequential and elastic tasks. These runtime systems automatically decide the number of workers assigned to each elastic task based on the current runtime conditions.
- Experimental results which indicate that elastic tasks can provide locality benefits for certain computations and provide runtime adaptability.

5.2 Benefits of elastic tasks

In this section, we discuss different use-cases for which elastic tasks may be beneficial. In particular, we focus on the benefits of (1) *elasticity* — the ability of the runtime to dynamically decide the number of workers assigned to a task; and (2) *co-scheduling* — all the work within the elastic task is scheduled at the same time on dedicated workers.

5.2.1 Benefits of elasticity

Before we can analyze the benefits, let us consider some alternatives to creating an elastic task. Given a computation amenable to elastic tasks, we have a few alternatives: (1) We could execute it as a *sequential task* on a single worker; (2) We could create an *inelastic* data parallel task where the programmer specifies the number of workers (say $m(u)$) which must execute this data parallel node u (as in [69]); (3) We could convert it to a task parallel computation by dividing up the computation into independent tasks which need not be co-scheduled; or (4) We could create an elastic task.

Alternative 1 (create a sequential task) can increase the critical path length (span) of the entire computation, thereby decreasing its parallelism. The goal of the processor-oblivious task-parallelism is to enable the programmer to express as much parallelism as possible and allow the scheduler to use any number of available workers. This alternative is not desirable as it may reduce the scalability of the computation.

Alternative 2 (create an inelastic data-parallel task) has the similar disadvantage that it inordinately constrains the scheduler and increases the burden on the programmer. For the case of inelastic tasks where the programmer has to specify precise number of workers that *must* be allocated, it is difficult to guarantee good performance theoretically. For inelastic tasks with large parallelism, if the programmer accurately specifies $m(u)$ to be large, then the scheduler must somehow find all these workers to assign to this inelastic task before it can start execution. If most workers are busy executing other work, the scheduler must either wait for a potentially long time (idle workers for long periods) or it must interrupt the other workers in the middle of their execution leading to large overheads, and violation of the basic assumptions of nonpreemptive multiprocessor task scheduling. If the programmer artificially specifies $m(u)$ to be small to avoid this scenario, then we are potentially wasting the internal parallelism of task u and decreasing the scalability of the overall computation, as in Alternative 1.

Alternative 3 is to convert a parallel task to multiple tasks. This may be cumbersome for certain applications if the different iterations need to communicate — since we must add a control synchronization point for every communication link. This also increases overheads for such computations since barriers across independently-scheduled tasks can be very inefficient. In addition, as we will discuss in the next section, this transformation to task parallelism means that different iterations of this data parallel loop may be executed at very different times, leading to loss of locality.

5.2.2 Benefits of co-scheduling

We now focus on the benefits of co-scheduling provided by elastic tasks. Compared to alternatives 1 and 2, with elastic tasks the programmer is only responsible for providing the capacity — the maximum number of workers that the task can use — not the precise number of workers that the task must use. Therefore, the programmer can simply look at the elastic task locally and evaluate its parallelism without considering the rest of the application. The runtime then adjusts the number of workers allocated to the task based on runtime conditions. We use the remainder of this section to compare elastic tasks with equivalent task parallel computations, as described by the third alternative above.

Cache locality on multicores: Here we present a simple scenario where using elastic tasks for parallel loops is better than converting them into task parallel programs. Consider a situation where we have a parallel loop in which all iterations read some common (shared) data. Using an elastic task forces the scheduler to schedule all the iterations at the same time (potentially on different processors). Therefore, the shared data will be brought into the caches shared cache by multiple workers assigned to the elastic task and all of these iterations get the advantage of improved cache locality. On the other hand, if we had converted this loop into a task parallel program, each iteration or chunk of iterations would be a separate task. The scheduler may have scheduled these iterations at very different times. In this scenarios, since other tasks that access other data may execute between different iterations of the loop, the shared data may have to be brought in multiple times leading to poor cache locality. We will show experimental results that validate this intuition in Section 5.7.

Locality on future target architectures While in this thesis we show the importance of elastic tasks on multicores, we expect elastic tasks to become even more valuable for future extreme scale systems with hundreds of cores per socket, since temporal colocation based on data sharing will be even more important for these platforms. For distributed systems, an elastic task could be “stolen" as a whole, rather than split into sub-tasks. So the effective movement of data would be done once for the whole task, yet it will still be able to execute in parallel within a single node. An elastic task with data-parallel computations can also be automatically transformed into a GPU kernel using existing tools [70, 71, 24] and launched on a GPU. Therefore, elastic tasks can help provide seamless interoperability between CPUs and GPUs. If we have both CPU and GPU implementations of elastic tasks, the runtime can choose to either execute them on CPUs (if GPUs are busy or unavailable) or launch them on GPUs when CPUs are busy. In addition, the adaptability of elastic tasks to task granularity implies that applications can adjust to existing and future GPUs. Demonstrating the benefits of elastic tasks on distributed clusters and GPUs is a subject for future work.

5.3 Model of computation

We now describe the formal model for computation that have both elastic tasks and normal sequential tasks. These are *processor oblivious* computations — the programmer expresses the logical parallelism of the program (using constructs such as async/finish, spawn/sync or parallel for-loops) and the *runtime scheduler* is responsible for dynamically scheduling and executing the computation on P worker threads.

As in much prior work, the computation can be abstractly expressed as a directed acyclic graph called a computation DAG G; nodes are computation kernels and edges are dependences between nodes. A node is *ready* to execute when all of its predecessors have been executed. Without loss of generality, we assume that the maximum out-degree of any node is at most 2. There are two types of nodes: *strands* — sequential chains of instructions and *elastic nodes*, which are internally parallel.

An elastic node u has the following properties: (1) Work $w(u)$ is its execution time on one worker. (2) Capacity $c(u)$ is its maximum internal parallelism. (3) Before an elastic node u can execute, the runtime scheduler must allocate it $1 \leq a(u)$ dedicated worker threads to its execution. Once u starts executing, these $a(u)$ workers can not work on anything else until u completes and no other workers can participate in u's work. (4) We assume that each elastic node provides linear speedup up to $c(u)$ workers and no additional speedup thereafter. That is, if an elastic node is assigned $a(u)$ workers, it completes in $w(u)/a(u)$ time if $a(u) \leq c(u)$ and in $w(u)/c(u)$ time if $a(u) > c(u)$. Therefore, there is no benefit to assigning more than $c(u)$ workers to an elastic node. When it is allocated $c(u)$ workers, we say that the node is *saturated*; otherwise we say that it is *unsaturated*.

As with traditional task parallel processor oblivious computations, we define two parameters for G. *Work* W of a computation is the sum of the computational requirements of all the nodes or the execution time of the computation on 1 processor. *Span* of a computation dag is the weighted length of the longest path through the dag where each node has weight equal to its span. (Note that the span of an elastic u node is $w(u)/c(u)$ since that is the minimum amount of time it takes to finish u after it starts, no matter how many workers are assigned to it.) Span can also be seen as the execution time of the computation on an infinite number of processors. The parallelism of the program is defined as W/S and describes the maximum number of workers the computation can use effectively. Note that the execution time of the computation on P workers is at least $\max\{W/P, S\}$.

5.4 Theoretical guarantees in a work-sharing runtime

In this section, we present our theoretical results when a computation with elastic nodes is executing on P workers using a modified greedy work-sharing scheduler. For computations without elastic nodes, a greedy work-sharing scheduler operates as follows: The scheduler maintains a centralized pool of ready nodes. Every worker has at most one *assigned node* at any time — the node that it is currently executing. When a worker p finishes executing its assigned node, it is assigned another node from the centralized pool of ready nodes. If the pool is empty, then the worker is idle. When a worker enables new ready nodes, these nodes are added to the central pool. Graham [72] proved that this type of greedy work-sharing scheduler guarantees that a program with work W and span S completes in time at most $W/P+S$ on P processors. We mention that our experimental results use a work-stealing scheduler for its better performance, but we believe that the simpler analysis of a work-sharing scheduler provides more intuition about how this algorithm works. For this reason, we include first the work-sharing analysis, followed by the work-stealing one.

For computations with elastic tasks, our work-sharing scheduler is not a pure greedy algorithm since occasionally we allow nodes to not do work even when ready work is available. We show that this new scheduler also has a comparable completion time guarantee with regular work-sharing. The following rules apply:

1. If no elastic node is ready, then the scheduler operates in exactly the same way as the normal greedy work-sharing scheduler described above.

2. An elastic node does not immediately execute when it is assigned to a worker. Instead, it starts waiting. While an elastic node u is ready with capacity $c(u)$ and $a(u) < c(u)$, when a worker p finishes its assigned strand, u is assigned to this worker

and $a(u)$ is incremented by 1. Therefore, an elastic node may be assigned to more than 1 worker.

3. The scheduler keeps track of $wait(u)$ — the total wait time (number of waiting time slots) for elastic node u. That is, on a time step when u is waiting, if $a(u)$ workers are assigned to u, then $wait(u)$ is incremented by $a(u)$ on that time step.

4. An elastic node u starts executing when either (1) $a(u) = c(u)$ — that is $c(u)$ workers are waiting on it and u is saturated; or (2) the total wait time $wait(u) \geq w(u)$.

5. If more than one elastic node is ready, the available workers may choose to wait on any of them or to execute any other strand. However, once a worker is assigned to an elastic node, that worker may not do anything else until that elastic node finishes executing.

Work-sharing scheduler provides linear speedup We prove the following theorem about the work sharing scheduler for computations with elastic nodes.

Theorem 1. *Given a computation graph with work W, span S, and E elastic nodes, the execution time of this computation on P workers using the work-sharing scheduler is $O(W/P+S+E)$.*

PROOF. At any time step, a worker can be working (on a strand or an elastic node), assigned to an elastic node which is waiting, or it can be idle (if the ready pool is empty). The total work steps, over all workers, is W. In addition, since the scheduler keeps track of $wait(u)$ and launches an elastic node as soon as $wait(u) \geq w(u)$, we know that $wait(u) \leq w(u) + P - 1$ since in any one time step, the quantity $wait(u)$ can increase by at most P. Therefore, the total number of waiting steps, over all workers is at most $\sum_{u \text{ is an elastic node}}(w(u) + (P-1)) \leq W + (P-1)E$.

We now argue that the total number of idle time steps is at most $W + (P-1)(S+E)$. Note that idle steps only occur if no elastic node is waiting — otherwise this worker would be assigned to this elastic node rather than idling. There are a few cases:

1. An idle step occurs when either no elastic node is executing or all elastic nodes which are executing are saturated. The total number of such time steps is at most S, since the remaining span reduces by 1 every time such a step occurs. Therefore, the total number of idle steps of this kind is at most $(P-1)S$, since at most $P-1$ workers can be idle on any time step. This argument is similar to the argument used for greedy scheduling for programs without elastic nodes.

2. An idle step occurs when (one or more) elastic nodes are executing and at least one (say u) is unsaturated. In this case, this elastic node waited for $wait(u) \geq w(u)$ worker steps and was launched with $a(u) < c(u)$ workers. Say it waited for $wait-time(u)$ time steps — we know that $wait-time(u) \geq wait(u)/a(u) \geq w(u)/a(u)$. In addition, the total execution time of this node is $w(u)/a(u) < wait-time(u)$. Therefore, the total idle time of any worker p while u is executing is at most $wait-time(u)$. Note that while u was ready and waiting, p was always assigned to some other node, but it was never idle, since a node is only idle if no node is ready. Therefore, the total number of idle steps of this type can be bounded by the sum of work steps (bounded by W) and the waiting steps (bounded by $W + E(P-1)$).

Therefore, the total number of processor steps, which is the sum of work steps, idle steps and waiting steps is at most $O(W + (P-1)(S+E))$. Since we have P workers, there are P steps in each time step. Therefore, the total execution time is $O(W/P + S + E)$. $\square$

5.5 Theoretical guarantees in a work-stealing runtime

In this section, we will analyze the work-stealing scheduler for executing computations with elastic nodes. We first describe the regular work-stealing scheduler for computations without elastic nodes and state its bounds. We then describe the modifications to handle the elastic nodes and prove the runtime bounds.

