RICE UNIVERSITY

Mapping a Dataflow Programming Model onto Heterogeneous Architectures

by

Alina Gabriela Sbirlea

A Thesis Submitted
in Partial Fulfillment of the
Requirements for the Degree

Master of Science

Approved, Thesis Committee:

Vivek Sarkar

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

Keith D Cooper

Keith D. Cooper
L. John and Ann H. Doerr Professor of
Computational Engineering

John Mellor Crummey
Professor of Computer Science
Professor of Electrical and Computer
Engineering

Zoran Budimlic
Research Scientist
Computer Science Department

Houston, Texas
November, 2011

ABSTRACT

Mapping a Dataflow Programming Model onto Heterogeneous Architectures

by

Alina Gabriela Sbirlea

This thesis describes and evaluates how extending Intel's Concurrent Collections (CnC) programming model can address the problem of hybrid programming with high performance and low energy consumption, while retaining the ease of use of data-flow programming.

The CnC model is a declarative, dynamic light-weight task based parallel programming model and is implicitly deterministic by enforcing the single assignment rule. These properties ensure that problems are modelled in an intuitive way. CnC offers a separation of concerns by allowing algorithms to be expressed as a two stage process: first by decomposing a problem into components and specifying how components interact with each other, and second by providing an implementation for each component. By facilitating the separation 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 for better performance, we ensure that tuning and future development, such as replacement of a subcomponent with a more efficient algorithm, become straightforward.

A recent trend in mainstream desktop systems is the use of graphics processor units (GPUs) to obtain order-of-magnitude performance improvements relative to general-purpose CPUs. In addition, the use of FPGAs has seen a significant increase

for applications that can take advantage of such dedicated hardware. We see that computing is evolving from using many core CPUs to "co-processing" on the CPU, GPU and FPGA, however hybrid programming models that support the interaction between multiple heterogeneous components are not widely accessible to mainstream programmers and domain experts who have a real need for such resources.

We propose a C-based implementation of the CnC model for enabling parallelism across heterogeneous processor components in a flexible way, with high resource utilization and high programmability. We use the task-parallel HabaneroC language (HC) as the platform for implementing CnC-HabaneroC (CnC-HC), a language also used to implement the computation steps in CnC-HC, for interaction with GPU or FPGA steps and which offers the desired flexibility and extensibility of interacting with any other C based language.

First, we extend the CnC model with *tag functions* and *ranges* to enable automatic code generation of high level operations for inter-task communication. This improves programmability and also makes the code more analysable, opening the door for future optimizations. Secondly, we introduce a way to specify steps that are data parallel and thus are fit to execute on the GPU, and the notion of task *affinity*, a tuning annotation in the specification language. Affinity is used by the runtime during scheduling and can be fine-tuned based on application needs to achieve better (faster, lower power, etc.) results. Thirdly, we introduce and develop a novel, data-driven runtime for the CnC model, using HabaneroC (HC) as a base language. In addition, we also create an implementation of the previous runtime approach and conduct a study to compare the performance. Next, we expand the HabaneroC *dynamic work-stealing* runtime to allow *cross-device stealing* based on task affinity. Cross-device dynamic work-stealing is used to achieve load balancing across heterogeneous platforms for improved performance. Finally, we implement and use a series of benchmarks for testing the model in different scenarios and show that our proposed approach can yield significant performance benefits and low power usage when using a hybrid execution.

Acknowledgments

To my advisor, Professor Vivek Sarkar, for his invaluable advise, support and feedback throughout this work. To Professor Keith Cooper for his kind words of encouragements and advice on improving both the work and the exposure of this thesis. To Professor John Mellor-Crummey for evaluating and proving detailed feedback for the improvement of this work and thesis. To Zoran Budimlic for the discussions on the implementation of CnC-HC, his feedback and thoughts.

To the HabaneroC team for the development of the language my work is based on, for their timely response to any language issues that arouse as my work progressed, enabling both HabaneroC and CnC-HC to mature together. For being there to help either with a simple code understanding or complex synchronization issues, (alphabetical order) Zoran Budimlic, Vincent Cave, Vivek Sarkar, Yonghong Yan. To Sagnak Tasirlar for the implementation of Cholesky in CnC-HabaneroJava, used as reference for my CnC-HC implementation and to Aparana Chandramowlishwaran for the Intel-CnC Cholesky implementation. To Max Grossman, undergraduate student at Rice University, for the work done on CnC-CUDA [1] which introduced the GPU extensions in CnC-HabaneroJava. To Yi Zou, PhD. student in Computer Science at UCLA for the implementation of the imaging pipeline on the Convey machine located at UCLA. To the NSF Expeditions Center for Domain Specific Computing for funding and motivating this work.

Last but not least to my family for their support in pursuing this road and a special thanks to my husband for the late long talks, advice, support and care.

Contents

Illustrations

Tables

Chapter 1

Introduction

The computer industry is at a major inflection point in its hardware roadmap due to the end of a decades-long trend of exponentially increasing clock frequencies. Future hardware is expected to be built using homogeneous and heterogeneous many-core processors with order of hundreds of cores per chip and complex architectures designed to target the challenges of concurrency, fault tolerance, energy efficiency and resilience. The current trend is to develop massively multicore architectures that can be specialized to satisfy the energy constraint. These radical changes in hardware architectures have already begun to impact software, as the burden shifts to the programmer to make parallelism explicit. Old views have already begun to change, from finding ways to map software onto complex hardware, to designing hardware based on software needs. These software challenges are further compounded by the need to enable parallelism in mainstream workloads and application domains that have traditionally not had to employ parallelism in the past. Despite over four decades of research, few high-level parallel programming models are available to domain experts who are not at the same time experts in parallelism. Fortunately, this situation is starting to change. Frameworks such as Map-Reduce [3] successfully exploit implicit parallelism on distributed systems and have also been extended to heterogeneous platforms such as GPU [4] and FPGA [5]; however, have a restricted programming model. Other models, such as CUDA [6] and OpenCL [7], provide a restricted programming model to the users of GPU accelerators, but also expose a significant amount of hardware details.

A recent trend in mainstream desktop systems is the use of graphics processor units (GPUs) to obtain order-of-magnitude performance improvements relative to general-purpose CPUs. In addition, FPGAs have seen increasing use by applications that can take advantage of such dedicated hardware. We see that computing is evolving from using many core CPUs to "co-processing" on the CPU, GPU and FPGA. However, hybrid programming models that support the interaction between the two are not widely accessible to mainstream programmers and domain experts who have a real need for such resources.

At the other end of the spectrum, there are coordination languages such as Intel's Concurrent Collections* (CnC) [9] which are explicitly designed to be implicitly parallel and easy to use by programmers with no knowledge of parallel programming. CnC is a declarative coordination language that can express any dependency graph through combinations of task and data parallelism. CnC computations are built using steps that are related by data and control dependences. These are specified in a straight forward manner in a meta language that can describe a CnC graph. CnC is provably deterministic [9]. While this restricts CnC's scope, it is more general than other deterministic programming models including dataflow and stream-processing, and can incorporate static and dynamic forms of task, data, loop, pipeline, and tree parallelism. All of these properties make the CnC model a very good choice for a domain expert, but what it lacks is the ability to target emerging heterogeneous architectures. To address this problem, we first worked on extending the model for execution on the GPU [1], creating the first such extension. This was developed by extending a variant of Concurrent Collections[9] built on top of HabaneroJava (HJ)

*An earlier version of CnC was called TStreams [8].

[10, 11] programming model with promising results. However, since the model was implemented in a high-level language such as Java, we needed to make the calls to native code through an additional layer (the Java native interface -JNI), which added an additional overhead of copying data and had limited interoperability.

In this work, we show how to extend the CnC programming model to address the problem of hybrid programming while retaining the ease of programming introduced by CnC and enabling host-device interaction. To ensure that we can express parallelism in an easy way and also allow users of the CnC model to express parallelism within a computational step if they so desire, we have chosen a research parallel programming language that is based on C, called HabaneroC (HC). We name this C-based approach Concurrent Collections-HabaneroC (CnC-HC). We have chosen the C language due to past experience with a Java approach, having as a target obtaining the best possible performance, a goal which implied using a language that is low level enough but provide this, but still reasonable to program in. In addition, we wanted to target heterogeneous architectures for which programs are written in languages that can interact easily with the C language. We use HC as the language for implementing the CnC-HC runtime, the CnC computational steps as well as for the interaction with CUDA GPU steps.

The first extension we will present is the introduction of tag functions and ranges in the the CnC graph file specifications. Tag functions specify the relation between the tag that uniquely identifies a step and the tags identifying the data items that the step is either reading or writing, while ranges define contiguous sets of items read or written by a step. These extensions enables the compiler to automatically generate high level operations for inter-task communication, more precisely, the auto-generation of the calls to the functions that read the data items (Get functions) within a computational step. It also enables the generation of the function calls that write data items (Put functions) and that spawn of other steps, yet, as these can be optional depending on each step instance, the generated code will be considered helper code rather than a requirement. C being a low level language, the ability to have much of the interaction code generated automatically, improves programmability and ensures ease of use. This addition of tag functions also makes the code more analysable opening the path for research topics: static analysis of the graph for guarantees such as graph termination, item garbage collection or graph correctness (e.g., there is a path from the steps created by the environment to all steps specified in the graph; all the data read by the environment after the graph execution will be available as it was created during the run).

The second aspect we present is a novel, data-driven runtime for the CnC model, developed in a C framework, using HabaneroC (HC). We also create an implementation of the previous runtime approach and conduct a study between the two runtimes to determine the situations when each performs best. The result can be dependent on the application but also on the scheduling policies defined by HabaneroC, which we shall detail these throughout this thesis.

The third aspect we will tackle is the extension of CnC-HC for heterogeneous architectures. To achieve this, we introduce a way to specify steps that are data parallel and thus are fit to execute on the GPU, and the notion of task *affinity*, a tuning annotation which offers hints on where a computational step should be run. Affinities are used by the runtime during scheduling and can be fine-tuned based on application needs to achieve better (faster, lower power, etc.) results. Further, we expand the HabaneroC dynamic work-stealing runtime to allow cross-device stealing based on task affinity. Cross-device dynamic work-stealing is used to achieve load balancing across heterogeneous platforms for improved performance.

Finally, in our experimental results, we use a series of a series of benchmarks to show that, by using the new CnC-HC, we can match or exceed the performance of Intel's CnC and OpenMP, as well as yield a significant increase in performance and decrease in power consumption when using CnC-HC for a hybrid CPU/GPU/FPGA execution.

1.1 Motivation

Nowadays the roadmaps of hardware and software seem to be on a collision course. The direction of hardware design is to develop a highly customizable hardware whose parameters can be altered based on application needs. For this to be possible, a lot of work is necessary on the application front to abstract the right principles and identify the relevant primitives that characterize a domain.

Our ongoing work in the NSF Expeditions Center for Domain Specific Computing (CDSC) project [12, 13] targets precisely this problem and motivates this thesis. The project aims to determine what hardware features are most useful for a given domain. The high-level idea is that customization is a two-stage process. One involves configuring for the domain and one for a particular application. The configurations that this project is attempting to select for a specific domain involve the instruction window, cache/icache locality, register number, register file organization, allocation of interconnect bus resources. Possible hardware configurations for the customizable platform developed in this project include turning on/off some of the cores for energy efficiency. The work we present here is a first step in achieving our long-term goal of learning from software when developing the appropriate hardware architecture and vice-versa, also known as hardware-software co-design. This first step involves running sample applications for the medical imaging domain in a flexible modelling language and using it to study the computation and communication in these applications.

In this thesis, we propose using CnC for mapping such applications onto heterogeneous architectures, more precisely we propose using CnC as a flexible programming model that can be used by domain experts for such a customizable hardware platform. The Concurrent Collections programming model was chosen for its ease of use for domain experts and the mainstream programmer, who don't have to worry about low level details of parallelism. The extensions we will present keep these properties but target the use of more complex architectures. Work on CnC-HC is further motivated by the fact that, being a C based language, it is very well suited for heterogeneous architectures.

My thesis is that a high level data flow model targeted for domain experts can be and is desirable to be extended for execution on hybrid architectures, with very good performance results, high programmability and low energy consumption.

1.2 Contributions

The main contributions of this thesis are:

1. introducing tag functions to the CnC model to relate a step's unique identifier - the tag - to the data that it reads, writes and other steps it may start,

2. extending a translator for generating appropriate C code, in accordance with the tag functions specified in the CnC graph file. The generated code performs read operations (Get), task creation and launching to increase programmability,

3. developing the CnC-HC Restart and Replay runtime, the previous runtime approach for the CnC model,

4. developing the CnC-HC Data Driven runtime, our proposed alternative to 3, and comparing the two runtimes,
5. adding GPU steps to the CnC-HC model and generating the appropriate C and most of the CUDA interface code with the CnC runtime and user code,
6. extending the specification language, translator and runtime for CnC to include an affinity component for GPU and FPGA devices,
7. extending the HC work-stealing runtime to perform cross device work-stealing based on the affinity metric in order to achieve load balancing of the work across devices,
8. porting a series of benchmarks to CnC-HC and CnC-HC-Hybrid.