5.5.1 Work-stealing scheduler for computations without elastic nodes

A work-stealing scheduler for programs without elastic nodes works as follows: A program is executed by P workers each of which has its own private deque (double ended queue) of ready nodes. At any time, a worker p may have a node u ***assigned*** to it — which means that p is in the process of executing u. When a worker finishes u, if u enables one other node v, it may enable 0, 1 or 2 nodes. Note that a node u enables v if u is the last parent of v to finish executing; since the out-degree of any node in G is at most 2, a node can enable at most 2 children. If u enables 1 child v, then v is immediately assigned to p. If it enables two nodes v_1 and v_2, the left child v_1 is assigned to p and the right child v_2 is put at the bottom of p's deque. If u enables no nodes, then the node at bottom of p's deque is assigned to it. If p's deque is empty, then p becomes a thief, selects another worker p_1 as a ***victim*** at random and tries to steal from the top of p_1's deque. If p_1's deque is not empty and p gets a node, then the steal attempt is successful, otherwise p tries to steal again.

Blumofe and Leiserson [67] prove that this randomized work-stealing scheduler finishes a computation with work W and span S in expected time $O(W/P+S)$ time on P worker threads.

5.5.2 Work-stealing scheduler for elastic computations

We can now understand the changes that we propose for work-stealing schedulers when they execute computations with elastic nodes. Now a worker's assigned node may be a strand or an elastic node. When worker p completes an assigned node u, again it enables 0, 1 or 2 nodes and it behaves in exactly the same way as the regular work stealing scheduler. Similarly, if a worker's deque is empty, it becomes a thief. When a strand is assigned to p, p immediately starts executing it. However, the changes due to the elastic nodes affect what happens on steals and what happens when a worker is assigned an elastic node:

1. If an elastic node u is assigned to p, then u does not start executing immediately. Instead, it starts waiting and p is also said to be waiting on u.

2. When p is a thief, it randomly chooses a victim q and tries to steal from q. If q has a waiting elastic node assigned to it (that is q is waiting on an elastic node u), then u is also assigned to p and p also starts waiting on it. At this time $a(u)$ is incremented by 1. Therefore, an elastic node may be assigned to multiple nodes. Otherwise (if q is not waiting on an elastic node) p steals the node at the top of q's deque. Just as the normal work-stealing scheduler, if it succeeds in stealing node u, then u is assigned to p; if the deque is empty then the steal is unsuccessful and p tries again.

3. While u is waiting, its total waiting time (number of waiting time slots) $wait(u)$ is incremented by $a(u)$ in every time step.

4. An elastic node starts executing when either $a(u) = c(u)$ — $c(u)$ workers are waiting on it and it is saturated; or its total wait time $wait(u) \geq w(u)$.

5. When an elastic node finishes executing, the worker which first enabled the elastic node enables its children. All other workers which were assigned to the elastic node

start work stealing. Note that all their deques are empty at this time since they were allocated to the elastic node when they tried to steal.

5.5.3 Analysis of work-stealing scheduler

We will now prove the bound on the completion time of the above work-stealing scheduler.

Theorem 2. *Given a computation graph with E elastic nodes, work W and span S, the expected execution time of this computation on P workers using the work-stealing scheduler is $O(W/P + S + E \lg P)$.*

If we compare this result to the result for computations without elastic nodes, we notice that the additional term is only $E \lg P$. This term is negligible for any computation where the number of elastic nodes is $O(T_1/P \lg P)$ — which implies that most elastic nodes have parallelism $\Omega(P)$ and at most $1/\lg P$ fraction of the work of the computation is contained in elastic nodes.

We mention that the constant factors hidden within the asymptotic bounds are not much larger than those hidden within the standard work-stealing bounds. In particular, an additional term similar to $O(E \lg P)$ would also appear in the standard work-stealing scheduler if we consider the contention on the child counter (which is generally ignored).

In this section, without loss of generality, we assume that each strand is a unit time computation. A longer strand is simply expressed as a chain of unit time strands. This simplifies the analysis. As with work-sharing analysis we separately bound the types of steps that a worker can take at any time step. A worker could be working, waiting on an elastic node or stealing. As in the work-sharing scheduler, the total number of work-steps is at most W; and the total number of waiting steps is at most $W + PE$. Therefore, we need only bound the number of steal steps.

We classify steal attempts in three categories: (1) regular steal attempts occur when no elastic node is waiting and no unsaturated elastic node is executing. (2) waiting steal attempts are those that occur when some elastic node is waiting. (3) irregular steal attempts occur when some unsaturated elastic node is executing and no elastic node is waiting. We will bound the number of steal attempts in these three categories separately.

Intuition for analysis

We adopt a *potential function* argument similar to Arora et al.'s work-stealing analysis [73], henceforth referred to as ABP. In the ABP analysis, each ready node is assigned a potential that decreases geometrically with its distance from the start of the dag. For traditional work stealing, one can prove that most of the potential is in the ready nodes at the top of the deques, as these are the ones that occur earliest in the dag. Therefore, $\Theta(P)$ random steal attempts suffice to process all of these nodes on top of the deques, causing the potential to decrease significantly. Therefore, one can prove that $O(PS)$ steal attempts are sufficient to reduce the potential to 0 in expectation.

The ABP analysis does not directly apply to bounding the number of steal attempts for computations with elastic nodes for the following reason. When an elastic node u becomes ready and is assigned to worker p, it remains assigned until it completes execution. But u may contain most of the potential of the entire computation (particularly if p's deque is empty; in this case u has all of p's potential). Since u cannot be stolen, steals are no longer effective in reducing the potential of the computation until u completes. Therefore, we must use a different argument to bound the steal attempts that occur while u is assigned to p.

Regular steal attempts: These occur when either a worker is assigned an elastic node (the normal ABP argument applies) or any elastic node that is assigned is saturated and is executing. To analyze this case, we use a potential function argument very similar to the ABP argument, but on an augmented DAG in order to account for steal attempts that occur while a saturated elastic node is executing.

Waiting steal attempts: These occur when some elastic node (say u) is waiting — at this time, u is assigned to some worker(s), say p and p'. If any worker q tries to steal from p or p' during this time, then q also starts waiting on u and $a(u)$ increases by 1. Therefore, only a small number of steal attempts (in expectation) can occur before $a(u) = c(u)$ and u becomes saturated and stops waiting. We use this fact to bound the number of waiting steal attempts.

Irregular steal attempts: These occur when no elastic node is waiting and some unsaturated elastic node is executing. The analysis here is similar to the one we used to account for idle steps in the work-sharing scheduler. Since this elastic node started executing without being saturated, it must have waited for at least $w(u)$ time — and during this time, all the workers not assigned to this elastic node were busy doing either work or waiting steal attempts. Therefore, any steal attempts by these workers can be amortized against these other steps.

We now provide the formal analysis.

Bounding regular steal attempts

As mentioned above, for analysis purposes, we augment the computation dag G and create a new dag G'. In G', we add $c(u)$ chains of $w(u)/c(u)$ *dummy nodes* each of unit execution time between each elastic node u and its parent node that enables it. We first state a relatively obvious structural property of G' that states that augmenting the dag does not asymptotically increase its work or span.

Property 1. *The work of G' is $W_{G'} = O(W)$ and its span is $S_{G'} = O(S)$.*

For analysis purposes, we assume that the work-stealing scheduler operates on G' instead of G with the following modifications.

Elastic node u starts executing on $c(u)$ workers (saturated) $w(u)/c(u)$ dummy nodes are placed at the bottom of all of these $c(u)$ workers' deques (we have enough dummy nodes to do so, since we have $c(u)$ chains each of length $w(u)/c(u)$ nodes). When the elastic node u finishes executing, all of its dummy nodes disappear from all the deques of workers assigned to it. Note that there are never any nodes below dummy nodes on any deque, since no other nodes are enabled by any of these $c(u)$ workers while they are executing the elastic node u.

p tries to steal from q who has only dummy nodes Since a dummy node is at the top of q's deque, p simply steals this top dummy node and this dummy node immediately disappears and q tries to (randomly) steal again. Note that q's deque would otherwise be empty and p's steal attempt would be unsuccessful. Therefore, this operation simply converts some unsuccessful steals into successful ones for accounting purposes.

Note that these rules for the execution of G' do not change the actual execution of the scheduler at all on G; they are purely used for making the analysis easier.

Lemma 3. *While a saturated elastic kernel u is executing, none of workers assigned to u ever have an empty deque.*

PROOF. A saturated elastic node executes for exactly $w(u)/c(u)$ time. On each time step, there is at most one successful steal attempt from each worker. Therefore, there can be at most $w(u)/c(u)$ steal attempts from each of the workers executing saturated elastic node u while it is executing. Since we placed $w(u)/c(u)$ dummy nodes on each of these workers' deques, their deques are never empty while u is executing. □

We will define a potential function similar to one defined in the ABP paper, except that we will use the augmented computation dag G' to define this potential. Each node in G' has ***depth*** $d(u)$ and ***weight*** $w(u) = S_{G'} - d(u)$. The weights are always positive.

Definition 1. *The* ***potential*** Φ_u *of a node* u *is* $3^{2w(u)-1}$ *if* u *is assigned, and* $3^{2w(u)}$ *if* u *is ready.*

The ***potential*** of the computation is the sum of potentials of all (ready or assigned) nodes $u \in G'$.

The following structural lemmas follow in a straightforward manner from the arguments used throughout the ABP paper [73], so we state them without proof here. * Instead of giving the full proof, we explain the idea about why these lemmas apply and why the augmented dag G' helps with that. As we mentioned above, if an elastic node u is assigned to p and p's deque is otherwise empty, then u contains all of the potential of p and therefore, the steal attempts with p as the victim no longer appropriately decrease the potential. However, note that the augmented dag G avoids this situation, at least for saturated elastic nodes. Due to Lemma 3, a worker p's deque is never empty while it is executing a saturated elastic node u. Therefore, any *regular steal attempts* (since they only occur when either there is no elastic node assigned or all assigned elastic nodes are saturated) from p's deque still decrease the potential of the augmented dag G'.

Lemma 4. *The initial potential is* $3^{S'_G}$ *and it never increases during the computation.* □

* [73] does not explicitly capture these three lemmas as claims in their paper — some of their proof is captured by "Lemma 8" and "Theorem 9" of [73], but the rest falls to interproof discussion within the paper .

Lemma 5. *Let $\Phi(t)$ denote the potential at time t. After $2P$ subsequent regular steal attempts, the potential is at most $\Phi(t)/4$ with probability at least $1/4$.* □

Lemma 6. *Suppose every "round" decreases its potential by a constant factor with at least a constant probability. Then the computation completes after $O(S_{G'})$ rounds in expectation.*

□

We can now use these lemmas to bound the total number of regular steal attempts.

Lemma 7. *The total number of regular steal attempts is $O(PS + P\lg(1/\varepsilon))$ in expectation.*

PROOF. We say that a round consists of at least $2P$ consecutive regular steal attempts. Using Lemmas 4, 5, and 6, we can conclude that the total number of regular steal attempts is $O(PS'_G + P\lg)$ in expectation. In addition, the from Property 1, we know that $S'_G = O(S)$. □

Bounding waiting steal attempts

A waiting steal attempt occurs when some worker p is waiting on an elastic node u.

Lemma 8. *There are $O(P\min\{\lg c(u), \lg P\})$ steal attempts in expectation while a particular elastic node u is waiting. Therefore, the expected number of waiting steal attempts over the entire computation is $O(PE\lg P)$.*

PROOF. A waiting elastic node starts executing either when it has waited for long enough or when it is saturated. We will argue that after $O(P\min\{\lg c(u), P\})$ steal attempts in expectation, it is saturated. If it starts executing before that by exceeding its wait time, then the lemma is trivially true.

We say that a waiting steal attempt succeeds if some worker p steals from another worker q who was waiting on some elastic node u and as a result p also starts waiting on u.

Note that this is a different definition of success than we generally use, but it is useful for this section.

We will divide the steal attempts that occur while node u is waiting into phases. The first phase $R_u(1)$ starts when the node u starts waiting. Phase i starts when the previous phase ends and ends after either an additional worker is assigned to u (there is a successful waiting steal into u). It is obvious that there are at most $\min\{c(u), P\}$ phases while u is waiting. Now we argue that during the phase $R_u(i)$, the probability that a waiting steal is successful and an additional worker starts waiting on u is at least i/P. During the first phase, one worker is waiting on u. Therefore, during this phase, the probability of a waiting steal succeeding is $1/P$. After the first phase, 2 workers are waiting on u and therefore, the probability of a waiting steal succeeding is $2/P$. Similarly for phase j. Therefore, while u is waiting, there are $\sum_{j=1}^{\min\{c(u),P\}} P/j \leq P \min\{\lg c(u), P\}$ waiting steal attempts in expectation (linearity of expectation).