The rest of this thesis is organized as follows. Chapter 2 summarizes the background and related work. Chapter 3 introduces a new specification language for CnC-HC. Chapter 4 describes the changes to the tool that interprets the input CnC graph (the translator) needed to support and enable the language extensions proposed in Chapter 3 while Chapter 5 presents the runtime developed for CnC-HC. Further in Chapter 6 we discuss the extensions needs in the graph language and the necessary additions to the translator and runtime for heterogeneous execution. We present and discuss the experimental results in Chapter 7 and finally conclude the thesis and outline possible future work in Chapter 8.

Chapter 2

Background

In this chapter we present in more detail aspects which will help understand our work and relate it to other work in the field. First, as this work proposes a parallel programming model, we shall discuss how it fits into the world of parallel programming models. Secondly, we shall detail the HabaneroC programming model, which we have chosen as the basis for this work, because it is a general model which express parallelism in a straight forward manner, because it is a C based language which can interact easily with languages which address accelerators and also because we were able to access and modify the runtime to achieve this work's objectives. Next we describe the Concurrent Collection programming model, a variant of which this thesis brings forward, using as a lower level layer HabaneroC. As our work develops extensions for heterogeneous architectures, which include GPUs, and generates CUDA code for such devices, we next discuss the CUDA programming model. Finally, we shall discuss other work which extends the Concurrent Collection model for various goals and also describe other programming models which address the problem of modelling applications for heterogeneous architectures.

2.1 Parallel programming models

A parallel programming model refers to the means of specifying a program such that it can be compiled to run simultaneously on multiple execution units. The primary ways of classifying parallel programming models are based on the approaches taken for process interaction and problem decomposition. These classifications are by no means exhaustive; our aim is not to cover all the models in existence but to highlight those that are most relevant to this thesis.

Considering process interactions, we can distinguish between shared memory and message passing or distributed memory models. In the latter category, we briefly mention MPI [14] and Java RMI [15] as the models most commonly used currently and focus further on the shared memory models which are more relevant to this work. In a shared memory model, parallel tasks share a global address space which is accessed asynchronously when reading and writing data. The model thus requires the use of synchronization primitives to control concurrent accesses to shared variables, to ensure correctness and determinism. We detail a few of the representative shared memory models developed thus far that are most relevant to this work.

At the lowest level, parallelism is expressed by means of threads. For UNIX systems, a standardized C language threads programming interface has been specified by the IEEE POSIX 1003.1c standard. Implementations that adhere to this standard are referred to as POSIX threads, or Pthreads [16]. The routines that are included in these implementations enable the programmer to manage all the low-level details of parallel programming, such as thread management and synchronization. However, not many programmers are trained in all the intricacies of Pthreads, and many models have been designed to hide the thread management details while still offering effective means for exploiting parallelism.

One of the best known shared memory parallel programming models is OpenMP [17, 18]. It is an API (Application Program Interface) implemented for C and Fortran that encompasses a series of runtime library functions and directives for specifying parallel regions, work sharing, synchronization and environment variables [19].

Considering the problem decomposition classification, we distinguish between data parallelism and task parallelism. Data parallelism is achieved in SIMD, when the same operations are performed on different pieces of the data. This usually refers to vector parallelism, done through vector instructions support from the underlying architecture as well as from the compiler. Examples of data parallel languages include High Performance Fortran [20] and NESL [21]. Another more recent execution model for data parallelism can be found in SIMT models for GPU, such as CUDA (see section 2.4).

Our main interest in this thesis is in task parallel languages, in which different processors may execute different parts of the computation. In general, a whole computation can be viewed as a work flow, where the nodes contain pieces of computation and the edges symbolize data communication and synchronization.

In particular, the dynamic light-weight task-based model defines the basic unit of computation as a task, which may be far smaller than an OS thread, thus enabling fine grained parallelism. The coarsening is performed dynamically at runtime by assigning a set of light-weight tasks to each worker thread, depending on the inherent dependences in the work flow. Some of the better known programming models which adhere to this task-parallel model are Cilk [22], Intel Threading Building Blocks [23], Chapel [24], Fortress [25], OpenMP 3.0 [26], X10 [27], HabaneroJava [10] and our choices for this work, HabaneroC [28] and Concurrent Collections (CnC) [29] [9]. We have chosen HabaneroC, because it is a C-based language thus low level enough to allow us to obtain the best possible performance and easy interaction with languages that address accelerators, while also being a general model which expresses parallelism in a straight forward manner. CnC adds another level of abstraction through the definition of any problem by means of a dependency graph, offering ease of use for non-expert parallel programmers and enabling the definition of arbitrary task graphs.

2.2 HabaneroC

The HabaneroC (HC) language is a parallel programming model currently being developed at Rice University in the Habanero Research project. Previous work on HabaneroJava motivated this new C-based approach as a means of designing a language well suited for homogeneous and heterogeneous processors, in particular the Customizable Heterogeneous Platform (CHP) being developed in the NSF Expeditions Center for Domain-Specific Computing (CDSC) [13], which includes CPUs, GPUs, and FPGAs.

The two main features of HC we refer to throughout this work, and that we detail further are:

1. The *async* and *finish* constructs which define lightweight dynamic task creation and termination [30]

2. Hierarchical place trees for locality control [31].

HabaneroC has two basic primitives for the task parallel programming model borrowed from X10 [27]: async and finish.

The async statement (async ⟨ stmt ⟩ or async { stmt1 .. stmtN }) causes the parent task to fork a new child task that will execute one or more statements. The parent task continues executing the statements that follow the async statement and does not wait for the child task to finish its execution. The finish statement (finish ⟨ stmt ⟩ or finish { stmt1 .. stmtN }) performs a join operation. The task entering a finish scope will execute all the statements in the finish scope which may spawn one or more children. It will then wait for all tasks created in the finish scope to terminate. The language permits arbitrary level of nesting of async and finish scopes.

The HabaneroC runtime uses a work-stealing scheduler that supports work-first and help-first policies [30]. In the work-first policy, the current thread will start to execute a newly spawned task, adding a continuation, representing its parent task, to its work queue from which other threads can steal it. In the help-first policy, the worker will make the child task available for stealing and continue executing the parent task. The work-first policy is good for scenarios when work-stealing is rare, but it performs poorly in situations when the task continuation encompasses high parallelism that is not split into tasks, such as a for loop creating tasks using the async statement. It also has a provable memory bound but can overflow the stack for large irregular computations. The help-first policy performs well when stealing is frequent, it uses little stack but the memory usage can be large. Previous work [30] has shown how the two policies perform in different scenarios and that an adaptive approach that can switch between the two policies can yield good performance improvement and efficient stack and memory usage.

For locality, HabaneroC uses Hierarchical Place Trees (HPTs) [31]. HPTs define hierarchical trees of execution location which are an abstraction for underlying hardware. These places could be cores, groups of cores sharing different levels of cache or devices such as GPUs or FPGAs. The HC language allows the programmer to spawn a task explicitly at such a place and the work-stealing runtime is designed to take advantage of this information and preserve locality.

2.3 Concurrent Collections

Concurrent Collections (CnC) [9] is a shared memory, dynamic light-weight task-based parallel programming model. The model can also be described as a macro-dataflow coordination language as it specifies how steps interact with each other or depend on each other, separating these dependences into data and control dependences. A program in the CnC model is defined by a graph describing the dependences between serial pieces of computation called steps.

A graph specification for any program will be built using three components: steps, items and tags. These are grouped into collections:

1. a step collection is a group of steps with the same functional behaviour with respect to their inputs. They are declared using parenthesis and are represented graphically using circles.

2. an item collection is a group of data items having the same type. They are declared in the textual representation using brackets and in the graphical representation using rectangles.

3. a control collection is a group of tags or keys used to uniquely identify items and steps within their respective collections, hence the name tag collection is often used as well. These are declared using angle brackets and are pictured as triangles.

The Concurrent Collection model enforces a dynamic single assignment for items added (*Put*) in an item collection, thus also ensuring that the value that a step will read (*Get*) is the same every time (if the item is empty, the Get operation simply blocks). This property makes the model provably deterministic and race-free [9].

CnC Flavours	Intel CnC	Rice CnC-HJ	Rice CnC-HC
Language	C++	Java	C
Single Assignment Enforced	-	✓	✓
Gets before Puts	✓	✓	✓
Data Dependent Gets	✓	✓	-
Tag Functions	-	-	✓
Auto-generated Step Stubs	✓	✓	✓
Auto-generated Gets	-	-	✓
GPU Extensions	-	✓	✓

Table 2.1 : Comparison between different implementation flavours of the Concurrent Collection model

An implementation dependent property is that of dead-lock freedom. Dead-lock can occur if a step is waiting on an item that is never written, and if the manner this behaviour is implemented in the runtime involves a blocking operation. The CnC-HC implementation we present in this thesis is dead-lock free.

The original definition and implementation of the Concurrent Collections model was done by Intel. The language chosen for the step and runtime implementation in the Intel release is C++ [32]. A second implementation designed and implemented in the Habanero group is CnC-HJ which uses HabaneroJava (HJ) [11] for defining steps, modelling parallelism and ensuring data availability. This thesis presents our work on CnC-HC which uses HabaneroC as the language for implementing CnC steps and for interaction between the CnC-HC runtime, CPU steps and device steps.

We list the differences between these three flavours of CnC in the table 2.1. The single assignment rule is defined in the specification language of the CnC model, but it is only enforced in the HJ and HC implementations making these more accurate in terms of respecting the language specification.

In all flavours of CnC, a translator is used to interpret the graph specification and generate a series of files that are either used to interact with the runtime or are used as step stubs or suggestion code, all of these offering help to the programmer. The Intel CnC implementation also includes an API which allows more experienced users to bypass the translator.

In all implementations, the calls to the Get functions within a step, that read data, precede the calls to the Put functions, but this restriction is not imposed in Intel's implementation, whereas it is in the Java and C versions. However, this is not in fact a limitation, as any step executing a sequence of Get-Put-Get-Put operations can be split into multiple steps, each adhering to the rule of calling Get before Put. Implementing a source-to-source translator to automate splitting of steps, is a good opportunity for future research.

A feature which we introduced for the first time in the CnC model is a means to auto-generate Get functions by introducing *tag functions* in the graph specification language. We detail this in Chapter 3. Another aspect relating to this is that of data-dependent Gets. This means that information from the one of the items read is used to determine which item is read next. The HC implementation requires that items read be independent of each other due to the addition of tag functions which we believe are a good improvement for usability. This restriction can in fact be relaxed as any step that needs data-dependent Gets can also be split into two or more steps which do not have this requirement. Splitting a step can be fully automated and is a subject for future work, as with splitting steps to ensure the Gets-before-Puts rule.

An extension explored in CnC-HJ and CnC-HC is the ability to execute CnC steps on GPU devices, while the model manages the data transfers to and from the device. In addition, we extended the model with a notion of affinity, as a way of specifying multiple types of device steps and having the ability to achieve load balancing between CPUs, GPUs and FPGAs. This is described in Chapter 6.

Another flavour currently under development at Rice University is CnC-Python [33, 34].

2.4 The CUDA programming model

The NVIDIA CUDA programming model is an interface which enables programmers to access the highly parallel hardware of programmable Graphics Processing Units (GPUs). Though initially designed for graphics rendering, these GPUs are now being used for a wide variety of applications, including image and video processing, computational biology and chemistry, fluid dynamics simulation, CT image reconstruction, seismic analysis, ray tracing, and much more [35].

CUDA is an extension of the C programming language, with the CUDA runtime library providing a collection of device memory management, host-device stream synchronization, and execution control functions (among others). The general flow of a CUDA program consists of the following steps [36], where all allocation and copying of device memory is controlled explicitly or implicitly by the host using the CUDA runtime library:

1. CPU initiates a data copy from CPU main memory to GPU memory

2. CPU instructs GPU to start a kernel

3. GPU executes the kernel in parallel and accesses GPU memory

4. CPU initiates a copy that moves results from GPU memory to CPU main memory.

CUDA is a data parallel SIMT architecture, in which the same programmer-defined kernels execute on all launched threads. These threads are launched in batches of blocks and grids, where blocks are collections of threads and grids are collections of blocks [1].

2.5 Related work

Previous work was done for extending different aspects of the Concurrent Collections model. We first worked on developing GPU extensions in the Java flavour of CnC, thus creating CnC-CUDA and obtaining encouraging results [1]. This was a strong motivation for dropping the overhead introduced by making native calls from the Java language and defining steps in a lower level language, more flexible in terms of communication with devices, while keeping the ease of use introduced by CnC. Other extensions include obtaining performance through determining streaming patterns [37].

An active area of research is that of developing programming models to make full use of a set of heterogeneous hardware resources including one or more of CPUs, GPUs, FPGAs or PPUs and SPUs in IBM's Cell processor. Several models have been developed that attempt to abstract the hardware resources for the user, provide an easy to use model while achieving better performance than homogeneous architectures. OpenCL [7] has proposed introducing a standard for heterogeneous programming, so that programs written using the OpenCL API could run on both CPUs and GPUs. The standard was implemented by different graphics card vendors and seems be gaining favour in the community, however, it is a low-level model that relies on the programmer to understand low-level hardware details. Another major drawback that will need to be addressed is the capability of running on different kinds of GPUs within a program.

There has been extensive work on extending existing languages for hybrid execution. Many have proposed using OpenMP-like constructs. StarSs ([38]) is an example which proposes the use of pragmas to split a program into tasks and define data dependence requirements. This does not address the usability problem for domain experts and it restricts parallelism to forall statements rather than the arbitrary graphs that CnC can define.

StarPU [39, 40] is a another runtime system designed for hybrid CPU and GPU execution. It is similar to our work on affinity (Chapter 6) in that a task can be specified to be "fit" for running on the CPU or GPU (".where = CORE|CUDA") and the choice can be made by runtime systems. In CnC-HC we use a HabaneroC runtime call to determine the nature of the current thread to decide which code to run, whereas in StarPU the information is retrieved from a structure making the device-function associations. StarPU also implements a complex memory model in order to manage coherence among CPU and device. This is not needed in CnC as all steps are functional with respect to their inputs and the model is dynamic single assignment, so the data can only be written once. StarPU has extensions for the GPU and Cell but does not address FPGAs. On the opposite side, there are systems that address just CPUs and FPGAs [41], and companies such as Convey [2] which enable the usage of the C language using a shared memory view for accessing FPGAs. Aside from addressing multiple architectures, we also focused on offering high usability by allowing users to specify their algorithm at a high level by means of tasks dependences and automatically generating most of the code necessary to enable parallelism.

A less recent programming model is Merge [42], which distinguishes itself by offering a predicate dispatch-based library system for managing and invoking function variants for different architectures. Merge is a high level language based on map-reduce, having a compiler and runtime responsible for dynamically selecting the best function variant depending on the program and the machine it is run on. Merge is built on top of EXOCHI[43], which allows the available execution units to execute in the same address space, thus making it easy to involve all of them in the computation. This facilitates scheduling (only one global scheduler is needed) and cooperation for Merge. However, in order to use Merge, a new processor/accelerator would need to support the EXOCHI interface, then to be assigned a unique identifier used for predicate dispatch and finally implement function variants for it. What is more, the EXOCHI model has several other drawbacks: it relies on the programmer to ensure cache coherency, it has specific requirements that limit the set of architectures that could use the cache coherence mechanism they provide and it makes strong assumptions about hardware capabilities (e.g., the existence of a TLB on EXO sequencers).

At the other end of the spectrum we find domain specific languages (DSL), which tackle the problem of running on heterogeneous hardware by extending a high-level language like Java. The most recent ones are the DeLite project from Stanford [44, 45] and the Lime language from IBM Research [46]. These approaches are much more complex as they need a source-to-source compiler to convert the conventional code into “native” code (= suited for a particular application) such as C, OpenCL, CUDA or Verilog and a runtime that can interact with the native libraries generated. Work done on previous so called domain specific virtual machines (DSVM) like RapidMind (acquired by Google) or PeakStream (now used in Intel’s Array Building Blocks), targeted lower level languages such as C and C++, but in a similar way: the computation would be expressed in a data-parallel manner and compiled for different architectures.

The DSVM's and DSLs approach in general is to enable the same code to be compiled for different processing units, thus missing on many optimizations and lowering performance. They can be more restrictive than what this work proposes (they don't allow an existing algorithm in a different language to be "plugged-in" the application and interact with the rest of the steps, even if that language is supported as a "native" variant). On the other hand, compared to a multi-variant language such as Merge, DSVMs certainly offer a significantly simpler programming model. Quilin [47] is a system which combines the two approaches. It proposes automatic mapping of applications on heterogeneous architectures by using a training run to collect runtime profile information and then use this to divide computations within an application between a CPU and GPU, while also allowing filters to have CPU and GPU hand coded variants. Their approach yields good performance improvements as well as reduction in energy consumption. One assumption Quilin makes for determining the best mapping is that the step's performance varies linearly with its input, which is not true for all applications; also, it only targets CPUs and NVIDIA GPUs, restricting the languages in which a user can hand-code their computation step.

An improvement no other language proposes and that our work brings, is that CnC introduces another level of abstraction where domain experts will simple write "this_step writes into this_item_collection" in a simple format:

(this_step) -> [this_item_collection], before moving to such language details as declaring variables, understanding the syntax, etc.. This approach also offers a highly desirable property: easy maintenance, as previous algorithms or computation steps can be reused by knowing their behaviour, not the implementation details, and they can also be replaced with no effort even by algorithms implemented in other languages if the application was initially designed as a dependence graph. In addition, none of the languages presented here can express arbitrary task graphs as CnC, thus offering more generality than other models.

The approach taken in this work is a mix of some of the best parts from the languages presented so far. We introduce ranges in an attempt to describe data-parallel computations, which could map a computation to the GPU transparent to the user by auto-generating the host-device communication. This feature can limit performance as the application would not be tuned for the GPU and also limits the applications that can be written using it to those with data-parallel constructs, yet it would be easy to write by a programmer, with good speedup compared to the CPU only run. On the other hand, we also allow multiple variants of the same step for different architectures and introduce step affinity for different devices in order to dynamically choose where a step is run based on available implementations, hardware availability and knowledge of where the step may run faster.

We conclude that this research builds on and improves past work because the individual tasks in our data-flow programming model can be still expressed in almost any language (including explicitly parallel languages), providing flexibility and reuse of the existing algorithms.

Chapter 3

Graph language extensions for CnC-HC

The CnC model was designed to enable the separation 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. However, in the current high-level definition of a program described by a domain expert, the provided information on the data flow is rather vague. The data in a CnC program is grouped in collections, called item collections. The information one can draw from a CnC specification is from what item collection is a step reading and into which item collection it is writing. However, there is no means of knowing from that specification which item it should read so such information wll need to be given also by the domain expert inside the step code. As a result, the domain expert would have to get familiar with an API provided by the CnC runtime in order to make accurate function calls for reading and writing items inside a step.

This work offers an alternative, which removes the need for a domain expert to learn such implementation details, thus increasing programmability.

We introduce *tag functions*, a means of accurately specifying the dependences on specific items belonging to an item collection and between instances of the same or different steps. In this work, a tag function maps a step's unique identifier, the control tag, to the tags (keys) of the items read or written through Get and Put operations respectively, and control items written that enable other steps. We also introduce a way to specify *ranges* of items for item and control collections, by means of two tag functions representing the start and end of a range.

These additions to the CnC graph specification provide support for automatic code generation of Get functions that retrieve the data required by a step - as well as suggested code for the items written and steps prescribed. What is more, the addition of tag functions makes the program flow more analysable opening new research opportunities, while introducing ranges can be used to extend CnC for other SIMD/SIMT architectures like the GPUs, by allowing parallelism within a step.

The extensions we propose in this work make clearer the separation between the problem definition and its implementation, ensure an easy way of programming through code generation and are further motivated by our goal to design CnC to be mapped onto heterogeneous hardware platforms such as the Customizable Heterogeneous Platform (CHP) being developed in the NSF Expeditions Center for Domain-Specific Computing (CDSC [12] [13]).

3.1 Tag functions

To better understand the notion of tag functions in the context of the CnC graph let us first look at the simple graph specification example shown in listing 3.1, an example used in the CDSC project.

In this example, line 1, using angle brackets, declares a tag of type int. In the classical CnC [9] representation of tag functions, the type can be any user defined type, which can be detailed as being composed of multiple components. For example, the declaration on line 1 could also be written as: $\langle$ *1DType reg_tag: pos* $\rangle$, where the tag would be of type 1DType, having one component called pos. Similarly more complex types can be defined such as: $\langle$ *3DType position_in_3D: row, col, depth* $\rangle$ which declares a type that will have 3 components.

```
1 < int reg_tag > ;
2 [ float*** input_matrices ] ;
3 [ float*** output_matrices ] ;
4 <reg_tag> :: ( registration );
5 [ input_matrices ] -> ( registration ) -> [ output_matrices ];
6 env -> <reg_tag>, [ input_matrices ];
7 env <- [ output_matrices ];
```

Listing 3.1: Example 1: CnC graph without tag functions

Lines 2 and 3 declare two item collections, both being 3D matrices of type float. This means that each item found in either of the item collections will be a matrix and each such matrix will be uniquely identified by a tag within the item collection.

Following the item collection declarations is a step prescription (line 4). This is equivalent to saying that when an tag is put in $\langle$*reg_tag*$\rangle$ tag collection, a step identified by the same tag can start to execute. In the CnC model steps are functional with respect to their inputs and outputs.

The next line (line 5) defines the producer-consumer relations for the step *registration*. We also call these relations "execution relations". Each instance of the registration step can read data from the item collection *input_matrices* and can write data into the item collection *output_matrices*. Both relations have full generality, meaning the step can read zero, one or more matrices and can also write a variable number of items as its output.

The last two lines in the graph specification (lines 7 and 8) define the I/O relation with outer environment, which will be responsible for writing into the *input_matrices* and *reg_tag* collections and read from the *output_matrices*.

```
< int [1] reg_tag > ;
[ float*** input_matrices ] ;
[ float*** output_matrices ] ;
<reg_tag> :: ( registration );
[ input_matrices : k-1 ] -> ( registration : k ) -> [
    output_matrices : k-1 ];
env -> <reg_tag>, [ input_matrices ];
env <- [ output_matrices  ];
```

Listing 3.2: Example 2: CnC graph with tag functions

Let us now look at the same example with some of the extensions added in this thesis (listing 3.2).

An addition to the classical definition of a CnC graph [9] is the introductions of tag sizes in the declaration of a tag, which defines how many components a tag has. For the code to be accurately generated, we need to know the type of each tag. Currently, both the translator and the code generator support any tag type, but the current method of creating a tag from multiple components is defined to accept just a subset of tag types: integer. The format of the tag may be extended to support tuples of different types, for example: (int, char, double, char*). Offering something transparent for the user for general tag types, in the current format, where tags are strings, requires additional compiler support (e.g., a call to "createTag(int, char, double)" being transformed into "initTag;addInt(int);addChar(char);addDouble(double)"). A more general and efficient alternative is to change the CnC runtime implementation such that tags become arrays of unions. We reserve this for future work, though the programmer can obtain that functionality by marshalling and unmarshalling any tuple into a tuple of ints or creating his own tag creation method.

Item collections are expected to be declared with the correct element type, either a primitive or a user defined type. All items in an item collection are stored internally as pointers but they can be declared both as pointers and non-pointer types (a primitive value will be “boxed” using a pointer).

Instances of computation steps are uniquely identified by tags, which are tuples of symbolic names chosen by the one writing the CnC specification. In a prescription relation, such as the one on line 4 in listing 3.2, the control collection (reg_tag) will contain a set of tags t which enable the execution of steps identified by t. Consequently, the format of such a tag does not need to be explicit in a prescription relation. This is synonymous with saying that the tag function of a tag t in a control collection is the identity: f(t)=t. Thus, is our implementation of the CnC model, step prescription are kept in the same format as in the classical CnC graph, which also preserves the previous behaviour that a tag can prescribe multiple steps and a step can only be prescribed by one tag. If the user chooses to add tag functions to step prescription as well, the translator will issue a warning, keep the implicit identity function relation and it will ignore the unnecessary tag functions.

In contrast, in an execution relation, a step can read and write multiple inputs and outputs respectively and these can vary greatly depending on the application. To provide flexibility in defining dependences, the items to be read and/or written by a step are defined as functions of the tag's components, thus the name tag function. In the above example, on line 5, the *registration* step is identified by a tag with one element, by means of variable k. The step is defined to get an item from item collection *input_matrices* and the function to find that element is $k-1$. This allows easy code generation without the need of computing the inverse of a function in step executions. Consider if in the above example, the step was identified by *k*2* rather than k. This would mean that the correct item to read for any step with tag T is (T-1)/2 obtained by applying the function defined for input *input_matrices* (f(x) = x-1) to the inverse function of the step identifier (f(x) = x/2). Our approach is the most intuitive way a user would define the task graph, by relating both the inputs and the outputs of a step to what uniquely identifies a step (its tag components).

In the general case of a tag declared to have N dimensions, a get will be performed on a collection: $[collection\_name : tagfunction1, tagfunction2, ..., tagfunctionN]$, where the functions accept the 4 basic operators (+,-,*,/) and any variable name, yet the variable names should relate to the variables found in the steps list for the generated code to be correct. All items read and written by the step, as well as the tags put to prescribe other steps, can be defined with functions relative to the step's control tag and constants. In addition, the step's outputs (both data and control items) can be declared as relative to the input data, with the indexing operator (e.g.: input[tag component]). In the above example, the step *registration* has only one output. It will write an element in collection *output_matrices* and this element will be uniquely identified by the key $k - 1$, representing a tag function. Extensions for more complex functions and user-defined functions are a subject for future work.

For a better understanding of how tag functions can be used, we present a second example, one simulating the execution of a loop with a variable stride.

The example in listing 3.3 contains only one type of computation steps called "loop" (the only key word in a CnC graph is *env* symbolizing the environment), which will in fact execute the body of a while loop, translating the condition to continue to the next iteration into a prescription of another instance of the step "loop"; the tag thus becomes the iteration number. The loop's exit condition is dependent on

```
< int [1] step_tag > ;
[ int size ];
[ int jump ];
[ float*** input_matrices ] ;
[ float*** output_matrices ] ;
<step_tag> :: ( loop );
[ size : 0 ], [ jump : k ], [ input_matrices : k ] -> ( loop : k ) -> [ jump : k + jump[k] ], [ output_matrices : k + jump[k] ], < step_tag : k + jump[k] >;
env -> [ start : 0 ], <step_tag : 0>, [ input_matrices : 0 ];
env <- [ output_matrices ];
```

Listing 3.3: Example 3: CnC graph simulating a loop

the value of size[0], where size is a collection with only one item. Each step with tag = k, will do some computation on the input matrix obtained from the item collection *input_matrix* and may enable the execution of the next loop iteration which is computed using the data read from the *jump* item collection. We notice that the item in which the step writes the output (*output_matrices*) is identified by a tag relating to the step's unique identifier (k) and to the input item jump[k].