We can sum over all elastic nodes and use linearity of expectation to get the bound on the total number of waiting steals over the entire computation. □

Bounding irregular steal attempts

We now bound the irregular steal attempts. The idea is very similar to the one used for bounding idle steps in a work-sharing scheduler.

Lemma 9. *The total number of irregular steal attempts is at most $O(W + PE\lg P)$.*

PROOF. irregular steal attempts occur when the some unsaturated elastic node u is executing and no elastic node is waiting. In this case, this elastic u node waited for $wait(u) \geq w(u)$ worker steps and was launched with $a(u) < c(u)$ workers. Say it waited for $wait - time(u)$ time steps — we know that $wait - time(u) \geq wait(u)/a(u) \geq w(u)/a(u)$. In addition, the total execution time of this node is $w(u)/a(u) < wait - time(u)$. Therefore, the total number of steal attempts by some worker p while u is executing is at most $wait - time(u)$. Note that while u was ready, p was either working or performing waiting steals; it could not perform regular or irregular steal attempts since they do not occur while some elastic node u is waiting. Therefore, the steals while u is executing can be bounded by the work or steal attempts that occurred while u was waiting. Summing over all elastic nodes u gives us the desired result. □

Proof of Theorem 2: Combining Lemmas 7, 8 and 9 tells us that the total number of steal attempts over the entire computation is $O(W + PS + PE \lg P)$. In addition, we know that the total number of work steps and waiting steps is at most $O(W + PE)$. Therefore, if we add all types of steps and divide by P (since we have P worker threads and each take 1 step per unit time), we get the overall expected running time of $O(W/P + S + E \lg P)$. □

5.5.4 Practical bound for work-stealing scheduler

We have proven that the work-stealing scheduler with elastic tasks will have an overall expected running time of $O(W/P + S + E \lg P)$, which adds an additional factor to the traditional work-stealing schedulers. We want to look now at the practical aspect of this bound and how the additional constant factor will affect performance. So we focus on how the factor $O(E \lg P)$ would factor into running time. The assumption that we make is that the number of elastic tasks should not be bounded, as they offer both expressiveness and as we will show in practice additional performance benefits. In addition, we assume that future architectures will encompass thousands and tens of thousands of cores, so the value of P is bound to become large. The conclusion that we draw from these observations is that the additional factor we introduce can become a performance bottleneck. Therefore we look into creating a tighter bound for the work-stealing runtime.

For practical purposes, we assume that the capacity of elastic tasks is bounded to a constant number K, where $K < P$ such that the overhead of waiting for up to K workers to join a task is covered by the benefits of using elastic tasks. We assume the existence of a compiler that would automatically enable the code transformation of the user definitions, such that elastic tasks with capacities greater than K will be split into multiple elastic tasks. Based on this conditions, the new work-stealing bound becomes $O(W/P + S + E \lg K)$. The new bound no longer has the factor that increases with the number of processors; this was instead replaced by a constant factor. Adding to this the original premise of a work-stealing scheduler, i.e. the program has parallel slackness, then the new bound is on-par with regular work-stealing schedulers. In the next section we will talk about how we took the idea of a practical approach further in order to create a system that offers competitive performance.

5.6 Implementation details

We created ElastiJ, a system that supports the creation and execution of elastic tasks. Our system extends the Habanero-Java library (HJlib) [74] with two new runtimes — one which uses the work-sharing scheduler and another that uses work-stealing scheduler; we include a JDK8 implementation and a JDK7 compatible one. We present results using the work-stealing runtime, due to its better performance, as shown by previous works [6, 75].

5.6.1 API

Elastic tasks are created using *asyncElastic*, which is a regular spawn call (*async* in HJlib) with two additional parameters:

- *work* - $w(u)$ The total amount of work performed by the task, described in estimated time in milliseconds, if the task were to run sequentially.

- *capacity* - $c(u)$ The amount of parallelism available, described by the maximum number of workers that can join the task.

The capacity of a task is simply the approximation of its average parallelism — for any task (even for non-data parallel tasks), we can simply assume that it is the task's work ($w(u)$) divided by its critical path length (or span). The work $w(u)$ is the total running time of the task on 1 processor, and many tools exist to measure parallelism. For instance, for CilkPlus programs, Cilkview measures the parallelism and this can be used as $c(u)$.

Two additional optional parameters: *($b(u)$, $e(u)$)* can be used to describe the computation much like describing the iteration space for a loop with begin and end bounds.. The runtime divides this range and assigns non-overlapping ranges to the workers executing the elastic node, similar to OpenMP's loop static scheduling [18], except that the number of workers is dynamic, based on the number of workers are allocated to elastic node when it starts execution. The body of the *asyncElastic* construct will receive two parameters, *(start, stop)*, which are set automatically by the runtime system when the task begins executing. Listing 5.6.1 shows a sample of code using JDK8 lambdas and Listing 5.6.1 shows an example using anonymous inner classes in JDK7.

For computations that are not data-parallel these values can have a different meaning or simply not be used in the body of the task. For example, we show in Section 5.7.2 an implementation of Quicksort, where the partition phase - a computation that is not data parallel - is implemented using an elastic task.

Listing 5.1: Code sample using JDK8.

```
asyncSimt(work, capacity, begin, end, (start, stop) -> {
  for (int i = start; i <= stop; i++) {
    doWork(i);
  }
});
```

Listing 5.2: Code sample using JDK7.

```
asyncSimt(work, capacity, begin, end,
          new HjSuspendingProcedureInt2D() {
  @Override
  public void apply(final int start, final int stop)
                        throws SuspendableException {
    for (int i = start; i <= stop; i++) {
      doWork(i);
    }
  }
});
```

5.6.2 Runtime implementation

ElastiJ implements both a work-sharing and a work-stealing runtime and the choice is made by setting a property either programmatically or as a JVM argument. Both runtimes use the same API, so applications can switch seamlessly between the two. We give an overview of the work-stealing runtime implementation, with a focus on the changes necessary to regular work-stealing runtimes, in order to support elastic tasks. The work-sharing runtime requires a similar approach, and we omit its description for brevity. Our implementation creates a from-scratch work-stealing runtime in Java, due to the need for explicit thread management; in particular, extensions of existing implementations, such as Java's ForkJoin do not allow such control.

The runtime executes in the same manner as is described in Section 5.5. The runtime starts the default number of workers (matching the machine number of cores), where each performs regular work-stealing. Each has its own private deque of tasks, and a flag indicating if it is currently handling an elastic task. We create a new kind of task u called an *ElasticActivity* which allows multiple workers to execute it, and contains the information provided by the user in the API call. When a elastic node u is at the bottom of some worker's deque or is stolen by some worker, then this worker does not immediately start executing it — instead it starts waiting on it. During the wait time τ, other workers have the chance to join the elastic task, each worker p keeping track of their own wait time τ_p. At any one point, there are some number of workers $a(u) \leq c(u)$ that have joined the elastic task u. If $\sum_{p=1}^{a(u)-1} \tau_p \geq \tau$, then the task begins to execute with $a(u)$ workers. If $a(u) = c(u)$ before the total wait time is reached, we say the task became saturated and it will start straight away. Each worker joining an ElasticActivity creates their own clone of the original activity, and will be assigned a fraction of the task's total work, once the waiting period is over

We now describe the mechanism used in practise by threads for waiting on an elastic task. The simplest approach is as follows: the first thread to join starts to wait $w(u)$, the second thread wakes up the first and both of them continue to wait for half of the remaining time, and so on. This approach causes a lot of overhead due to the several sleep and notify calls, in particular in the case when many threads want to join the task at once. A better approach is for the first thread to wait $w(u)$ and store his wait start time. The second thread uses this wait time to compute how much the first thread already waited, and waits half of the remaining time, but not before storing the time he starts the wait. The process goes on until the last thread either saturates the task or has waited the remaining fraction of time and it can wake up all threads. This second approach has the advantage that each thread only goes to sleep once and is woken up once when it can go ahead and start the work. However this approach also experiences a lot of overhead because of context-switching when the method wait is called. Since the waiting threads do not perform any work, we looked into a third approach involving a bounded busy-wait for each thread joining an elastic task. We have used the "perf" tool to examine context switches for these last two implementations and in practice noticed an order of magnitude less context switches when doing busy-waiting rather than wait/notify, and consequently a more performant runtime. It is this third approach that we use in our experiments. In addition, the runtime uses as the total wait time a fraction of the full estimated work given to the task, in order to account for the constant factor in the theoretical proof, and thus offer competitive performance.

The *asyncElastic* construct we propose also includes an implicit phaser [27] for all workers that join the task. As shown in previous work [66], the use of phasers instead of barriers can provide additional performance benefits. Task creation can be expensive, and by using fewer barriers, we remove the overhead of creating a new set of tasks at each barrier point [27].

5.7 Experimental results

In this section, we use a series of benchmarks to demonstrate how elastic tasks perform in different scenarios. In particular, we first assess if elastic tasks do have the potential to provide better locality using an synthetic micro-benchmark. We use Quicksort and FFT algorithms to demonstrate that elastic tasks are easily integrated into task-parallel recursive programs and provide easy adaptability between using normal task-parallel constructs and the task parallelism of elastic tasks. Finally, we evaluate the performance of ElastiJ using a set of benchmarks from the IMSuite [76] set. These benchmarks use simple forall constructs, and do not exhibit any nesting. We aim to use these to show little or no overhead from using elastic tasks, as well as how sensitive is the system to the theoretical assumptions and API parameters.

The performance results were obtained on two platforms:

1. IBM POWER7: runs performed on a node containing four eight-core POWER7 chips running at 3.86GHz. Each core has L1 cache split into separate data and instruction caches, and each being 32 KB in size for each core. The L2 cache is 256 KB in size and the L3 cache is 4 MB in size for each chip. The system has 256GB RAM per node, or 8GB per core.

   Java version 1.7.0_21: Java HotSpot(TM) 64-Bit Server VM (build 23.21-b01, mixed mode).

2. Intel Westmere: runs performed on a single Westmere node with 12 processor cores per node (Intel(R) Xeon(R) CPU X5660) running at 2.83 GHz with 48 GB of RAM per node (4 GB per core).

   Java version 1.7.0: IBM J9 VM (build 2.6, JRE 1.7.0 Linux ppc64-64 20110810_88604 (JIT enabled, AOT enabled)

5.7.1 Benefit from locality

In Section 5.2, we described scenarios in which the use of elastic tasks could be beneficial for temporal locality. In this section, we evaluate when the elastic node construct provides locality benefits. We create a synthetic fork-join style benchmark with the following structure:

- The benchmark spawns n tasks of type ua in a single finish scope, where n is a multiple of the number of cores P.

- Each *ua* task accesses the same M elements of a vector A.
- Each *ua* task spawn P *ub* tasks, and each *ub* task accesses the same M elements of a different vector B.

The capacity $c(u)$ of each elastic task is P, the maximum number of machine cores. All experiments in this section were obtained on the POWER7 for which P=32. The program accepts a parameter to set the fraction α of *ua* tasks that are elastic tasks, and creates $n \times \alpha$ elastic tasks and $(1-\alpha) \times n$ regular tasks (strands). The program spawns α elastic tasks using $asyncElastic$ (marked as "Elastic" in the graphs), and $(1-\alpha)$ simulated elastic tasks using *async* (marked as "Async"). This simulation essentially means creating P regular tasks (for each elastic task), which are all in parallel with each other.

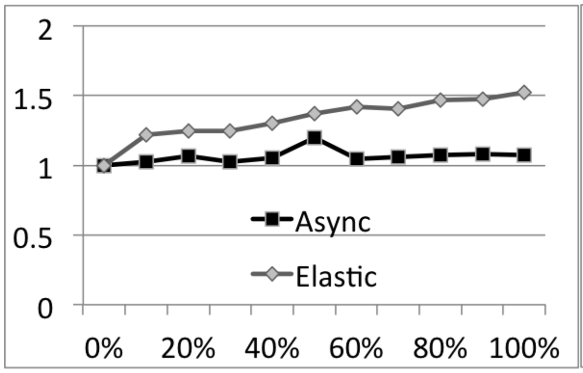
The setup described above enables us to compare regular task creation with the creation of elastic tasks for different data footprint scenarios obtained by changing the value of M.