This example shows clearly that a step can choose whether to write or not in the outputs (item or tag collections) declared in the graph specification. This loop will eventually terminate when some condition regarding k and size[0] is met (e.g. k>=size[0]) which is synonymous with concluding that we cannot generate *Put* functions or *Prescribe* functions with the same certainty we can generate the *Gets*. For program completeness we require the user to write the tag functions both for inputs and outputs, however the code generated for outputs will be considered as a suggestion rather than a requirement. We believe this design choice will help the user since he would only have to reason about dependences rather than writing code (the suggested code will need very little modifications, if any).

We must mention that we kept backward compatibility to the classic CnC model, such that if tag sizes or tag functions are omitted, the translator will generate suggested code in comment detailing how the Gets should be performed. Writing the proper code remains the users responsibility in this case, as in the original CnC specification.

3.2 Ranges

Another addition to the CnC graph specifications is the concept of ranges. Several applications require that several pieces of data in a continuous range be either read or written or both. For a fix number of items known at compile time, this could have been represented by scalar tag functions. For example:

$(\mathit{some\_step} : k) \rightarrow \langle \mathit{step\_tag} : k \rangle, \langle \mathit{step\_tag} : k+1 \rangle, \langle \mathit{step\_tag} : k+2 \rangle$.

However, if the number of items read or written is determined at runtime, we need a new way of way of allowing the user to provide this information. Ranges are declared in the CnC graph file using braces and two dots to separate the beginning and end of the range: { *start_range* .. *end_range* }.

Let us consider the example in listing 3.2 once again. In essence, the graph in the example models the execution of multiple instances of fluid registration [48] with no dependences among each other. One will imagine that after defining this graph, the user would have to write the Main function to Put a set of tags, each of which will start an independent step. Considering this, let us create another example that

```
< int [1] singleton_tag > ;
< int [1] reg_tag > ;
[ int size ] ;
[ float*** input_matrices ] ;
[ float*** output_matrices ] ;
<singleton_tag> :: ( singleton_step );
<reg_tag> :: ( registration );
[ size : 0 ] -> ( singleton_step : t ) -> <reg_tag : { 1 .. size[0]+1 }>, [ input_matrices : { 0 .. size[0] } ];
[ input_matrices : k-1 ] -> ( registration : k ) -> [ output_matrices : k-1 ];
env -> [ size ], <singleton_tag>;
env <- [ output_matrices ];
```

Listing 3.4: Example 4: CnC graph with ranges

would make the Main call simpler for the user, by using ranges.

In the example from listing 3.4, the user main function would just have to just prescribe a single step, the *singleton_step* and write the code of initializing the inputs and starting all the registration steps within it. We have shown a possible use case of ranges, but in practice there are several others such as split join operations and reducers. In addition, specifying that a step reads a range of items may be extended to mean that the computation performed is data parallel and thus fit for SIMT execution.

In this chapter, we presented how tag and item collections allow and need the ability to specify tag functions for automatic generation of high level operations for inter-task communication. We will go into more detail of how the code is generated from these specifications in chapter 4.

Chapter 4

Compiler support for CnC-HC

In this chapter we present the CnC-HC translator, a tool that generates the necessary code to link the user step code to the CnC runtime as well as suggested step code regarding the items to be put or steps to be prescribed, all these based on the CnC graph specifications.

4.1 Software lifecycle with CnC-HC

Let us first discuss what are the steps that a user has to make in order to write a program using CnC-HC. Figure 4.1 presents the implementation flow for a CnC-HC program.

The user will first decompose his algorithm into steps and write a graph specification as described in the previous chapter. The translator is a tool which will generate a series of “glue-code” to enable transparent parallelism for the user based on the guidelines he provided in the graph file. What is more, it will offer step stubs as suggested step code to handle the items a step puts or the steps it may enable. In order to ensure an easy build, the translator will also generate a makefile.

The user can then proceed to writing the code for each of the computation steps and run the makefile to build his application. This will use the HC compiler and gcc compiler and generate an executable. If additional libraries are required, they can be easily added in the provided Makefile.

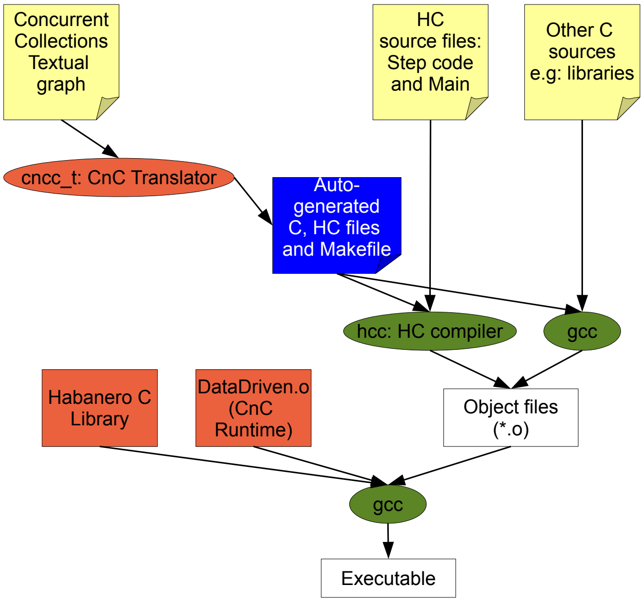

Concurrent Collections Textual graph
HC source files: Step code and Main
Other C sources e.g: libraries
cncc_t: CnC Translator
Auto-generated C, HC files and Makefile
hcc: HC compiler
gcc
Habanero C Library
DataDriven.o (CnC Runtime)
Object files (*.o)
gcc
Executable

Figure 4.1 : CnC-HC Build Model

A common scenario in developing a program is making changes to the initial specification. In our case, if the user were to make changes to the graph file and rerun the translator, only the code linking his steps with the runtime will be overwritten to match the new dependences. Thus, if the computation had been written by a programmer prior to running the translator, his code will not be overwritten, but preserved intact.

4.2 Auto-generated executable code

The need for generation of "glue code" is due to application dependent components like item collections and steps but also to the usage of the C language. Some of the artifacts that we need to generate include: an unique identifier for each step, methods that initialize or delete a CnC graph, methods that launch the execution of other steps

```
#define Step_registration 0
struct Context{
    ItemCollectionEntry ** input_matrices;
    ItemCollectionEntry ** output_matrices;
};
typedef struct Context Context;
```

Listing 4.1: Auto-generated code defining a unique step id and the context structure for the example in Listing 3.2.

and methods that link the user code to the CnC runtime.

For the example in Listing 3.2 the auto-generated code declaring the step identifier and the item collections is presented in Listing 4.1.

The generated methods can be divided into two parts: the methods available to the user as APIs and the methods generated to transparently enforce constraints as specified by the CnC graph. Let us describe these below using code samples for the example in Listing 3.2.

The auto-generated methods included in the user API are: initGraph, deleteGraph, prescribeStep.

The pair of methods initGraph and deleteGraph are called to instantiate and clear the allocated space respectively, for a CnC graph. The initGraph method will reserve the space for all item collections used in the graph and perform other initialization steps if required, while its complement method will free this allocated space. Each item collection is implemented as a hashtable with a fixed number of buckets (TABLE_SIZE) defined by the runtime. Here is some sample auto-generated code

```
Context* initGraph(){
    int i;
    Context* CnCGraph = (Context*) cnc_malloc (1 * sizeof(Context));
    CnCGraph->output_matrices = (ItemCollectionEntry**) cnc_malloc(TABLE_SIZE * sizeof(ItemCollectionEntry*));
    for(i=0; i<TABLE_SIZE; i++){
        CnCGraph->output_matrices[i] = NULL;
    }
    return CnCGraph;
}
void deleteGraph(Context* CnCGraph){
    cnc_free(CnCGraph->output_matrices, TABLE_SIZE * sizeof(ItemCollectionEntry*));
    cnc_free(CnCGraph, 1 * sizeof(Context));
}
```

Listing 4.2: Auto-generated code initializing and disposing of a graph.

declaring one item collection: Listing 4.2 (we extracted only the initialization for one of the collections for brevity).

The method prescribeStep instantiates a new step, with the corresponding tag and, depending on the chosen runtime, prepares for step execution accordingly (either begins to check dependency satisfaction or sends the step straight for execution taking the chance of the step to fail if data is unavailable; the runtimes are details in chapter 5). An extract of the code generated for this method can be seen in listing 4.3.

As the generated files are independently compiled, we use a HabaneroC pragma

```
#pragma hc suspendable
void prescribeStep(char* stepName, char* stepTag, Context* context){
    Step* step = cnc_malloc(sizeof(Step));
    step->tag = stepTag;
    // ... step initialization
    if(!strncmp(stepName, "registration\0", 12)){
        step->stepID = Step_registration;
        #ifndef _Data_Driven_Runtime
            dispatchStep(step);
        #else
        registration_dependencies(step->tag, (Context*)step->context, step);
        int status = resolveDependencies(step);
        if(status == CNC_SUCCESS)
            dispatchStep(step);
        #endif
        return;
    }
    printf("Step %s not defined\n", stepName);
    assert(0);
}
```

Listing 4.3: Auto-generated code for prescribing a new step.

```
1 #pragma hc suspendable
2 void* registration_gets(char * tag, Context * context, Step* step
    ){
3     int k = getTag(tag, 0);
4     float*** input_matrices0;
5     char* taginput_matrices0 = createTag(1, k);
6     CNC_GET((void**) & (input_matrices0), taginput_matrices0,
          context->input_matrices, step);
7     registration( k, input_matrices0, context );
8     return 0;
9 }
```

Listing 4.4: Auto-generated code for getting the input data for a step.

(line 1 in listing 4.3) to mark as interruptible, methods that can spawn tasks and therefore need to be treated differently by the source to source compiler (the details are beyond the scope of this work). We also note on lines 9, 11, 14 calls to auto-generated methods and on line 12 a call to the CnC runtime.

The auto-generated methods that are needed to link the user code to the runtime are: stepName_dependencies, stepName_gets, dispatch. For each step we will generate a pair of functions called stepName_dependencies and stepName_gets. The former will make runtime calls to store the list of all dependences needed to be satisfied for a step to execute and is only used in the Data Driven runtime. For the graph in Listing 3.2, the method performing the Get calls is presented in listing 4.4. Here, we highlight three runtime calls on line 3, 5 and 6 and a call to the user-written step code on line 7.

```
void dispatchStep(Step* step){
    switch(step->stepID){
    case Step_registration: async IN(step){ registration_gets(step->tag, (Context*)step->context, step); }; break;
    default: CNC_ASSERT(0, "Attempt at executing an unknown step!");
    }
}
```

Listing 4.5: Auto-generated code for asynchronously spawning a task.

Calling the dispatch method (listing 4.5) is equivalent to asynchronously executing a step. The generated code for this implies using the async construct provided by HabaneroC to asynchronously launch the execution of a step. This means running the auto-generated code for getting the data needed by a step and then doing the actual user code call. The runtime chosen will determine whether the creation of an asynchronous launch will be equivalent to a successful step execution or a possible failed attempt that will need to be retried. For the scheduling of the tasks launched using the async keyword, CnC will be using HabaneroC's work stealing mechanism. The policy (work first or help first) can be chosen through a command line argument to the executable.

4.3 Auto-generated helper code

The translator will use the information provided by the user in the graph file to also generate helper code for the steps. For example if a step was defined to possibly create a control item (tag), the step code will include a comment of how this would translate into code. More precisely, the generated code will call prescribeStep on all steps prescribed by that tag. The user can use this code as he/she sees fit, though in most cases simply uncommenting the suggested code and making minor changes will give the desired user code. In listing 4.6) we present some sample suggested code for a more complex example (the graph from listing 3.4). The step stub for the singleton step contains two comment blocks. The first one refers to data items written, while the second contains the prescription code. The graph specification defined the step to write a range of items, consequently, the generated code will have a for loop with the number of iterations equal to the elements in the range, and it will make the runtime call that writes an item on each iteration. If the specification was accurate, the user would simply have to uncomment this code block and read from an external source, initialize or compute the items to be written. Similarly, the suggested code prescribing the registration steps would be uncommented and, in this case, no additional changes would be needed.

As mentioned before, running the translator on a graph file will also generate a Makefile for building the application using the generated files. This can be altered by the user to meet the application needs, such as adding other source files or libraries.

```
void singleton_step( int k, int size0, Context* context){
    /* This is suggested code
    int _index1_0;
    float*** input_matrices1[size0 -0];
    for(_index1_0 = 0; _index1_0 < size0 - 0; _index1_0++){
        // allocate memory if necessary and fill in values to put here
        char* taginput_matrices1 = createTag(1, _index1_0 + 0);
        Put(input_matrices1[_index1_0], taginput_matrices1, context->input_matrices);
    }
    */
    /* This is suggested code
    int _index2_0;
    for(_index2_0 = 0; _index2_0 < size0; _index2_0++){
        char* tagreg_tag2 = createTag(1, _index2_0);
        prescribeStep(”registration”, tagreg_tag2, context);
    }
    */
}
```

Listing 4.6: Auto-generated suggested step code for handling data and control output edges.

Chapter 5

Runtime support for CnC-HC

The CnC-HC model was developed on top of the HabaneroC (HC) programming language, and it uses the async and finish constructs available in HC. However not all dependency graphs can be expressed using only async and finish, so we need a CnC runtime. In this chapter we will first present previous approaches of expressing arbitrary task graphs using the async construct. We will then detail two implementations of the CnC runtime in HC. The first one was developed to reflect the behaviour of a previous CnC runtime systems [9], using the best performing scheduling policy: restart and replay, which essentially will abort a step when data is not present and restart the step when the data becomes available. The second one focuses on spawning a step only when all data is available, aiming to eliminate the overhead of repeated task creation when necessary, yet introducing the possibility of loss of parallelism in the process. Both runtimes use the tag functions introduced in this work and take advantage of the code generated by the translator for making the writing of the step code significantly easier than in previous CnC implementations.

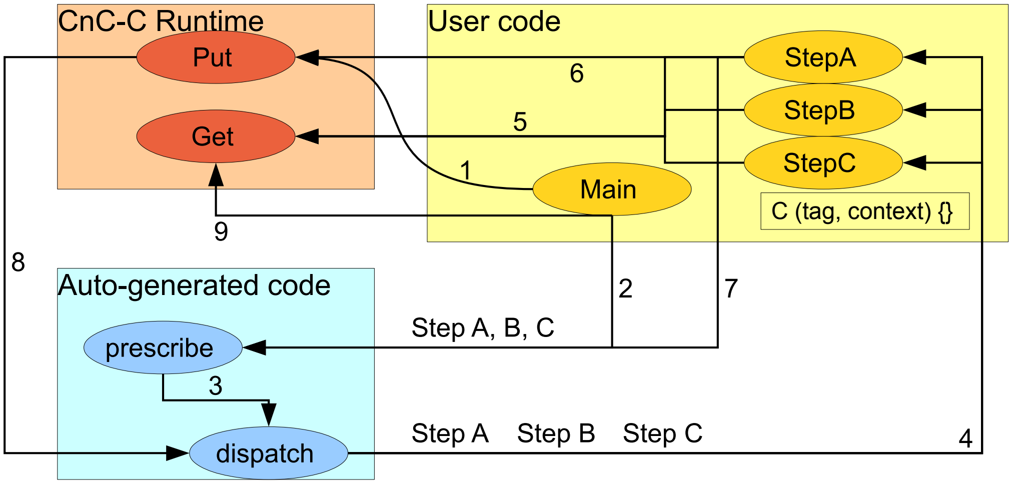
5.1 Previous runtime approaches

In this section we present brifly previous work on the runtime approaches for a Java implementation of the CnC model. The Rice CnC-HJ variant, mentioned in section 2.3 and table 2.1, takes a classical approach on the model, where within each step, the user will call the Get methods for reading the input data, then perform the computation and finally write items through Put calls or enable other steps for execution. The runtime provides three scheduling policies: Blocking, NonBlocking and RestartAndReplay.

In the Blocking policy, once a Get fails to find an item, the thread which executed the Get blocks waiting for the data to become available and a new thread is spawned in its place to avoid the situation when all threads block. This causes considerable overhead as the newly created threads are not terminated when others are unblocked. What is more, for large applications, if too many threads block, the execution will fail due to the limited Java thread pool size.

The NonBlocking policy introduces a ready function associated with each step that the user has to provide. The ready function signals whether the step's input data is available or not. The runtime will continuously check for each prescribed step by calling this function, whether the input data is present, in which case it will spawn (async) that step. This ensures that all Get calls within a step will be successful. Though the NonBlocking policy performs better that the Blocking policy and resolves the problem of creating additional threads, it still incurs considerable overhead from polling using the ready function.

According to previous work [9], in most cases the best performing policy is the RestartAndReplay. This method allows steps to spawned as soon as they are prescribed, so calls to the read input items may fail to find the data. In this situation, the runtime is engineered to abort execution of the current step and store it, as a closure, in a queue waiting for the item it failed on. When the missing item is written, the step that Put it will check the associated queue and respawn all the previously failed steps. As a result every Get can fail at most once, so it is possible to repeat

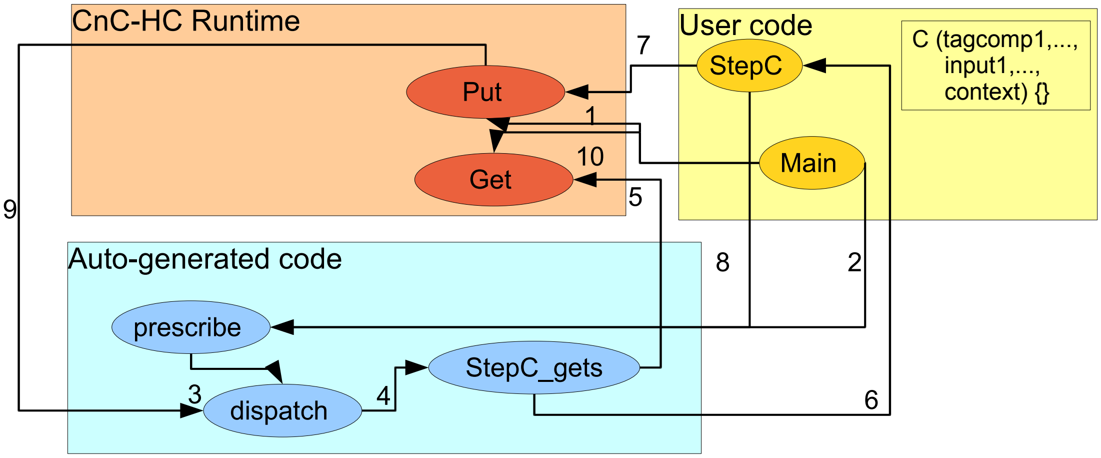


1. Main puts input data in Item Collections
2. Main sends a request to create and execute one or more steps
3. After being created, steps are dispatched to the scheduler for execution
4. Steps begin to run asynchronously
5. Steps try to get the data they need, may fail if data is not available in which case it will put itself in a queue waiting for that data.
6. Steps may put data in item collections
7. Steps may prescribe other steps
8. A put on a piece of data may trigger the call of a step if there is one waiting on that data.
9. When all steps have finished Main gets the output data.

Figure 5.1 : Hypothetical C implementation of a CnC Runtime, using the Restart and Replay policy.

the Gets up to the number of items the step needs before being able to execute.

With the objective of enabling heterogeneous architectures, we pictured in figure 5.1 a C-base implementation of CnC which would implement the Restart and Replay policy. This figure simply shows how the current methodology would translate into a concrete implementation. However, in CnC-HC we went one step further. As mentioned in chapter 3, our goal was to ensure ease of programming in a low level language such as C, for which reason we introduced tag functions and ranges in the specification language. This allows us to engineer the runtime to take advantage of the auto-generated code. We will present how we do this in the next section.



1. Main puts input data in Item Collections
2. Main sends a request to create and execute one or more steps
3. The step is dispatched to the scheduler for execution
4. Step begins to run asynchronously
5. Attempts are made at retrieving the needed data. If all necessary data is not available, the step puts itself in a queue waiting for the missing data and the execution is aborted for this step.
6. If all data is available, the user step code is called with that input data.
7, 8. Steps may put data in item collections(7) and may prescribe other steps(8)
9. If there are steps waiting on a data Put, those steps are run asynchronously once again. These steps may once again abort and be restarted if all data is not yet available.
10. After all steps have finished executing, Main gets the results.

Figure 5.2 : Control Flow for the Restart and Replay Runtime

5.2 Restart and Replay runtime

5.2.1 Runtime description

The Restart and Replay Runtime is pictured in figure 5.2. The flow of any program using this runtime is as follows. The Main function written by the user, will instantiate the graph and Put the items it wants to use as inputs in one or more item collections (step 1). It will then start the graph execution by prescribing one or more steps (steps 2). These prescriptions need to be executed inside a finish scope, ensuring that the main program waits for all steps pertaining to the graph to finish executing. After the finish scope, the user can Get the data from one or more item collections as outputs of the CnC execution(step 10). Let us now go into depth into what happens once a step is prescribed.

In CnC-HC, a step is a C function. On each prescription, a step is said to be dispatched or spawned into a new task using the HC async construct (step 3). In previous implementations of CnC, each step could make calls to the Get and Put functions, but according to the CnC semantics, all Gets had to precede all Puts. The reason for this is to ensure that a step can run to completion without interruption once all its inputs are available. It is not considered a limitation, as any step can be split such that the condition is met.

One of CnC-HC's contributions is the usage of tag functions for generating the Get functions (step 4) outside the user code, as well as generating suggested code for items to be put and steps to be prescribed. We use the information given in the graph file to generate the Get functions, removing this burden from the user and also ensuring consistency between the graph specification and the step code. Thus, a step can be viewed as split into two function calls: one retrieving the inputs (step 4) and one executing the computation using them (step 6).

When calling the Get function (step 5), the runtime will check if the item exists (was Put by another step) and if so it returns it. If however the item does not exist, a placeholder for the item will be created and also a queue will be associated with it. The current step will put itself in that queue and then abort (the step function will return).

To ensure light weight synchronization, we only use the "compare and swap" primitive and no blocking operations. For both Get and Put calls, we first need to retrieve the location for the item requested. This implies finding it if it exists through a simple search or else creating one. In the latter case, the placeholder associated with an item is created and initialized, and then an attempt is made to insert it in the item collection using compare and swap. If the attempt fails, then another thread succeeded and we will use the location it created. If however both attempts came from Put calls, then the runtime reports a “Single assignment rule violation”.

When doing a Get, a step either finds the item in the location it just retrieved, in which case it returns it, or else it needs to wait for the item to become available. Once the Get has the location for the item, it can also access the waiting queue it has associated. A step will always enqueue itself at the beginning of the waiting queue through a compare and swap operation. If it fails, there are two possible reasons: a Put was done while the step was enqueuing itself, in which case the item is now available and can be returned, or another step has succeeded in enqueuing itself first, in which case the step will retry to add itself.

When doing a Put, the step will write the item in the location it found and then proceed to respawn th steps that failed on that item. For this, it uses once again a compare and swap to retrieve the queue of failed steps. If the attempt fails, it means some other step successfully enqueued itself in the meantime so a retry is necessary. Otherwise, all steps in the waiting queue will be respawned.

Previous implementations of CnC modelled the prescription of a new step by adding a tag to a control collection. The specifications were designed such that each step prescribed by such a tag collection will start to execute with the tag put in the collection. We model this in a more straight forward way, by directly making a call which enables the desired step (step 8), thus eliminating an additional overhead introduced by control collections.

When a step puts an item (step 7), it will check the waiting queue associated with that item, and if it is not empty it will create an async for each step waiting for that item (step 9). It is thus possible for a step which gets N items to be restarted between 0 - when all items are found the first time - and N times when all Gets fail the first time. This behaviour leverages the restriction of having Gets before Puts, because of the single assignment rule imposed in CnC. If a step were to do a Put followed by a failing Get, on the second run of that step, a second Put will be done on the same item causing a single assignment rule violation. Also, since Puts are the only side effect permitted in a CnC step it is safe to restart the step whenever a Get fails.

The suggested code for items to be Put or prescribed is generated in a comment block that can be edited to accommodate the application needs. A typical example is of a step that prescribes another step if a set of conditions is true (the example in Listing 3.3 is one of them). In such a case, the generated code can be uncommented and used inside a user written if clause. Details on the code generated by the translator were given in chapter 4.

Having understood how the Restart and Replay Runtime behaves, we now motivate the need for the Data Driven Runtime. The aim is to eliminate the need of restarting a step for every failed attempt to get an item. We also note that the creation of an async implies the allocation of a task frame in HabaneroC and the copying of data to that new frame, which can incur significant overhead if done repeatedly as in the case of many failing Gets described above. Further, the abort-restart approach is impractical across heterogeneous processors.

5.2.2 Theoretical guarantees

The CnC model does not guarantee dead-lock freedom. It is possible for step1 to wait for an item produced by step 2 while step 2 waits on an item from step1. However depending on the runtime implementation we can detect such cases with more ease. In CnC-HC we took a non-blocking approach, which means that in the situation described above, both step1 and step2 will be placed into a waiting queue without keeping a thread blocked. The HC finish scope will detect quiescence (there are no more asyncs running in the scope) and the program will terminate. Ongoing work is being done for the runtime to issue a warning that there were steps waiting for items that were never Put. This information can then be used to determine the missing items and resolve the dead-lock.

The first theoretical guarantee we present is described by theorem 5.1.

Theorem 5.1

The Restart and Replay runtime correctly implements CnC *semantics*

Proof 5.1 The CnC semantics state 1) that the Gets must precede Puts, 2) that data is dynamic single assignment and 3) that the model is race-free.

1)The first property is deduced trivially by the fact that we generate a function which reads the data before calling the user step code.

2)To enable the second property, we engineer the runtime to check that an item is not put more than once. This is done by providing light-weight synchronization using compare-and-swap. The key cases that we handle the following. First, if a step tries to perform a Get while another does a Put there are two possible sequences of events:

1. The Get succeeds in creating a place holder for the item and putting itself in a queue waiting for it; it then terminates its execution. In this case, the step doing the Put will fill in the value and respawn the step that aborted.

2. The Put succeeds in adding the item, in which case the Get will immediately

succeed.