We expect a locality benefit due to the co-scheduling of the elastic tasks since all the workers executing the elastic task access the same array A. Therefore, it is likely that A will remain in the shared cache while this elastic task is executing. On the other hand, when we convert this elastic task into a normal task-parallel for-loop with P strands, these P strands may execute far apart in time — therefore, some of the *ub* tasks may pollute the cache in the meantime.

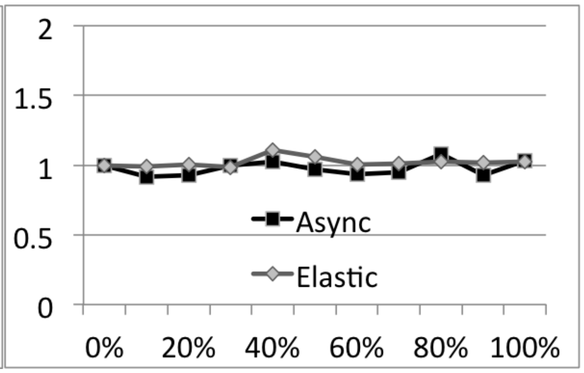
We run experiments on the POWER7, by varying the fraction of elastic tasks: $0\% \leq \alpha \leq 100\%$ (0% means all tasks are regular tasks (strands), while 100% means all tasks are elastic). All experiments report the average of the 20 best iterations over 3 runs, each with 30 iterations; this is to remove the JVM warm-up and variability between runs [77].

The first experiment targets the L2 cache. We set the size of the arrays to $M = 64000$; with A being an integer of arrays this adds to 256KB of data. The results for this run are shown in Figure 5.1a. We see that for the variants using regular tasks (*Async*) the time is essentially constant between 0-100%, which shows that the adjustment in task granularity do not affect the running time (i.e. there is already enough parallelism to begin with, splitting the tasks could only incur overhead by making the tasks too fine-grained, yet this is not the case either: both work just as well). For the elastic tasks however we notice the time increases as more tasks are set as elastic. Since the data was still rather small and the runs were short (70-100ms), this lead us to believe that the overhead of the wait time is affecting the results. Before benchmarking, we estimate the work with a sequential run of a single task, and extrapolate to estimate the total work; but for such short runs this was far from precise. As such, we introduce a slightly larger computation (32 more additions per iteration inside each task), leading to runs of 2-3s. The results are shown in Figure 5.1b. We notice that there is no longer a time increase for elastic tasks, and thus claim that in practical cases the use of elastic tasks should not hurt performance. We repeated the above experiments for bigger loads and a larger number of tasks for the same M, as well as for $M = 72000$ which amounts to the size of the L1+L2 caches. The results were similar. We see that as long as a task runs for at least 1 ms, it can be expressed as an elastic task with little or no overhead (note that originally we had 800 tasks, all running in $<$80ms).

The second experiment involves larger data and targets the L3 cache. We set the size of the arrays to $M = 1.000.000$; with A being an integer of arrays this adds to 4MB of data. Note that we ignore the L1 and L2 caches, under the claim that the size chosen is large enough to ensure at least some evictions for, even if not the data in its entirety. The experimental results are in Figure 5.2a. show elastic tasks make a difference when M is large enough to cause the *ub* tasks to evict part of the A data from L3 cache. We notice that for elastic tasks the execution time remains essentially constant, while for regular tasks the performance is degrading as we increase their number. As noted in the previous experiment,



(a) Very small tasks: M iterations, M array accesses, M additions.



(b) Medium tasks: M iterations, M array accesses, M*32 additions.

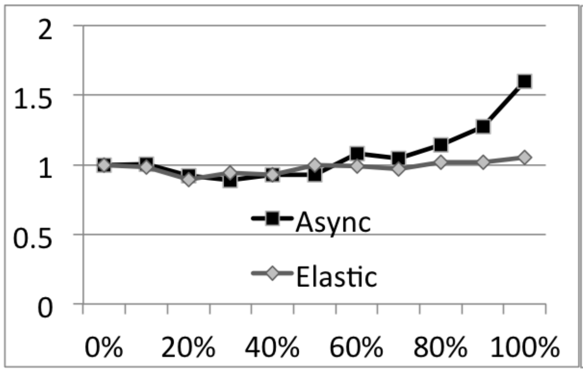
Figure 5.1 : Comparing Elastic with Async. The Y axis shows the times divided by the 0% time. The X axis is the fraction of tasks that are elastic and simulated elastic for the Elastic and Async lines respectively. Data accessed is 256KB, the size of the L2 cache (A is an array of integers, $M = 64000$). The time for 0% is the same for both curves: (a) 76ms and (b) 2.2s. Figure (b) shows increased load to account for the overhead of waiting.

the task granularity was not a problem (task size is kept identical as in the experiments from Figure 5.1a), so the action of splitting the task cannot be the cause. We therefore go back to what we inferred and assume the data accessed by tasks DA is being invalidated by tasks DB. We use the $perf$ tool on the POWER7 to confirm this. Figure 5.2b plots the cache misses obtained by the $perf$ tool. We see that for the number of cache misses becomes up to 1.7$\times$ when using regular tasks as opposed to elastic tasks.

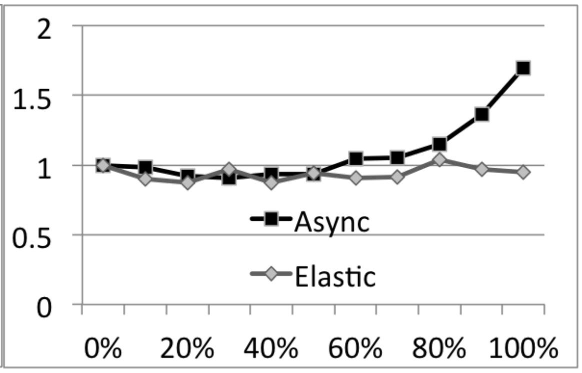
We conclude that the use of elastic tasks should be used in a setting where their granularity amortizes the overhead of setting them up and that they can offer performance benefits due to temporal locality.

5.7.2 Adaptability

We argue that elastic tasks provide a benefit that they integrate seamlessly with normal task parallelism which is suited to recursive algorithms. The quicksort algorithm is a recursive divide-and-conquer algorithm and can be easily expressed as a task-parallel program using async-finish or spawn-sync constructs. However, the divide step of this divide-



(a) The times for 0% are approximately 1.4s.



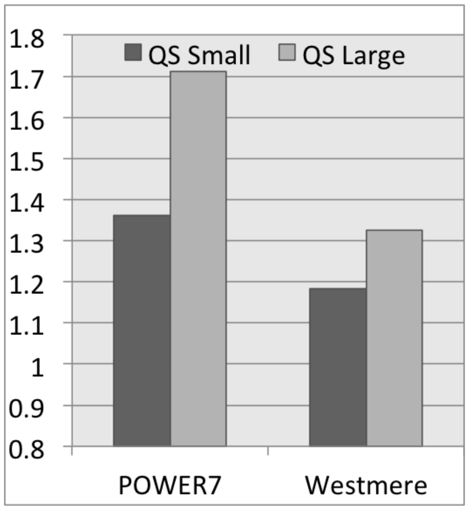
(b) The cache misses for 0% are 7.3billion.

Figure 5.2 : Comparing FJ-Elast with FJ-Async, where data accessed is 4M, the size of the L3 cache ($M = 1.000.000$, A is an array of integers). The X axis is the fraction of tasks that are elastic and simulated elastic for the Elastic and Async lines respectively. The Y axis shows normalized values based on the 0% value.

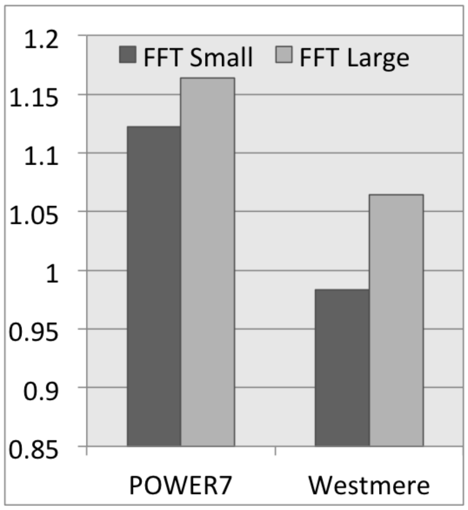
and-conquer program consists of a partition step. While partition can be expressed as a divide-and-conquer algorithm, that is a cumbersome implementation with a large overhead — most parallel implementations of quicksort simply use a sequential partition algorithm.

In this section, we evaluate the benefit of implementing the partition algorithm as an elastic task [78]. Note that the partition step is called at every level of recursion of the quicksort algorithm. The advantage of using elastic tasks for expressing it is that the number of workers assigned to this partition step is decided at runtime. At the shallow levels of the recursion tree where there is not much other work being done, the partition will be automatically assigned more workers. At deeper levels of the tree, when there is already enough other parallel work available, there won't be too many steals and partition will execute mostly sequentially (since workers only join an elastic node when they steal). Therefore, elastic tasks provide automatic adaptability without any in-code instrumentation.

In Figure 5.3a, we compare a parallel implementation for quicksort with a parallel partition phase implemented with N asyncs, with N=# of machine cores, or as an elastic task. We present the results normalized on the aync runs, to make clear the gain percentage rather



(a) Quicksort: Small dataset is 10mil, large dataset is 100mil.



(b) FFT: Small dataset is 2ˆ22, large dataset is 2ˆ23.

Figure 5.3 : Elastic task runs normalized over async finish runs. Values > 1 mean elastic runs are better.

than times. In Figure 5.3b, we compare two implementations of FFT, where the recombine phase is implemented either as a binary tree spawn using asyncs or as an elastic task. For both quicksort and FFT we used a rough estimate for the work: $array\_length/a\_constant$; additional opportunities exist for auto-tuning the work based on the input, recursion depth, etc. We present results for both benchmarks on the POWER7 and the Westmere, by using the 30 best iterations out of 90 iteration, using 3 JVM invocations. For both architectures we get on average 10% gain from using elastic tasks in the parallel partition. We also note that the gains are larger when the datasizes increase, i.e. where the use of elasticity offers freedom in scheduling. We believe this trend will be beneficial with future large scale applications, but additional work is required to ensure this.

5.7.3 Sensitivity analysis

We use 8 benchmarks from the IMSuite Benchmark Suite [76] to compare regular task spawning with the creation of elastic tasks. All benchmarks use the forAll construct which we convert to elastic tasks. In these benchmarks, there is at most a single forAll loop active at a time, with no additional tasks running simulataneously with it, and the data accessed by any loop is small enough and it will always fit in the lower cache levels. This means that this is the worst case scenario for using elastic tasks since it does not offer any opportunity for benefits from elasticity. Our goal in using these benchmarks is to demonstrate low overhead for elasticity, by showing on-par performance compared to regular task spawning (async/ finish primitives). We discover however that with proper tuning elastic tasks can offer benefits even in this scenario due to their resilience to the theoretical assumptions and to the elastic task parameters. We create a sensitivity analysis to the assumption that elastic task u should exhibit linear scaling with $c(u)$ and to the variations on the estimated work $w(u)$ that the user provides.

The benchmarks we use are the following:

- *BFS Bellman Ford (Bell)*: computes the distance of every node from a given root. The benchmark simulates message passing, starting with the root node passing its neighbors its distance to them.
- *BFS Dijkstra (Dijk)*: creates a tree starting from a given root of a network. Each time-step t, the nodes at distance t from the root are added to the tree.
- *Byzantine Agreement (Byz)*: obtain a consensus vote among all the nodes in a network with hazardous nodes.
- *k-Committee (Kcom)*: partitions the nodes in a network into committees of size at most k (k=6), using some nodes designated as initial committee leaders.
- *General Leader Election (Leaddp)*: finds the leader for a set of nodes organized in any general network. It computes the leader and the diameter of the network.

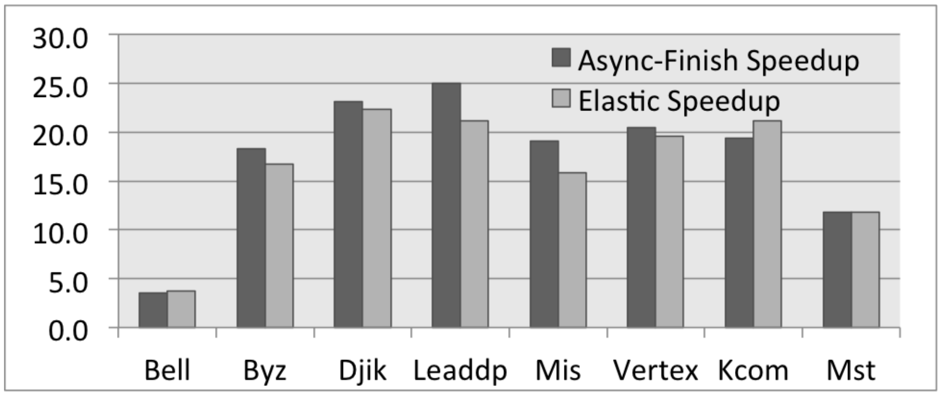


Figure 5.4 : Speedups for IMSuite benchmarks on POWER7. Geomean of elastic times normalized over async finish runs (>1 if elastic runs are better) is 0.953

- *Minimum Spanning Tree (Mst)*: generates a minimum spanning tree for a weighted graph. The minimum weighted edge is used to link together parts of the graph, starting from individual nodes.

- *Maximum Independent Set (Mis)*: generates a maximal independent set of nodes using a randomized algorithm.

- *Vertex Coloring (Vertex)*: colors the nodes of a tree with three colors, first by coloring the whole tree with six colors and then using a shift down operation to reduce the coloring to three colors.

All benchmarks use an input size of 512 nodes, and the input files from the IMSuite website.

Figures 5.4 and 5.5 give the speedups for 8 benchmarks from the IMSuite on the Westmere and POWER7 platforms. We use an average over the 50 best runs out of 150 runs over 3 JVM invocations, and compute speed-ups compared with a pure sequential implementation.

We notice that the performance is comparable with regular task spawning, but that this varies between platforms. On the Westmere platform we notice better performance using elastic tasks for many of the benchmarks. We tested how these benchmarks scale (Figure

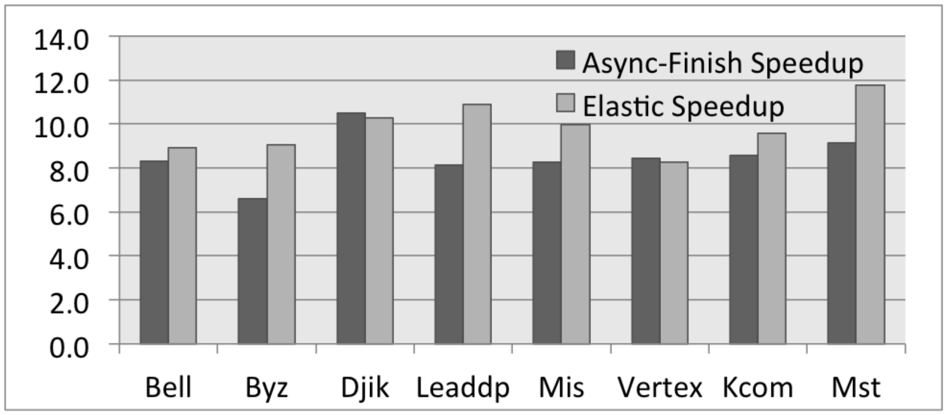


Figure 5.5 : Speedups for IMSuite benchmarks on Westmere. Geomean of elastic times normalized over async finish runs (>1 if elastic runs are better) is 1.160

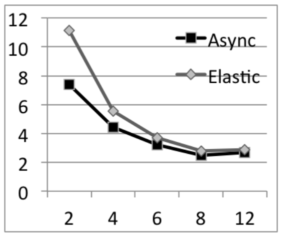
5.6 and found that all of them fail to scale past 8 cores on this platform. As a results, even though for 2 cores the Async implementation runs faster, on 12 cores it is the Elastic implementation which performs best. This shows that elastic tasks can be used to tune applications even when the theoretical guarantee of linear scaling is violated.

Furthermore, we looked into how sensitive the applications are to the API parameters. We look into sensitivity with the estimated work $w(u)$, as it doesn't make sense to restrict the capacity of a task (this can only limit the parallelism, something that can be done using $w(u)$ too). We ran all benchmarks with the same $w(u)$ used in the previous experiments – W–, as well as values smaller and larger than W(W/10, W/5, W*5, W*10, W*20, W*100), in order to determine how these benchmarks behave when the work estimation is not perfect. We show these results in Figure 5.7.

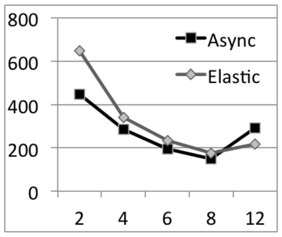
Let us make some important observations on these results:

- First of all, a glance at these results shows that the running time variation when Wis varied is not large and that in most cases the added overhead is at most 10%. The few cases where the percentage-wise variation is larger, are quantitatively very small in terms of actual time (see Figure 5.8).

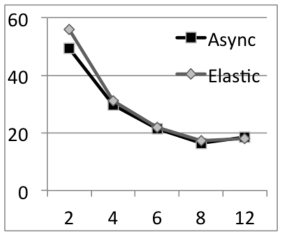
- We see that for almost all benchmarks the times are large when the estimated work



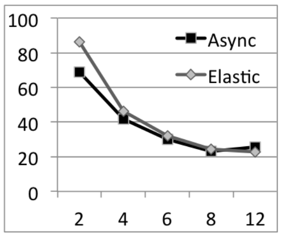
(a) Bel



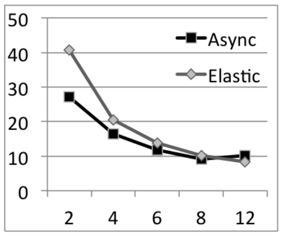
(b) Byz



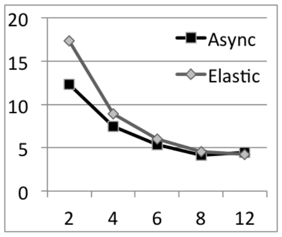
(c) Dji



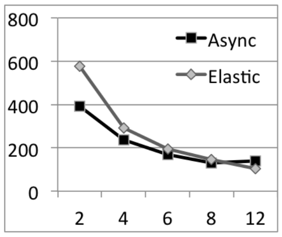
(d) Kcom



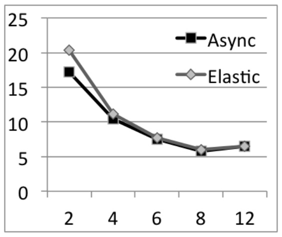
(e) Leaddp



(f) Mis



(g) Mst



(h) Vertex

Figure 5.6 : Scaling results. Y axis is Time(s), X axis is the number of cores.

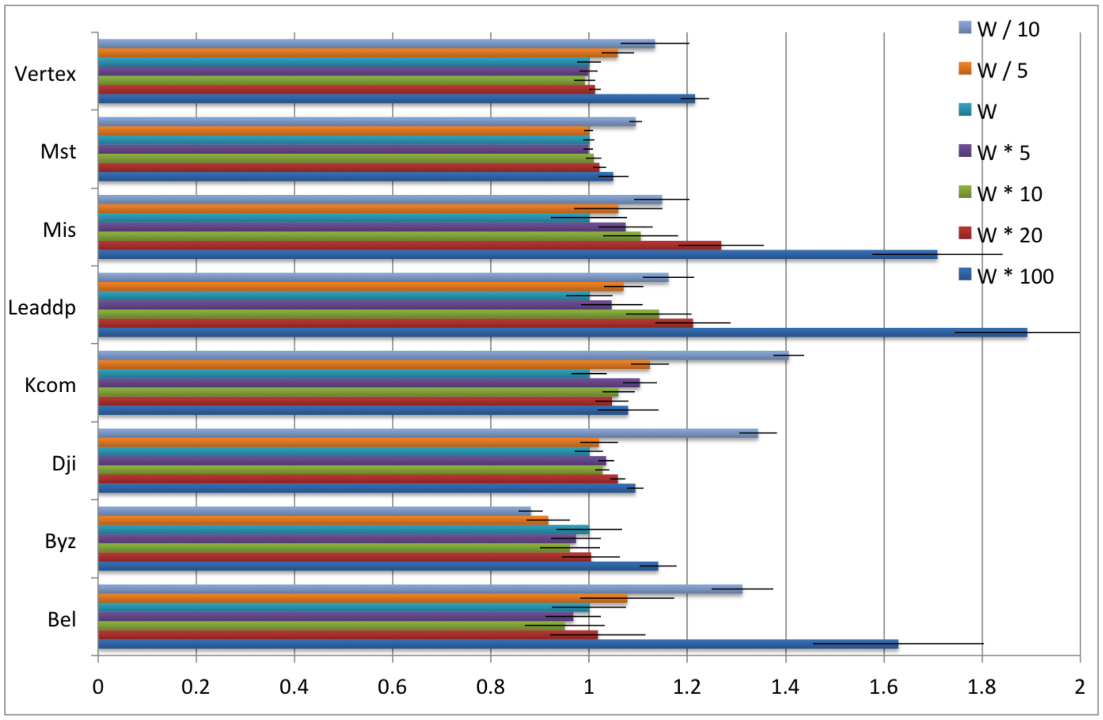


Figure 5.7 : Sensitivity results on Westmere. X axis is time normalized on W.

is too small (e.g.W/10). This is expected, since a small estimation means not enough wait time for threads to join the task, so we are wasting the internal parallelism of the task and decreasing the scalability of the overall computation.

- On the other side of the spectrum, we see that if we have a large estimated work, we delay the computation long enough for $c(u)$ to join. Considering the earlier scalability results this is not the ideal case. So we have two combined causes for the performance degradation: a longer wait time for threads to join the task and a larger running time on 12 cores.

- Benchmarks Bel, Leaddp and Mis seem to exhibit a larger variation, especially when the estimated work is high. Note however that these are short running tasks and Figure 5.7 plots the normalized times. If we look at the same results when we plot Time on the X axis instead (Figure 5.8), we see that the time variation is quite small.

- The Byz benchmark is the longest running and also the one which exhibits the worst performance loss past 8 cores. The results we see in Figure 5.7 tell us that the estimated time Wis not optimal, but that a smaller value, such as W/10 will yield better performance. It is obvious that for any long running task, the estimated time will also be large, which will allow more threads to join; considering the scaling of Byz, this is sub-optimal.

- We also note that the largest errors are obtained for the largest wait times. This is easily explained by the randomized way that threads join an elastic task, which can vary between runs.

Our intuition is that the use of elastic tasks can be sensitive to the number of threads that join. On the POWER7, 32 threads need to join; on the Westmere, only 12. This can lead to

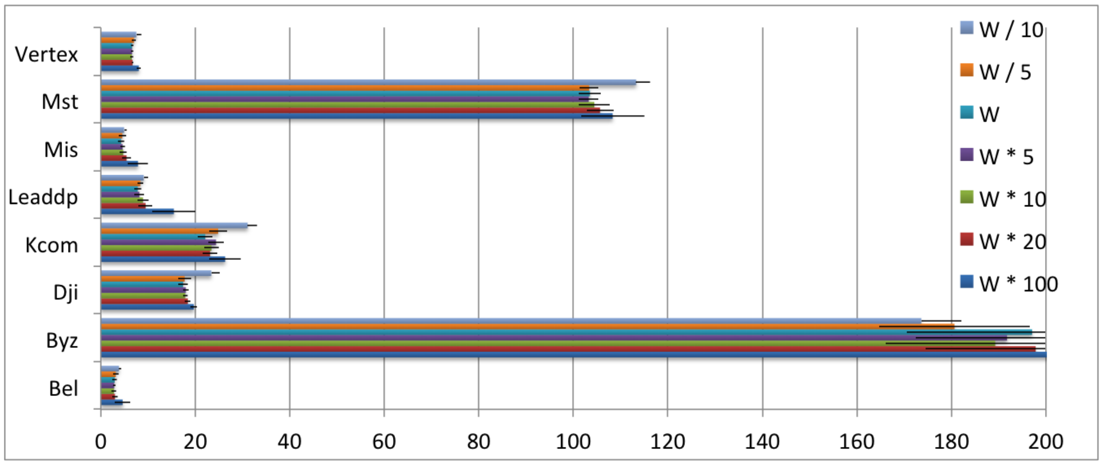


Figure 5.8 : Sensitivity results on Westmere. X axis is time.

a performance penalty on a large number of cores, due to the additional wait time. This can be resolved with an accurate estimation and using a compiler transformation which splits an elastic task into multiple tasks using a threshold for the maximum amount of threads a task can wait on. This is saying that the additional constant factor $E \lg P$ in Theorem 2 can become detrimental when P is very large.