The second key case is when there are two threads trying to do a Put on the same item. We distinguish two subcases: one when there was an attempt to get the item previously in which case a place holder for the item exists and another when the task doing the Put needs to create that location. We implemented the synchronization needed to handle both of these situations. The execution terminates with a "Single assignment violation" exception if a Put is attempted twice on the same item.

3) Having proven that the Restart and Replay runtime correctly implements the single-assignment rule we can also deduce that there cannot exist a race for reading an item, as its value will not change once it's written. Thus the runtime exhibits race-freedom. □

A second guarantee is described by theorem 5.2.

Theorem 5.2

The Restart and Replay runtime cannot exhibit live-lock.

Proof 5.2 We prove this result based on the result from theorem 5.1. We know that the Gets are executed before the Puts in any CnC-HC execution. The Get function will abort the step if data is unavailable so the step will no longer be "live". Respawning the step is solely done when that data becomes available. In addition, the dynamic single assignment rule enforces that an item be written only once and never be deleted. Thus it is not possible to have continuous failing on retrieving the inputs which would cause a live-lock. □

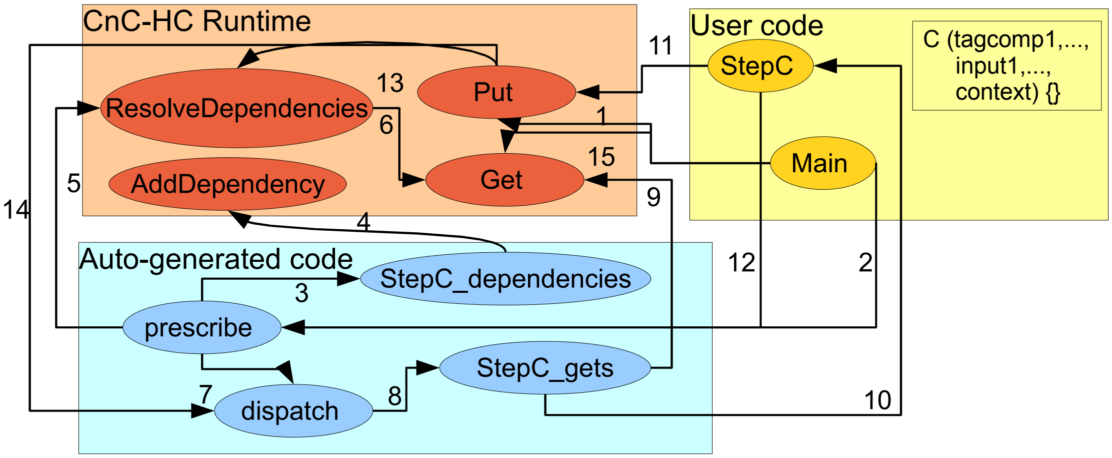
5.3 Data Driven runtime

The Data Driven Runtime was developed to solve the problem of the Get function being called multiple times each time a Step fails to find an item. It also avoids the overhead of creating a new task (async) when this occurs.

In addition to the information we currently store about a step, we need to add the step's data dependences. We use once again the tag functions introduced in the graph file to generate code for an additional function "Step_dependencies" - presented in Section 4.2, to store a record for each datum that needs to be available before the step can execute.

We extended the runtime to support the addition of dependences to each step and we provide an API for adding and checking them. The auto-generated code uses these interfaces to start a step only when all data is available. Instead of starting a step as soon as it is prescribed, we first add a record for each dependence through calls to the runtime, and then we check each of its dependences. The check is interrupted when data is not found and is restarted from where it left off when the missing data is put. Once all required data exists, the step is spawned (dispatched).

Figure 5.3 shows in detail the structure of the Data Driven Runtime system. As before, the Main function will provide the initial input data and will prescribe one or more steps as startup computations (steps 1,2 in figure 5.3). After being created, a step will then have all of its dependences filled in in the auto-generated code (steps 3,4). Further each of these are checked for availability (steps 5,6) and only if all of them are present the step will be asynchronously dispatched for execution (step 7). The required data is retrieved, and as it is known to be available, the attempt of getting the items cannot fail (steps 8, 9). The next step is to call the user code parametrized with the data obtained (step 10). However if all data was not present,



1. Main puts input data in Item Collections
2. Main sends a request to create and execute one or more steps
3, 4. After being created, a step is being filled in each of its dependences
5, 6. The dependencies are checked, if data does not exist the step will be put in a waiting queue waiting for the first item it failed to find.
7. If data exists, the step is dispatched to the scheduler for execution
8, 9, 10. Step begins to run asynchronously(8). Needed data is retrieved(9), then the user step code is called with that input data(10).
11, 12. Steps may put data in item collections(11) and may prescribe other steps(12)
13. If there are steps waiting on a data Put, their dependencies will be checked where they were left off when they failed to find the current item
14. If all dependencies are satisfied, the step is dispatched to the scheduler for execution
15. After all steps have finished executing, Main gets the results.

Figure 5.3 : Control Flow for the Data Driven Runtime

the step will not be launched for execution but it will be put in a queue to wait for the missing data. A step can prescribe other steps (step 12) and also write (Put) data items (step 11). When a step writes a piece of data (step 11) and sees that there are steps that had been waiting on it, it will retry the data dependency checks from where they were interrupted (step 13). Thus a step can be moved through up to N queues, where N is the number of dependences it needs to satisfy. When finally the last of the items is obtained the step can begin to execute asynchronously(step 14).

We have shown so far how the Data Driven runtime removes the overhead of task creation by resolving a step's dependences before it is spawned. Yet for applications where the input data is available on the first attempt, this approach loses some parallelism, because the dependences are checked sequentially. In real applications however, the computational work is significantly larger that reading the input data, so the loss in parallelism wouldn't be noticeable. In our experimental results, we shall conduct a comparison study between the two runtimes to determine the scenarios where one will outperform the other based on the observations we have detailed in this chapter.

The theoretical guarantees proven for the Restart and Replay runtime also hold true for the Data Driven runtime.

5.4 Domain expert versus tuning expert

As we mentioned earlier, the CnC-HC runtime builds on the async, finish constructs in HabaneroC. The only explicit HC construct that the CnC-HC user sees is the finish scope surrounding the graph execution. The asyncs created are transparent to the user, and can be found in the auto-generated code. We offer the possibility of generalizing this approach by allowing the more experienced user to use async-finish constructs within the step code. A tuning expert who knows how to improve application performance by using async-finish explicitly within steps, can take advantage of this feature. Other forms of parallelism can also be used within a CnC step if it is designed to run on a specific device. For example, a CUDA kernel designed for the GPU or a Verilog module written for the FPGA, as will be demonstrated later in our experimental results.

Chapter 6

CnC-HC-Hybrid: A declarative language for heterogeneous architectures

6.1 Overview

With architecture design moving towards a heterogeneous approach and co-located on-die CPU-GPU soon to become a reality, programming models need to reinvent themselves to be able to take advantage of these hardware resources. We propose using the data-flow model introduced by Concurrent Collections as a foundation since it is a model that was proven to have large generality and also a series of desired properties including determinism, race and live-lock freedom [9]. The extension we propose enables the use of a Graphical Processing Unit (GPU), by extending the language specifications to include a definition for GPU steps. This information can be used to generate the required code for transferring data to and from the device, launching and waiting for the execution. Further, we add more generality through the notion of affinity, which enable steps to run on multiple locations which can include CPUs, GPUs and FPGAs. Changes to the CnC translator and to the HabaneroC runtime enable us to obtain very good performance for applications that can take advantage of such architectures.

6.2 Language extensions

We designed two approaches for specifying device steps, one targeting domain experts and the other tuning experts. Further work can take advantage of both to provide a mechanism that offers generality with respect to the available hardware.

Our first approach required extension to the specifications language in order to specify steps which would map well to a graphical unit. The notation for GPU steps uses braces instead of parentheses as follows.

```
<reg_tag> :: { registration };
```

This change was added to the language grammar to enable GPU step recognition. We will discuss in the next section how the translator uses this information to generate special linkage code between the user step code and the CnC runtime.

The second approach was motivated by the CDSC project [13], where applications needed the possibility of executing some steps on accelerators such as GPUs or FPGA and relying on the user to write the appropriate CUDA or C (calling libraries that access the FPGA) step code. We decided to see how such a functionality would work for a model like CnC.

We applied the CnC model in practice in the medical imagining domain, enabling the domain experts to reuse their previously implemented computational kernels. Certain kernels were only implemented on a particular platform, such as an FPGA. In contrast, other kernels had multiple implementations and could run on more than one platform, for example on a CPU and a GPU, but not on an FPGA. From our preliminary experiments, we also knew that some steps would perform significantly better on a particular platform. Thus, we introduced the notion of *affinity* into the CnC graph specification to allow tuning of an application based on knowledge of where an algorithm can run and where it runs better if multiple variants are available. In this thesis, we use the term “affinity” to indicate a scheduling priority rather than an indication of locality. A snippet of the graph from a real application - the medical imaging pipeline detailed in chapter 7 - can be seen in listing 6.1.

```
1    <denoise_tag>::(denoise @ CPU = 20, GPU=10);
2    <reg_tag> :: (registration @ GPU = 5, FPGA = 10);
3    <seg_tag> ::(segmentation @ GPU = 12);
```

Listing 6.1: Step prescriptions with affinity for the medical imaging pipeline

Medical images are sent through the pipeline of image processing kernels (denoising, registration and segmentation), resulting in an image that can be interpreted by a medical expert. All the steps in the above example are image processing steps executing in sequence: denoising makes the image clearer, the registration step compares the image to previous scans and finally the segmentation step localizes the desired area (in our case, a tumor or an aneurysm). Line 1 defines the denoising step, which can be run on both a CPU and a GPU but is more fitting for CPU execution. Line 2 defines the step executing registration, which can execute on a GPU and on an FPGA but is better suited for FPGA execution. Finally, line 3 defines the segmentation step, which only has a GPU implementation.

Currently, the values assigned as affinity are considered for prioritizing which device a step should first attempt to execute on; the relative quantitative differences are used to enable the “device worker” to use a small look-ahead and choose a task which needs to be executed sooner based on its higher priority. In this particular above example, it may be desirable to execute the denoising steps with a higher priority, in order to enable the registration step for that image. Similarly, we may want to execute the registration as soon as possible, so that the segmentation can start. However, if we had a single GPU, note that the CPU and FPGA respectively can also execute the first two stages, while the last stage must run on the GPU. As a result, we have chosen the last stage to have the greater GPU affinity. We also notice that the first two stages run better on other platforms (CPU and FPGA respectively), motivating a choice of affinities as defined by the example.

In general, a "tuning expert" would need to do the analysis we described above and choose appropriate affinity values. Thus, the affinity notion can be viewed as a tuning annotation [49] which we provide as a means of, not only enabling the mapping of an application to a heterogeneous architecture, but also tuning it according to a set of constraints (these can include throughput, energy efficiency, latency).

We are currently also exploring opportunities to define a quantitative notion of affinity in which affinity values are assumed to be proportional relative to the expected performance on a given device (where a bigger number implies better performance). This feature would allow the runtime to perform even better informed scheduling decisions and possibly improve the results. In addition, future work would involve adjusting the affinities at runtime based on information retrieved from profiling during the run.

There has been previous work on tuning annotations in the CnC model [49], but they do not involve extensions for specifying affinities for heterogeneous architectures.

6.3 Code generation

Let us first describe code generation changes for the first approach presented in the previous section. The first difference is that launching a device step will use the place feature available in HabaneroC. Here is an example:

```
async (gpu_place) { step_code() }
```

This async statement is generated by the translator and is transparent to the user. The initialization of the place is also generated by the translator. However this generated code can be edited by an expert user if so desired e.g., in the event of multiple available GPU devices. In the sample code below, a HabaneroC runtime call makes available pointers to all NVIDIA GPU devices (places); by default the first one is chosen for GPU steps but can be changed by the user. In the future we plan to extend the runtime to automatically choose an available GPU place.

```
place_t* gpu_place = get_places(NVGPU)[0];
```

Further, we generate code for the data transfer to and from the device. A sample of such code can be seen in listing 6.2. This code was generated for the encryption benchmark we will describe in more detail in chapter 7. For simplicity, details such as variable declarations have been omitted in this listing. On line 1, we allocate contiguous memory for num_tags items, each of size GPU_SIZE (the amount of copied data is currently a user defined constant to allow adjustable chunking of data, for flexibility; this is both a limitation as the user needs to specify the size of the copied data and a great opportunity for tuning an application by testing an application with different kernel sizes). We check that enough memory is available on the device (lines 2-5) and, on line 6, we copy the data required for the computation. We omit additional data allocation, because of its similarity to lines 2-6. On line 8 we call the CUDA kernel which performs the actual computation. The kernel is parametrized with the number of blocks per grid and threads per block typical of CUDA kernel calls and takes as arguments: an array with the tags associated with each thread (d_crypt_tag - a copy of the CnC tags placed on the device), the size of the tag array and a series of inputs (the array to be crypted - plain -, the encryption key - z -, and the array to store the encrypted text - crypt).

```
1   error = cudaMalloc((void **)&d_plain, GPU_SIZE * sizeof(unsigned char
        ) * num_tags);
2   if(error != cudaSuccess){
3       printf("Failed to allocated memory for plain in gpu_encrypt\n");
4       exit(1);
5   }
6   cudaMemcpy(d_plain, plain, GPU_SIZE * sizeof(unsigned char) *
        num_tags, cudaMemcpyHostToDevice);
7   //allocation for additional structures
8   gpu_encryptKernelCaller<<<blocks_per_grid, threads_per_block>>>(
        d_crypt_tag, num_tags, d_plain, d_z, d_crypt);
9   cudaThreadSynchronize();
10  if((error = cudaGetLastError()) != cudaSuccess) {
11      fprintf(stderr,"CUDA error in encrypt: %s\n",cudaGetErrorString(
            error));
12      exit(1);
13  }
14  cudaMemcpy(plain2, d_plain, GPU_SIZE * sizeof(unsigned char) *
        num_tags, cudaMemcpyDeviceToHost);
15  cudaFree(d_plain);
```

Listing 6.2: Automatically generated data transfer code for CPU-GPU communication.

All parameters are prefixed with "d_" referring to data stored on the device. On line 9 we wait for the kernel to finish executing and lines 10-13 check that the GPU execution completed successfully. Lastly, line 14 copies the data back to the host and line 5 frees the device memory.

Listing 6.3 gives more insight on what the generated kernel, called on line 8 of Listing 6.2, looks like.

```
1 __global__ void gpu_encryptKernelCaller(int *crypt_tag, int
    num_crypt_tag, unsigned char *plain, int *z, unsigned char *crypt) {
2     int tid = blockDim.x*blockIdx.x+threadIdx.x;
3     if(tid < num_crypt_tag) {
4         gpu_encryptKernel(crypt_tag[tid], &plain[GPU_SIZE * tid], z, &
              crypt[GPU_SIZE * tid]);
5     }
6 }
```

Listing 6.3: Generated kernel code calling user code for each GPU thread

Line 2 computes the GPU thread id, based on which, we check on line 3 that only the right threads perform work. For example if there are N pieces of data, on which some data parallel computation can be performed on the GPU, but M total threads that are running in a block (usually a power of two), with M>N, then only the first N threads should work. The condition on line 3 ensures that only the threads with available data call the user written code (line 4).

Although we have implemented the support for generating the code above, there are still details to sort out in terms of correct generation due to limited amount of information present in the graph file specification. Currently, we don't have a way of specifying, in the graph file, the sizes of each item to be copied to and from device. An option we considered is to use ranges to get a series of items thus being able to deduce the amount of data from the range bounds and data type. This however involves additional memory allocation and copying of data (each piece of data will be copied into device memory individually rather than a contiguous data copied all in one) and less flexibility in terms of tuning an application. In the above example, the data to be encrypted and the encrypted result are the same length and are divided amongst the GPU threads (each GPU thread uses a piece of the input and computes a piece of the result); however, the encryption key z is shared among the threads, and has a fixed size. As a result, even if we describe the input and output as ranges for a crypt step, we need a way to also specify shared data among the threads. In addition, for an efficient implementation, we need to consider coarsening of the data; if the input ranges from 1 to N, we may not want to assign a piece to N threads but create a coarser grained alternative where M pieces are assigned to N/M threads. For the purpose of demonstrating the usefulness of this approach we hand coded the data copy code for the encryption benchmark and obtained very encouraging results. This motivates further research on how to accurately specify such information while keeping the simplicity of the graph file specification language.

For the second approach detailed in section 6.2, which involves specifying affinities for steps, the changes involve generating the construction of an affinity field and passing this to the HabaneroC *async* construct. The code generated for listing 6.1 with affinities can be seen in listing 6.4.

```
1  void dispatchStep(Step* step){
2     int affinity = 0;
3     switch(step->stepID){
4     case Step_segmentation: affinity = 0 | GPU_AFFINITY(12);
5          async (gpu_pl) IN(affinity, step){ segmentation_gets(step->tag,
              (Context*)step->context, step); }; break;
6     case Step_registration: affinity = 0 | GPU_AFFINITY(5)| FPGA_AFFINITY
         (10);
7          async (fpga_pl) IN(affinity, step){ registration_gets(step->tag,
              (Context*)step->context, step); }; break;
8     case Step_denoise: affinity = 0 | CPU_AFFINITY(20)| GPU_AFFINITY(10);
9          async (gpu_pl) IN(affinity, step){ denoise_gets(step->tag, (
              Context*)step->context, step); }; break;
10    default: CNC_ASSERT(0, ”Attempt at executing an unknown step!”);
11    }
12 }
```

Listing 6.4: Steps launched with device affinity

We note that for each step, an affinity field is created using the values provided in the graph file, but these are also evaluated in order to launch the task at the device place with the higher affinity. In this context, device includes only GPU and FPGA, not CPU. Let's look at the example in order to better understand the procedure. For segmentation, the only affinity is with the GPU, thus the step is launched at gpu_pl (line 5). The registration step can run either on a GPU or an FPGA but the FPGA affinity is larger, so the step is launched at fpga_pl. As explained in the next section, the HabaneroC runtime may move this step from an FPGA queue to a GPU queue, if there's a backlog on the FPGA queue. Finally, the denoising step can execute on a CPU or a GPU and its largest affinity is with the CPU. However, for efficiency reasons that we will explain further, a task cannot be transferred from a CPU queue to a device queue, so the step will be launched at a GPU place. The motivation for this choice will become clear in the next section, when we explain how the runtime manages the tasks.

6.4 HabaneroC runtime support

As generating CUDA code did not require any changes in the runtime, we will dedicate this section to give an overview of the HabaneroC runtime support required for the second approach - launching steps with several device affinities.

The original HabaneroC runtime system was implemented for homogeneous multi-core processors, with support for work stealing among CPU workers. We extended the HC runtime to use the place/affinity annotation to facilitate task scheduling across heterogeneous processors by enabling work-stealing *among devices*. The *affinity* annotation presented in the previous section is used to specify which variants of code are available for a given task (currently, these can be some subset of CPU, GPU, and FPGA).

Figure 6.1 shows a work-stealing scenario for a group of tasks from the example in listing 6.1. The environment first launches all the tasks $(D_1, R_1, S_1, D_2, R_2, S_2, \ldots)$ on devices specified by their initial affinities. In this particular example, denoising $(D_1, D_2, \ldots)$ tasks can run on a GPU device or on a CPU device, and are initially launched on a GPU device. Segmentation $(S_1, S_2, \ldots)$ tasks can run on a GPU only. Registration $(R_1, R_2, \ldots)$ tasks can run on an FPGA or a GPU device, and are initially launched on the FPGA device. The CPUs can steal denoising tasks from the GPU, and the GPU can steal registration tasks from the FPGA.

In our implementation of work-stealing between devices, instead of stealing the

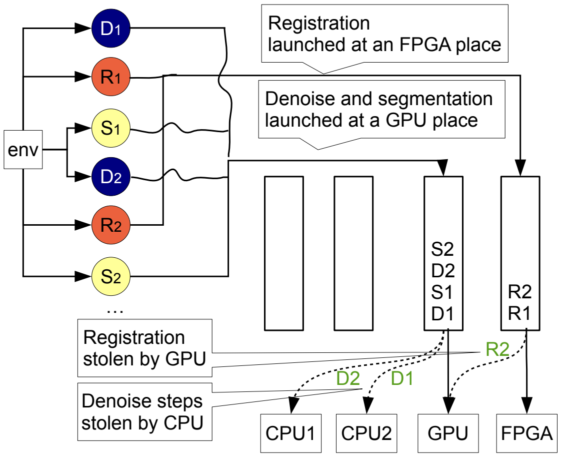


Figure 6.1 : Dynamic work-stealing between devices for the medical imaging pipeline

first task available on a worker's dequeue, the thief worker will choose the one with the highest affinity towards himself from the first N tasks on the dequeue, where N is a small number typically less than 10 defined by the runtime (in our results we used a lookahead of N=5). For example, if the top 5 tasks in the FPGA queue have GPU affinities of 10, 20, 30, 15 and 5, the GPU worker, when stealing from the FPGA queue, will steal the one with the GPU affinity 30.

A CPU worker can steal from any kind of device and any device can steal tasks from other devices (as long as the task can run on the thief's device). For performance reasons, we have disabled stealing from CPU queues by device workers. The reasons for this are twofold: a) the CPU tasks usually have small granularity, and the overhead of launching a task on a GPU or FPGA device can be significant, and b) allowing device workers to steal from CPU tasks would require locking the CPU queues in order to implement the N task lookahead, which would introduce significant overhead to CPU-to-CPU work stealing, which is currently implemented using a non-blocking algorithm [50].

Because of this restriction — disallowing device workers to steal tasks from a CPU worker — a task that has both CPU and some other device affinity needs to be launched on a device. In HabaneroC, the launch mechanism implies adding the task to the waiting queue for the device, not necessarily running it immediately. Thus, such a task may later be stolen by a CPU thread, whereas if it were launched on the CPU it would never be stolen by a device worker.

To implement steps that can potentially run on different components we used HabaneroC library function *current_ place()*, which can be used to determine the type of the hardware component the step runs on. Based on the device type, the step code calls the appropriate code or library routine that is compiled for that device type.

This approach yielded very good results, and we shall present them in the next chapter.

Chapter 7

Experimental results

In this chapter, we present experimental results obtained for a variety of benchmarks, structured as follows: a comparison study between the two runtimes implemented, performance results on the CPU comparing the C implementation of Concurrent Collections to Intel's CnC and OpenMP for different scheduling policies and finally some results for hybrid CPU, GPU and FPGA execution. The notations used in the graphs are:

RR for the Restart and Replay Runtime in CnC-HC

DD for the Data Driven Runtime introduced in this thesis for CnC-HC

WF for the Work-First scheduling policy [30]

HF for the Help-First scheduling policy [30]

HC for HabaneroC [28]

These have been described in sections 2.2 and 2.3 and in Chapter 5.

The results presented in sections 7.1 and 7.2 (CPU only) were obtained on an Intel(R) Xeon(R) E7330 @ 2.40GHz with 16 cores. The results involving a GPU were obtained on 3 machines. Crypt was run on (a) an Intel(R) Xeon(R) E5504 with 8 cores and an NVIDIA Quadro FX 580, with 256 threads, 32 CUDA streaming multiprocessors and 512 MB memory and (b) an AMD Phenom(tm) 9850 Quad-Core Processor and an NVIDIA Tesla C1060, with 512 threads, 240 CUDA cores and 4

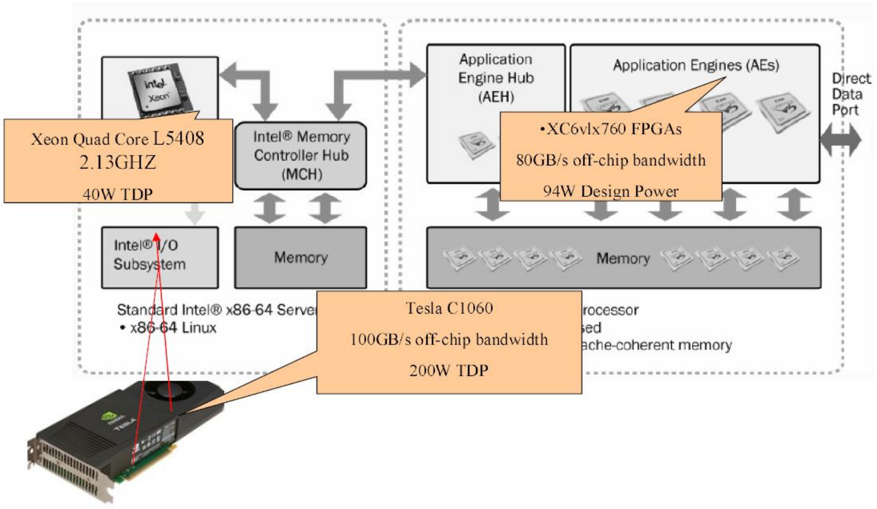


Figure 7.1 : The Convey HC-1ex architecture. [2]

GB memory. The medical imaging pipeline was run at UCLA, on 2-4 CPU threads, 1 GPU and 1 FPGA, on a Convey HC-1ex (see figure 7.1). Currently this testbed is using a PCI-express expansion box to host a Tesla compute card C1060 and a virtex6 LX760 as the user FPGAs. CPU and different FPGAs access the off-chip memory using a shared memory model. The system employs an onboard crossbar to realize the interconnection. Cache coherence is also handled through the FSB protocols. Convey HC-1ex provides a very large bandwidth (80GB/s peak) for coprocessor side memory. In practice, it was observed that around 30% to 40% of the peak bandwidth can be easily obtained.

7.1 Performance comparison study between the Restart and Replay and Data Driven runtimes

One of the first objectives was to conduct a performance study between the two runtimes implemented for CnC-HC. Judging by the implementation details, here's when the two differ and would perform better or worse. The Restart and Replay runtime will restart a step for every datum that was not found when attempting to execute the step. Thus, for applications with many data dependencies and/or much parallelism (many spawned tasks), the overhead of creating asynchronous tasks and aborting them would tend to affect performance. On the other hand, in the Data Driven runtime, keeping the dependency record is done serially for each step, before spawning the task, which wastes part of the parallelism available in the situation when the data was available so no restart would have been necessary. Creating the structure to hold the said dependencies can also be a source of overhead.

7.1.1 When the Data Driven runtime is the better choice

We consider a variant of the Java Grande ForkJoin micro benchmark [51] to measure how much overhead the creation and termination of a large number of tasks would incur. In this application, we create N tasks that will abort due to lack of data and will restart once the data becomes available. We engineer the application to create tasks with a single dependency that are ensured to not find the needed data on the first attempt. We plot how both runtimes perform for work-first and help-first scheduling policies in HabaneroC.