We conclude our overall results with the following observations: (a) elastic tasks give a common API for expressing task and SPMD parallelism and are straightforward and easy to use; (b) they can benefit from locality due to their coscheduling property; (c) they can be used to adapt the degree of parallelism for recursive benchmarks; (d) they can offer comparable performance with forall constructs, and, even though they can also incur a performance penalty, they are fairly resilient to the theoretical assumptions of elastic tasks and to the user-provided parameters. We claim the elastic primitive has a lot of potential and our results show it is worth integrating in future task parallel languages.

5.8 Elastic tasks for Hybrid CPU-GPU execution

5.8.1 Vision and challenges

Our overall vision is that the elastic task construct will part of the low level work-stealing backend used by DFGR, and that the system as a whole would have a wide applicability in terms of target architectures. So far we have shown results on shared memory systems, but nowadays heterogeneous architectures are becoming the norm, and programming them efficiently remains a challenge.

We propose a practical extension to the elastic task construct to allow for the creation of elastic tasks that can run on GPUs or split the work between CPU threads and a GPU. To this end we propose generating the OpenCL kernel code for the elastic tasks and developing the runtime that would dynamically decide the execution of these tasks across platforms.

Running elastic tasks on GPUs involves multiple steps. First, we need to ensure that there is a code variant that's amenable for the architecture, more precisely generate an OpenCL kernel that we can invoke as an alternative to the CPU code provided. Secondly, we need a runtime that can adapt dynamically to the availability of the resources and decide to execute a task on the CPU, GPU. We propose the creation of a set of policies for deciding how the work in an elastic task should be executed.

Our implementation relies on two components: a compiler toolchain for the generation of OpenCL code for the GPU and a runtime scheduler enhanced with various policies for dynamically deciding where tasks run.

5.8.2 Compiler toolchain

For the compiler toolchain we use HJ-OpenCL [79], an extension to the parallel Habanero Java programming language which enables the execution of *forAll* loops on any platform on which OpenCL can run, including heterogeneous processors encompassing CPUs and GPUs. This feature can be used together with regular *async* and *finish* constructs in the language, thus enabling applications to be written in Java, rather than low level OpenCL, while still taking advantage of the performance benefits of OpenCL for accelerating computationally-intensive parallel loops. The HJ-OpenCL compiler uses APARAPI [80] for the auto-generation of bridge code between the JVM and OpenCL. APARAPI is a comprehensive, open-source framework for executing computational kernels from Java applications on OpenCL devices. The work on the HJ-OpenCL compiler has extended the APARAPI framework to enable a static translation from Java bytecode to OpenCL([79]), rather than the dynamic compilation approach found in the publicly available APARAPI.

We extend the OpenCL code provided by the HJ-OpenCL compiler to enable interaction with our elastic tasks. We use the JNI stub code along with the code for OpenCL device discovery, OpenCL platform initialization, data movement between the JVM and OpenCL devices and OpenCL kernel execution. We change the OpenCL code generated to enable communication with ElastiJ as follows [*].

We consider the communication cost to be a key variable that would be a challenge to estimate for the average programmer. As such, a task that can be mapped onto OpenCL will decide where the execution is to take place after the data transfer has been completed to the device. We create a callback from OpenCL into the JVM which occurs after all data transfers to the device have completed. This callback inquires our ElastiJ runtime whether to proceed with the execution on the GPU or whether to abort the task and continue with the next available task. This approach is conservative in the sense that it uses the GPU as an

[*]We also change the building of an OpenCL kernel to use a binary rather than source, due to an existing bug in the OpenCL framework. Though we expect this to be fixed in the near future, this approach is also more efficient than building the kernel from source.

accelerator that may be used to boost performance, not the main execution unit. However, the runtime scheduling policies presented in the next section will show how we use this approach to obtain overall better performance with a hybrid approach.

5.8.3 Runtime scheduler

We extended our ElastiJ runtime to facilitate the execution of elastic tasks on both CPU and GPU.

First, we create a new type of task: *asyncElasticGPU*, which in addition to the parameters for regular elastic tasks has an additional argument, a *runtime policy*. The available policies in our preliminary implementation are:

- *CPU*, which behaves as the regular elastic runtime,
- *GPU*, which will ensure the task runs only on the GPU, and
- *Adapt* which uses a runtime availability of the devices to decide where the task will execute.

Next, we extend our work-stealing runtime to handle the newly created task. For this we use a thread dedicated to handling GPU tasks and for managing the communication with OpenCL through JNI. The runtime behavior is as follows. When an *asyncElasticGPU* task is found, the runtime first checks its policy. If the policy is *CPU*, the task is placed in the current worker's queue and the work stealing runtime behaves as before (threads may join the task when they try to steal, once the task has been picked up by one of the other threads). If the policy is *GPU*, the task is placed in the dedicated GPU worker's queue, and if the policy is *Adapt*, the task will be placed in both queues.

The dedicated GPU worker continuously pops tasks from its queue and tries to execute them by invoking the native method. The native code being called will perform all device initialization and start the copying of data to the device, followed by a communication with the JVM to obtain the *go ahead* for teh actual kernel execution. Depending on the policy, the communication varies: for the CPU policy this path is never taken so an exception would be thrown; for the GPU policy the callback is shortcutted, as the task will surely execute natively; finally, for the Adapt policy, the decision is based on the state of the task. In our approach, if the task has began execution, the GPU execution is aborted. If however the task is not yet running — even if some threads have joined the task — the task is set to run natively and the threads associated with the task (if any) are released to steal other tasks. A detail in the implementation of such a system is obtaining proper synchronization between threads and correct semantics with respect to the number of tasks executing within a *finish* scope. In this instance, the elastic task is generally duplicated by each of the threads joining, so that each thread can process its private instance. This implies that these new tasks need to be registered with the parent task's *finish* scope, and in the event the runtime decides the task is to run natively, the deregistering needs to be done accordingly.

From the view point of the CPU workers, the behavior is similar to the regular elastic runtime, with two additional checks for native execution. The first check is performed when threads join the task: if the task has began running natively, the threads look for another task to steal (remember that a thread joins an elastic task when its deque is empty and it is stealing). The second check is done during the waiting phase: if at any wake up point the task is found to have started natively, the threads release the current task, deregister it from the finish scope if the task is a duplicate and continue looking for a new task to run. Once threads begin the actual execution in the JVM, the task can no longer be executed natively and any callback from OpenCL after this point will receive an abort reply.

The framework we proposed is extensible and enables the future creation of more complex heuristics for the hybrid CPU+GPU execution. In the next section we will focus on

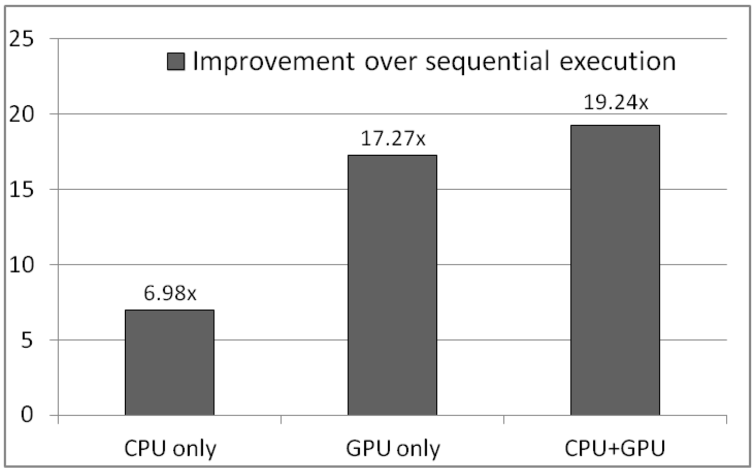


Figure 5.9 : JGF Crypt on size C (50 million elements), on Westmere, with CPU, GPU and Hybrid CPU+GPU policies.

the policies presented above.

5.8.4 Experimental results

We test our extended runtime on the Crypt benchmark from Java Grande Forum. The benchmarks consists of two phases: an encryption phase and a decryption phase, both being data parallel computations. We create a benchmark that runs 10 such encryptions and model each as an elastic task that can run on the GPU as well, in order to see the benefits of dynamic scheduling among multiple *asyncElasticGPU* tasks.

We run 5 JVM instances, each with 4 iterations of the benchmark, and use the 20 times we obtain to get the median value of the time for each of the policies. We use the same approach to obtain the median sequential execution and plot the benefit of the parallel approaches over the sequential run. Figure 5.9 shows the results for our extended runtime.

With the hybrid approach we obtain a 19x benefit over the sequential approach, a better overall benefit than either the CPU or the GPU policies.

Our experience in this work has lead of to a few insights when dealing with such hybrid execution. The first thing to note is that the benefits vary depending on the type of the computation. Another experiment we did was on the Vector-Addition benchmark which showed that a data intensive benchmark does not benefit from GPU execution. In fact Vector-Addition performed best when executing purely on the CPU. The GPU policy performed poorer, and the adaptive approach was on-par with the CPU policy, as all tasks we simply run in the JVM. Thus, in this instance, the accelerator was essentially wasted, with all attempted tasks being aborted after the data transfer. Yet, we note that the adaptive approach still yielded the best performance.

The second lesson we learnt is that the *asyncElasticGPU* arecmore sensitive to the work estimation, due to the different actual work times for the different platforms. For the Crypt benchmark, the GPU yield significatly better result than the CPU, because it is a compute intensive application. This means that the runtime should prioritize GPU execution. We achieved this by increasing the estimation of the work, in order to facilitate the data transfer ending before the tasks starts its CPU execution. In practice, 9/10 iterations of the encryption run on the GPU and only one runs on the CPU, which leads to the benefit shown in Figure 5.9. A more complex approach would be for the user to provide two work estimations (one for each device) along with an estimation of the data transfer. The runtime can than use this information to obtain a formula for computing a wait time for the CPU threads.

Though these results are but a proof of concept that elastic tasks are amenable for hybrid execution, we have shown that our extended runtime, enhanced with an adaptive policy can give the best performance available while running on CPUs and GPU.

5.9 Conclusions

In this chapter we introduced the concept of elastic tasks, a construct that enables programmers to express adaptive parallel computations and rely on an elastic runtime to offer good performance from locality and load balancing. We proved that the work-sharing scheduler completes a computation with work W and span S in $O(W/P+S+E)$ time, where E is the total number of elastic tasks, and that the work-stealing scheduler completes the computation in $O(W/P+S+E\lg P)$ expected time. We also showed preliminary practical results, that elastic tasks have the potential of improving the locality of computations, can yield comparable performance with regular tasks and that they are able to adapt at runtime, based on the load of the application. We have also shown that an enhanced runtime scheduler, coupled with a compiler for OpenCL generation can enable elastic tasks to run on hybrid architectures encompassing CPUs and GPUs, with good performance benefits.

There are many directions for future work. It would be interesting to see what kind of theoretical guarantees can be achieved using our enhanced runtime. We are also interested in investigating extensions to ElastiJ for distributed systems where we have clusters of multicore processors. On a related note, it would be interesting to extend our approach to take more of the machine topology into account, so that the workers assigned to an elastic task share a certain level of proximity. In this scenario, the locality benefits of elastic tasks should be more pronounced. Finally, elastic tasks have the potential to be useful for multiprogramming environments. In particular, say we have two programs, each of which is a data-parallel loop. If these are expressed as inelastic tasks, then they may result in extra overhead due to the creation of too many worker threads, as well as unpredictable allocations of processor cores to the two applications. We could extend elastic tasks to encode a fairness constraint allowing both programs to share the workers equitably without the programmer having to explicitly specify how many workers each program should get. Elastic tasks could also be used to support the execution of multiple OpenMP regions in parallel, should the OpenMP specification explore that possibility in the future.

Chapter 6

Related work

6.1 Related work to the data-flow programming model DFGR

DFGR has its roots in Intel's Concurrent Collections (CnC) [81, 16], a macro-dataflow models which takes a similar approach providing a separation of concerns between a domain expert, who can provide an accurate problem specification at a high level, and a tuning expert, who can tune the individual components of an application for better performance. In the original CnC implementation however there was no means of knowing which item from a particular item collection a step should read, so such information needed to be given by the user inside the step code. We extended the CnC model [21] to enable accurate definition of dependences at a high level, in a textual graph specification, removing the need to make a user familiar with a particular API. DFGR takes the idea of a high-level data-flow model one step further, by simplifying the way of specifying step-item and step-step relations one the one hand, and offering more complex means of describing complex applications on the other. DFGR is also designed to be a possible standard for an intermediate representation of parallel programs in a macro-dataflow setup. Another motivation for this approach is to reuse the optimizations performed at the graph level, such as granularity adjustments.

In this work we use the Habanero-C language to express parallelism. Other data-flow models take a similar approach of using either a threading library, such as pthreads used in TFlux [82], or a task library, such as TBB used in Intel's CnC, or a parallel language such as Cilk used in Nabbit [7]. As with Cilk, Habanero-C relies on a work-stealing scheduler

[67], which, when used with arbitrary task graphs can overload the machine. DFGR aims to ease addressing this problem by separating the application description from its concrete implementation, for instance by enabling graph transformations to coarsen the granularity of parallelism.

One of the most commonly used parallel language is OpenMP. In this model it was shown that having synchronizations using barrier is possibly very limiting for expressing applications. Recent work [29] has extended the OpenMP model to enable do-across parallelism for point-to-point synchronization, in order to make synchronization more efficient and enable data-locality. Our results show we can coarsen the granularity of paralellism and obtain good performance when using the OpenMP + DoAcross backend.

Legion [83] is another language which aims to ease the programmer's use by taking as input a sequential program and automatically determining the dependences by creating a complete set of read-write statements for all data and enabling paralellism when no dependences are found. This model however requires an initial sequential specification of a program, while DFGR does not. In addition DFGR could benefit from the locality optimizations performed in Legion.

Alpha [84] is a language which can be viewed as an embedded eDSL for the polyhedral model. However the specification for Alpha is that of a full language, whereas DFGR can be composed with optimized step code defined in other languages, as long as these can be built together.

A number of papers addressed data-flow analysis of parallel programs using the polyhedral model, including extensions of array data-flow analysis to data-parallel and/or task-parallel programs [85, 86]. These works concentrate on analysis whereas our main focus is on transformations of macro-dataflow programs. PolyGlot [87] is the first end-to-end polyhedral optimization framework for pure dataflow model such as LabVIEW, which describes streaming parallelism via wires (edges) among source, sink, and computation nodes. On the other hand, our framework aims at optimizing macro-dataflow model, where asynchronous tasks are coordinated via input/output variables in data-driven manner.

6.2 Related work to the elastic tasks

In this section, we discuss some of the closely related work to our elastic task paradigm. In previous work, Wimmer and Träff [69] have considered a similar construct for mixed-mode parallelism. However, in their work, the number of workers assigned to an elastic node is fixed, and specified by the programmer. In their work, described as team-building, they allow the programmers to express data-parallel tasks and the programmer specifies the number of workers that must be allocated to each of these tasks (therefore, these tasks are inelastic) and the runtime finds enough workers to allocate to this task before launching it. In contrast, our scheduling strategies dynamically allocate workers to elastic nodes when they become ready based on the current runtime conditions. If most workers are busy doing other work, then the elastic task is assigned fewer workers since there is already ample parallelism in the rest of the program. If most workers are idle (or stealing), then it indicates a lack of parallelism in the rest of the program and more workers are assigned to an elastic task. In addition, the modifications required to the work-stealing or work-sharing schedulers are small compared to those required by prior work. In particular, in contrast to Wimmer and Traff's work [69], we do not require any centralized coordination within the work-stealing scheduler — it remains entirely distributed. We are not aware of prior work that provides theoretical guarantees on the completion time for this combinations of sequential and elastic tasks within the same computation.

The problem of thread scheduling for locality improvements has been investigated in depth for work-stealing schedulers [75, 88, 89], both shared memory [75] and distributed

[88, 89], both in a UMA [90] and in a NUMA [91, 89] context. ADAPT [92] proposes a framework implemented at OS level, which adapts the number of threads it gives to a program, on machines where multiple programs concurrently. These works are orthogonal to elastic tasks and could benefit from one other.

There have also been ample work in the area of co-scheduling of tasks in distributed environments. DETS [93] proposes the use of a pool of tasks, much like a work-sharing scheduler, but with the limitation of assigning a master to handle the common work pool and the creation or restart of workers, which can be both unreliable and inefficient on distributed machines. A previous study [94] looked at a series of works which attempt to do coscheduling of tasks in general; this requires a coordinated effort which adds a lot of overhead in the absence of additional information. The authors formulate a mathematical model in order to analyze the approaches proposed thus far. Our work can ease the challenge of coscheduling of tasks, when computations are expressible as an elastic task, and it can also be coupled with existing strategies. A more restrictive form of co-scheduling is gang scheduling. This is generally used for inelastic tasks since the tasks often assume that they will have a fixed number of threads (generally, all the threads in the machine) when they are scheduled.

Bibliography

[1] J. Payne, V. Cavé, R. Raman, M. Ricken, R. Cartwright, and V. Sarkar, "DrHJ — a lightweight pedagogic IDE for Habanero Java," in *PPPJ'11: Proceedings of the 9th International Conference on the Principles and Practice of Programming in Java*, 2011.

[2] V. Sarkar, W. Harrod, and A. E. Snavely, "Software Challenges in Extreme Scale Systems," January 2010. Special Issue on Advanced Computing: The Roadmap to Exascale.

[3] Q. Lu, U. Bondhugula, T. Henretty, S. Krishnamoorthy, J. Ramanujam, A. Rountev, P. Sadayappan, Y. Chen, H. Lin, and T. fook Ngai, "Data layout transformation for enhancing data locality on NUCA chip multiprocessors," in *In Proceedings of the 18th International Conference on Parallel Architectures and Compilation Techniques*, pp. 348–357, 2009.

[4] S. Hong, S. Salihoglu, J. Widom, and K. Olukotun, "Simplifying scalable graph processing with a domain-specific language," in *Proceedings of Annual IEEE/ACM International Symposium on Code Generation and Optimization*, CGO '14, (New York, NY, USA), pp. 208:208–208:218, ACM, 2014.

[5] F. Darema, D. George, V. Norton, and G. Pfister, "A Single-Program-Multiple-Data Computational model for EPEX/FORTRAN," *Parallel Computing*, vol. 7, no. 1, pp. 11–24, 1988.

[6] M. Frigo, C. E. Leiserson, and K. H. Randall, “The implementation of the Cilk-5 multithreaded language,” in *Proceedings of the ACM SIGPLAN 1998 conference on Programming language design and implementation*, PLDI ’98, (New York, NY, USA), pp. 212–223, ACM, 1998.

[7] K. Agrawal, C. Leiserson, and J. Sukha, “Executing task graphs using work-stealing,” in *Parallel Distributed Processing (IPDPS), 2010 IEEE International Symposium on*, pp. 1 –12, April 2010.

[8] J. Reinders, *Intel threading building blocks*. Sebastopol, CA, USA: O’Reilly & Associates, Inc., first ed., 2007.

[9] “"MSDN Magazine: Task Parallel Library", Retrieved 2014-09-11, http://msdn.microsoft.com/en-us/magazine/cc163340.aspx.”

[10] V. Cavé, J. Zhao, J. Shirako, and V. Sarkar, “Habanero-Java: the New Adventures of Old X10,” in *PPPJ’11: Proceedings of the 9th International Conference on the Principles and Practice of Programming in Java*, 2011.

[11] S. Chatterjee, S. Tasırlar, Z. Budimlic, V. Cave, M. Chabbi, M. Grossman, V. Sarkar, and Y. Yan, “Integrating asynchronous task parallelism with mpi,” in *Parallel Distributed Processing (IPDPS), 2013 IEEE 27th International Symposium on*, pp. 712–725, May 2013.

[12] V. Sarkar and J. Hennessy, “Partitioning Parallel Programs for Macro-Dataflow,” *ACM Conference on Lisp and Functional Programming*, pp. 202–211, August 1986.

[13] “Building an open community runtime (ocr) framework for exascale systems,” Nov. 2012. Supercomputing 2012 Birds-of-a-feather session.

[14] "The Swarm Framework. http://swarmframework.org/."

[15] The STE||AR Group, "HPX, a C++ runtime system for parallel and distributed applications of any scale.."

[16] IntelCorporation, "Intel (R) Concurrent Collections for C/C++." http://softwarecommunity.intel.com/articles/eng/3862.htm.

[17] Z. Budimlić *et al.*, "Concurrent collections," *Scientific Programming*, 2010.

[18] OpenMP Architecture Review Board, "The OpenMP® API specification for parallel programming, version 4.0," 2013.

[19] UCLA, Rice, OSU, and UCSB, "Center for Domain-Specific Computing (CDSC)." http://cdsc.ucla.edu.

[20] J. Cong, V. Sarkar, G. Reinman, and A. Bui, "Customizable Domain-Specific Computing," *IEEE Design and Test*, pp. 6–15, Mar 2011.

[21] A. Sbîrlea, Y. Zou, Z. Budimlić, J. Cong, and V. Sarkar, "Mapping a data-flow programming model onto heterogeneous platforms," LCTES '12.

[22] A. Chandramowlishwaran, K. Knobe, and R. Vuduc, "Performance evaluation of concurrent collections on high-performance multicore computing systems," IPDPS, 2010.

[23] Alina Sbirlea, "Mapping a Dataflow Programming Model onto Heterogeneous Architectures (MS Thesis, https://wiki.rice.edu/confluence/download/attachments/4425835/AlinaSbirlea_MSThesis.pdf)."

[24] "https://wiki.rice.edu/confluence/display/HABANERO/Habanero-C."

[25] J. Cong, V. Sarkar, G. Reinman, and A. Bui, "Customizable Domain-Specific Computing," *IEEE Design and Test*, pp. 6–15, Mar 2011.

[26] P. Charles, C. Grothoff, V. Saraswat, C. Donawa, A. Kielstra, K. Ebcioglu, C. von Praun, and V. Sarkar, "X10: an object-oriented approach to non-uniform cluster computing," OOPSLA '05.

[27] J. Shirako, D. M. Peixotto, V. Sarkar, and W. N. Scherer, "Phasers: a unified deadlock-free construct for collective and point-to-point synchronization," in *ICS '08*, pp. 277–288, ACM, 2008.

[28] "OpenMP Application Program Interface, version 3.0, May 2008." http://www.openmp.org/mp-documents/spec30.pdf.

[29] J. Shirako, P. Unnikrishnan, S. Chatterjee, K. Li, and V. Sarkar, "Expressing doacross loop dependencies in openmp," in *9th International Workshop on OpenMP*, IWOMP, 2013.

[30] W. Ackerman and J. Dennis, "VAL - A Value Oriented Algorithmic Language," Tech. Rep. TR-218, MIT Laboratory for Computer Science, June 1979.

[31] J. McGraw, *SISAL - Streams and Iteration in a Single-Assignment Language - Version 1.0*. Lawrence Livermore National Laboratory, July 1983.

[32] Arvind, M. Dertouzos, R. Nikhil, and G. Papadopoulos, "Project Dataflow: A parallel computing system based on the Monsoon architecture and the Id programming language," tech. rep., MIT Lab for Computer Science, March 1988. Computation Structures Group Memo 285.

[33] A. Sbirlea, L.-N. Pouchet, and V. Sarkar, "DFGR: an Intermediate Graph Representation for Macro-Dataflow Programs," in *Fourth International workshop on Data-Flow Modelsfor extreme scale computing (DFM'14)*, Aug. 2014.

[34] A. Sbirlea, L.-N. Pouchet, and V. Sarkar, "Task Coarsening Through Polyhedral Compilation for a Macro-Dataflow Programming Model," in *The 5th International Workshop on Polyhedral Compilation Techniques (IMPACT) - Poster, 19 January, 2015, Amsterdam, The Netherlands*, 2015.

[35] Habanero Research Programming Group, "CnC on the Open Community Runtime (OCR)." https://github.com/habanero-rice/cnc-ocr.

[36] G. Kahn, "The semantics of a simple language for parallel programming," *Jack L. Rosenfeld (Ed.): Information Processing 74, Proceedings of IFIP Congress 74*.

[37] E. A. Lee and D. G. Messerschmitt, "Synchronous data flow," in *Proceedings of the IEEE*, vol. 75, pp. 1235–1245, IEEE, September 1987.

[38] D. Sbirlea, K. Knobe, and V. Sarkar, "Folding of tagged single assignment values for memory-efficient parallelism," in *Euro-Par'12*, pp. 601–613, 2012.

[39] D. Sbîrlea, Z. Budimlić, and V. Sarkar, "Bounded memory scheduling of dynamic task graphs," in *Proceedings of the 23rd International Conference on Parallel Architectures and Compilation*, PACT '14, (New York, NY, USA), pp. 343–356, ACM, 2014.

[40] D. Sbîrlea, A. Sbîrlea, K. Wheeler, and V. Sarkar, "The flexible preconditions model for macro-dataflow execution," in *DFM*, 2013.

[41] U. Banerjee, *Unimodular transformations of double loops*. University of Illinois at Urbana-Champaign, Center for Supercomputing Research and Development, 1990.

[42] P. Feautrier, "Dataflow analysis of scalar and array references," *IJPP*, vol. 20, no. 1, pp. 23–53, 1991.

[43] G. Gupta and S. Rajopadhye, "The z-polyhedral model," in *ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP'07)*, pp. 237–248, ACM, 2007.

[44] D. K. Wilde, "A library for doing polyhedral operations," Tech. Rep. 785, IRISA, Rennes, France, 1993.

[45] C. Bastoul, "Code generation in the polyhedral model is easier than you think," in *PACT*, pp. 7–16, 2004.

[46] S. Verdoolaege, "ISL: An integer set library for the polyhedral model," in *Mathematical Software–ICMS 2010*, pp. 299–302, Springer, 2010.

[47] C. Bastoul, "Code generation in the polyhedral model is easier than you think," in *Proceedings of the 13th International Conference on Parallel Architectures and Compilation Techniques*, PACT '04, (Washington, DC, USA), pp. 7–16, IEEE Computer Society, 2004.

[48] P. Feautrier and C. Lengauer, "The Polyhedron Model," *Encyclopedia of Parallel Programming*, 2011.

[49] D. G. Wonnacott, *Constraint-based Array Dependence Analysis*. PhD thesis, College Park, MD, USA, 1995. UMI Order No. GAX96-22167.

[50] L.-N. Pouchet, "PoCC - the Polyhedral Compiler Collection, http://www.cs.ucla.edu/ pouchet/software/pocc/."

[51] L. Lamport, "Time, clocks, and the ordering of events in a distributed system," *Commun. ACM*, vol. 21, pp. 558–565, July 1978.

[52] U. Bondhugula and J. Ramanujam, "Pluto: A practical and fully automatic polyhedral parallelizer and locality optimizer," tech. rep., 2007.

[53] U. Bondhugula, A. Hartono, J. Ramanujam, and P. Sadayappan, "A practical automatic polyhedral parallelizer and locality optimizer," in *PLDI*, 2008.

[54] J. Shirako, L.-N. Pouchet, and V. Sarkar, "Oil and Water Can Mix: An Integration of Polyhedral and AST-based Transformations.," in *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, SC '14, 2014.

[55] R. Cytron, "Doacross: Beyond Vectorization for Multiprocessors," in *ICPP'86*, pp. 836–844, 1986.

[56] "OpenMP Technical Report 3 on OpenMP 4.0 enhancements." http://openmp.org/TR3.pdf.

[57] J. Shirako, P. Unnikrishnan, S. Chatterjee, K. Li, and V. Sarkar, "Expressing DOACROSS Loop Dependencies in OpenMP," in *9th International Workshop on OpenMP (IWOMP)*, 2011.

[58] P. Unnikrishnan, J. Shirako, K. Barton, S. Chatterjee, R. Silvera, and V. Sarkar, "A practical approach to DOACROSS parallelization," in *Euro-Par*, 2012.

[59] "The PACE compiler project," *http://pace.rice.edu*.

[60] Louis-Noel Pouchet, "The Polyhedral Benchmark Suite."

[61] A. Sbîrlea, JunShirako, and V. Sarkar, "Polyhedral Optimizations for a Data-Flow Graph Language," in *under submission*, 2015.

[62] "Hydrodynamics Challenge Problem, Lawrence Livermore National Laboratory," Tech. Rep. LLNL-TR-490254.

[63] I. Karlin *et al.*, "Lulesh programming model and performance ports overview," Tech. Rep. LLNL-TR-608824, December 2012.

[64] A. Chandramowlishwaran, K. Knobe, and R. Vuduc, "Performance evaluation of concurrent collections on high-performance multicore computing systems," in *Parallel Distributed Processing (IPDPS), 2010 IEEE International Symposium on*, pp. 1 –12, April 2010.

[65] R. Cytron, J. Lipkis, and E. Schonberg, "A Compiler-Assisted Approach to SPMD Execution," *Supercomputing 90*, November 1990.

[66] V. K. Nandivada, J. Shirako, J. Zhao, and V. Sarkar, "A transformation framework for optimizing task-parallel programs," *ACM Trans. Program. Lang. Syst.*, vol. 35, pp. 3:1–3:48, Apr. 2013.

[67] R. D. Blumofe and C. E. Leiserson, "Scheduling multithreaded computations by work stealing," *J. ACM*, vol. 46, no. 5, pp. 720–748, 1999.

[68] A. Sbirlea, K. Agrawal, and V. Sarkar, "Elastic Tasks: Unifying Task Parallelism and SPMD Parallelism with an Adaptive Runtime," in *21st International European Conference on Parallel and Distributed Computing (Euro-Par'15)*, Aug. 2015.

[69] M. Wimmer and J. L. Träff, "Work-stealing for mixed-mode parallelism by deterministic team-building," in *Proceedings of the Twenty-third Annual ACM Symposium on Parallelism in Algorithms and Architectures*, SPAA '11, (New York, NY, USA), pp. 105–116, ACM, 2011.

[70] C.-K. Luk, S. Hong, and H. Kim, "Qilin: Exploiting parallelism on heterogeneous multiprocessors with adaptive mapping," in *Proceedings of the 42Nd Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO 42, (New York, NY, USA), pp. 45–55, ACM, 2009.

[71] J. S. Auerbach, D. F. Bacon, P. Cheng, and R. M. Rabbah, "Lime: a java-compatible and synthesizable language for heterogeneous architectures.," in *OOPSLA'10*, pp. 89–108, 2010.

[72] R. L. Graham, "Bounds on multiprocessing anomalies," *SIAM Journal on Applied Mathematics*, pp. 17(2):416–429, 1969.

[73] N. S. Arora, R. D. Blumofe, and C. G. Plaxton, "Thread scheduling for multiprogrammed multiprocessors," in *In Proceedings of the Tenth Annual ACM Symposium on Parallel Algorithms and Architectures (SPAA)*, pp. 119–129, 1998.

[74] S. Imam and V. Sarkar, "Habanero-Java Library: a Java 8 Framework for Multicore Programming," in *11th International Conference on the Principles and Practice of Programming on the Java Platform: virtual machines, languages, and tools*, PPPJ'14, 2014.

[75] Y. Guo, J. Zhao, V. Cavé, and V. Sarkar, "SLAW: a Scalable Locality-aware Adaptive Work-stealing Scheduler," in *IPDPS '10: Proceedings of the 2010 IEEE International*

Symposium on Parallel&Distributed Processing, (Washington, DC, USA), pp. 1–12, IEEE Computer Society, Apr 2010.

[76] S. Gupta and V. K. Nandivada, "IMSuite: A Benchmark Suite for Simulating Distributed Algorithms," *ArXiv e-prints*, Oct. 2013.

[77] A. Georges, D. Buytaert, and L. Eeckhout, "Statistically Rigorous Java Performance Evaluation," in *Proceedings of the 22nd annual ACM SIGPLAN conference on Object-oriented programming systems and applications*, OOPSLA '07, (New York, NY, USA), pp. 57–76, ACM, 2007.

[78] P. Tsigas and Y. Zhang, "A simple, fast parallel implementation of quicksort and its performance evaluation on sun enterprise 10000," in *PDP*, pp. 372–, IEEE Computer Society, 2003.

[79] A. Hayashi, M. Grossman, J. Zhao, J. Shirako, and V. Sarkar, "Accelerating habanero-java programs with opencl generation," in *Proceedings of the 2013 International Conference on Principles and Practices of Programming on the Java Platform: Virtual Machines, Languages, and Tools*, PPPJ '13, (New York, NY, USA), pp. 124–134, ACM, 2013.

[80] "APARAPI. API for Data Parallel Java.."

[81] Z. Budimlić, M. Burke, V. Cavé, K. Knobe, G. Lowney, R. Newton, J. Palsberg, D. Peixotto, V. Sarkar, F. Schlimbach, and S. Taşirlar, "Concurrent Collections," *Scientific Programming*, vol. 18, pp. 203–217, August 2010.

[82] K. Stavrou, M. Nikolaides, D. Pavlou, S. Arandi, P. Evripidou, and P. Trancoso, "TFlux: A portable platform for data-driven multithreading on commodity multicore systems," ICPP '08, 2008.

[83] M. Bauer, S. Treichler, E. Slaughter, and A. Aiken, “Legion: Expressing locality and independence with logical regions,” in *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, SC ’12, 2012.

[84] T. Yuki, G. Gupta, D. Kim, T. Pathan, and S. Rajopadhye, “AlphaZ: A System for Design Space Exploration in the Polyhedral Model,” in *LCPC*, 2012.

[85] J.-F. Collard and M. Griebl, “Array Dataflow Analysis for Explicitly Parallel Programs,” in *Proceedings of the Second International Euro-Par Conference on Parallel Processing*, Euro-Par ’96, 1996.

[86] T. Yuki, P. Feautrier, S. Rajopadhye, and V. Saraswat, “Array Dataflow Analysis for Polyhedral X10 Programs,” in *Proceedings of the 18th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP ’07, 2013.

[87] S. G. Bhaskaracharya and U. Bondhugula, “Polyglot: a polyhedral loop transformation framework for a graphical dataflow language,” in *Compiler Construction*, pp. 123–143, Springer, 2013.

[88] J. Paudel, O. Tardieu, and J. N. Amaral, “On the merits of distributed work-stealing on selective locality-aware tasks,” in *Proceedings of the 2013 42Nd International Conference on Parallel Processing*, ICPP ’13, (Washington, DC, USA), pp. 100–109, IEEE Computer Society, 2013.

[89] Q. Chen, M. Guo, and H. Guan, “Laws: Locality-aware work-stealing for multi-socket multi-core architectures,” in *Proceedings of the 28th ACM International Conference on Supercomputing*, ICS ’14, (New York, NY, USA), pp. 3–12, ACM, 2014.

[90] S. Zhuravlev, S. Blagodurov, and A. Fedorova, “Addressing shared resource contention in multicore processors via scheduling,” in *Proceedings of the Fifteenth Edi-*

tion of ASPLOS on Architectural Support for Programming Languages and Operating Systems, ASPLOS XV, (New York, NY, USA), pp. 129–142, ACM, 2010.

[91] S. Blagodurov, S. Zhuravlev, A. Fedorova, and A. Kamali, "A case for numa-aware contention management on multicore systems," in *Proceedings of the 19th International Conference on Parallel Architectures and Compilation Techniques*, PACT '10, (New York, NY, USA), pp. 557–558, ACM, 2010.

[92] K. Kumar Pusukuri, R. Gupta, and L. N. Bhuyan, "Adapt: A framework for coscheduling multithreaded programs," *ACM Trans. Archit. Code Optim.*, vol. 9, pp. 45:1–45:24, Jan. 2013.

[93] H. Zhan, L. Kang, and D. Cao, "Dets: A dynamic and elastic task scheduler supporting multiple parallel schemes," in *Proceedings of the 2014 IEEE 8th International Symposium on Service Oriented System Engineering*, SOSE '14, (Washington, DC, USA), pp. 278–283, IEEE Computer Society, 2014.

[94] M. S. Squillante, Y. Zhang, A. Sivasubramaniam, N. Gautam, H. Franke, and J. Moreira, "Modeling and analysis of dynamic coscheduling in parallel and distributed environments," in *Proceedings of the 2002 ACM SIGMETRICS International Conference on Measurement and Modeling of Computer Systems*, SIGMETRICS '02, (New York, NY, USA), pp. 43–54, ACM, 2002.