In figure 7.2, we plot the time for N = 10^3, 10^4, 10^5, 10^6, 10^7, when tasks are doing trivial work. As a result, the task creation overhead is larger than useful work and the application does not scale.

For both WF and HF HC policies, the Data Driven CnC runtime outperforms Restart and Replay with a time difference proportional to the number of tasks. For the small case with 10^3 tasks, the benefit is not obvious because the times are small, so we may have more noise than useful work; however, we notice that the trend is constant as the sizes increase and for the pathological case with 10^7 tasks performing little work the Data Driven runtime is two times faster than Restart and Replay on 16 cores, saving order of minutes depending on the policy. In this case, on the initial runs we noticed a spike for the HF case for 1 core (much larger time than expected). This occurred in both runtimes and could be explained by the creation of all the tasks (and their placement in a dequeue) before executing them. In HabaneroC, the dequeue of tasks expands at runtime as required by the application (size is doubled each time there isn't enough space; copying the data to a new dequeue makes this very expensive), and, in such a case, repeated memory allocations cause the HF version to take a much longer time. In the timing presented in this work, we set the initial dequeue to a large enough size to avoid the need of expansion. This is not always a solution, as high memory consumption of the application can cause the machine to swap when a large amount of memory is already preallocated.

To get a better look at how much we gained from avoiding the restarting of steps, we plot the time difference between the two runtimes (Restart and Replay minus Data Driven) in the cases presented above (Figure 7.3)

Also from figure 7.2 we notice that the HF implementations take less time than the WF ones. This again is to be expected based on the behaviour explained in section 2.2. In WF, the spawned task is executed by the thread that created it, and the continuation is placed on a dequeue for the others to steal. In this benchmark, the

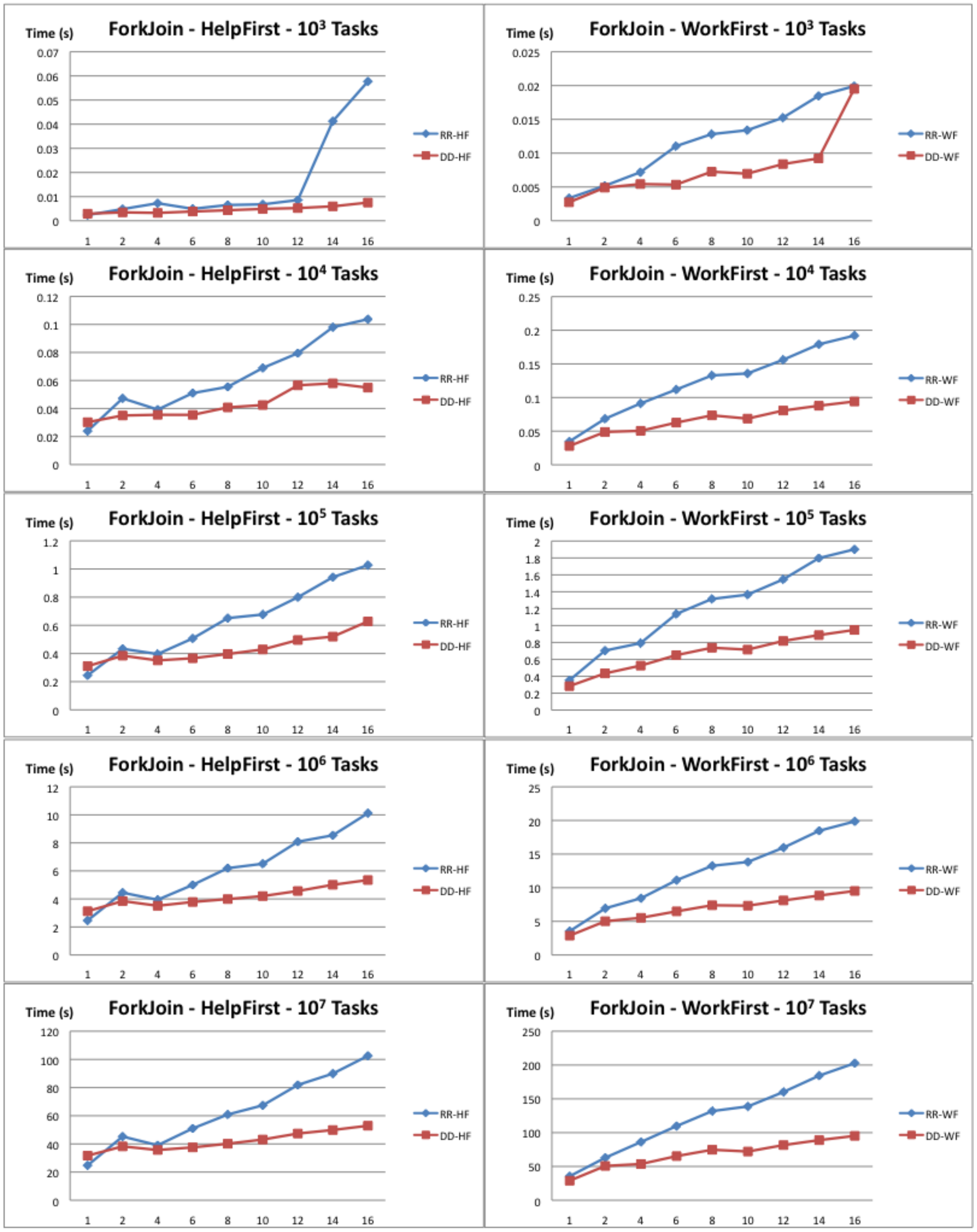


Figure 7.2 : Performance comparison between the Restart and Replay and Data Driven Runtimes for ForkJoin benchmark, with little work, for 1-16 cores

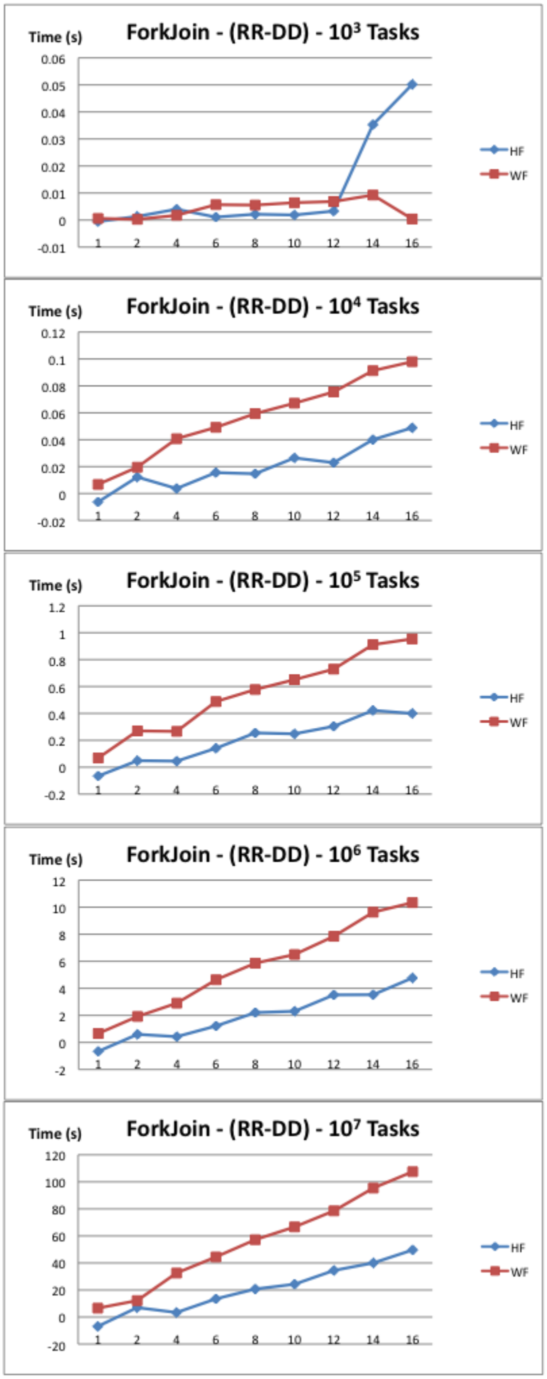


Figure 7.3 : Time difference between the Restart and Replay and Data Driven Runtimes for ForkJoin benchmark, with little work, for 1-16 cores

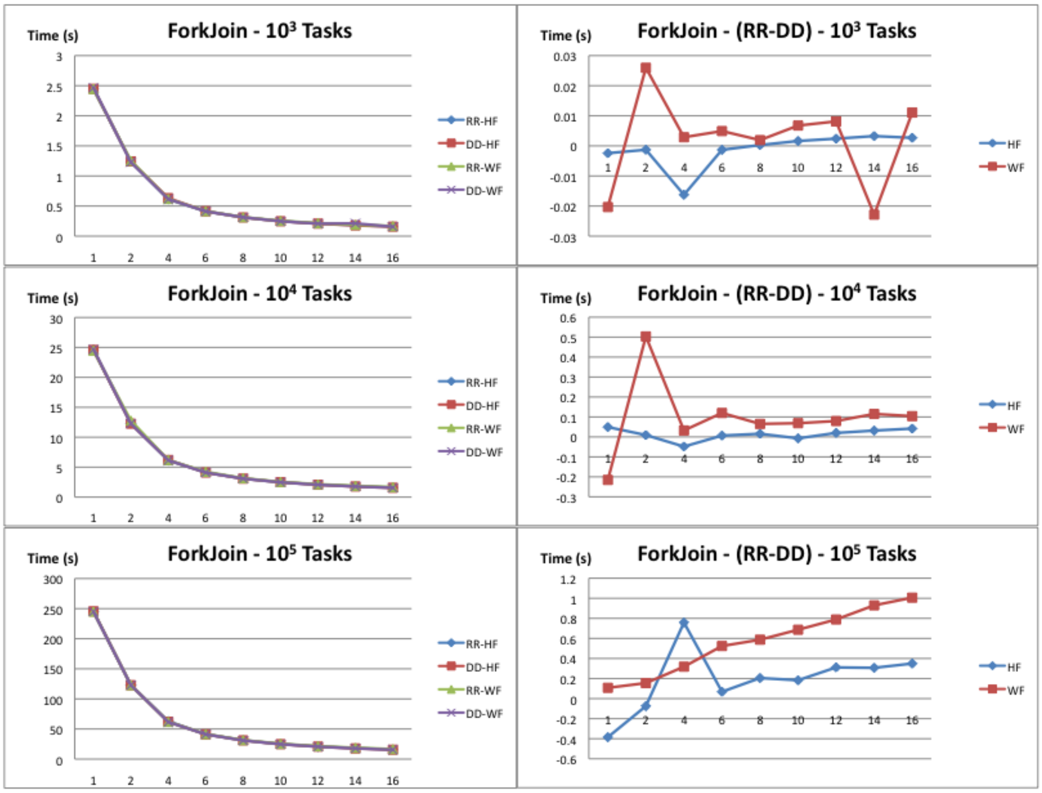


Figure 7.4 : Performance comparison between the Restart and Replay and Data Driven Runtimes for ForkJoin benchmark, with large work, for 1-16 cores. For the graphs on the right, a negative value indicates a case for which the Restart and Replay runtime performs better than the Data Driven runtime.

continuation consists of N-1 tasks. Thus, only one large task is available for stealing at any time, leading to a large number of steals ("steal conflicts" [30]). In contrast, in HF, the worker will "give" the spawned tasks to the other cores, thus reducing the contention due to stealing. The difference would also be less pronounced if the tasks were large enough so that the duration of the work performed in each task would hide the stealing overhead or if the tasks were spawned in a tree-like structure as done by the Cilk [22] language.

We take the application described above and enhance it to make each task perform useful work (50 thousand trigonometric operations). Figure 7.4 shows results for data sizes between $10^3 - 10^5$; larger sizes gave similar results and were ommitted for brevity. We see immediately on the left-side graphs that the application now scales and, what is more, the differences between the two runtimes and policies and now minimal. On the right-hand side we plotted these differences which are now negligible. We can see that occasionally the Restart and Replay runtime outperforms the Data Driven runtime, which can be explained by the sequential checking of dependencies and thus loss of parallelism.

In an attempt to lessen the difference between the WF and HF policies in the initial implementation, we created another variant of the ForkJoin benchmark which we call *TreeForkJoin*. In this approach, the main function will spawn M *node steps*, who will in turn each spawn M other node steps, and so on, thus creating a tree structure. After creating the child nodes (spawning the node tasks), each node step will spawn a *compute step*, which will perform the same small computation as ForkJoin. In order to properly see the difference between the two runtimes, we required that each compute step read 10 items and engineered the application such that the first Get attempt fails. The results are presented in figure 7.5, for a total number of compute steps ranging from $10^3 - 10^7$, where the number of nodes on each level of the tree is M=10. As before we notice that for the smallest size 10^3, the differences in actual time are not relevant and can be attributed to noise; however as the sizes increase, the trend becomes obvious and the differences more pronounces.

To get better insight into what is taking most time in each of the cases, we also timed separately, for each case, how much it took to spawn the node tasks and to allow the compute tasks to fail, and how much it took to run all tasks after the items became available for all tasks. We noticed that the first part took more time in the Data Driven runtime, as the data structure filling all dependencies was created serially

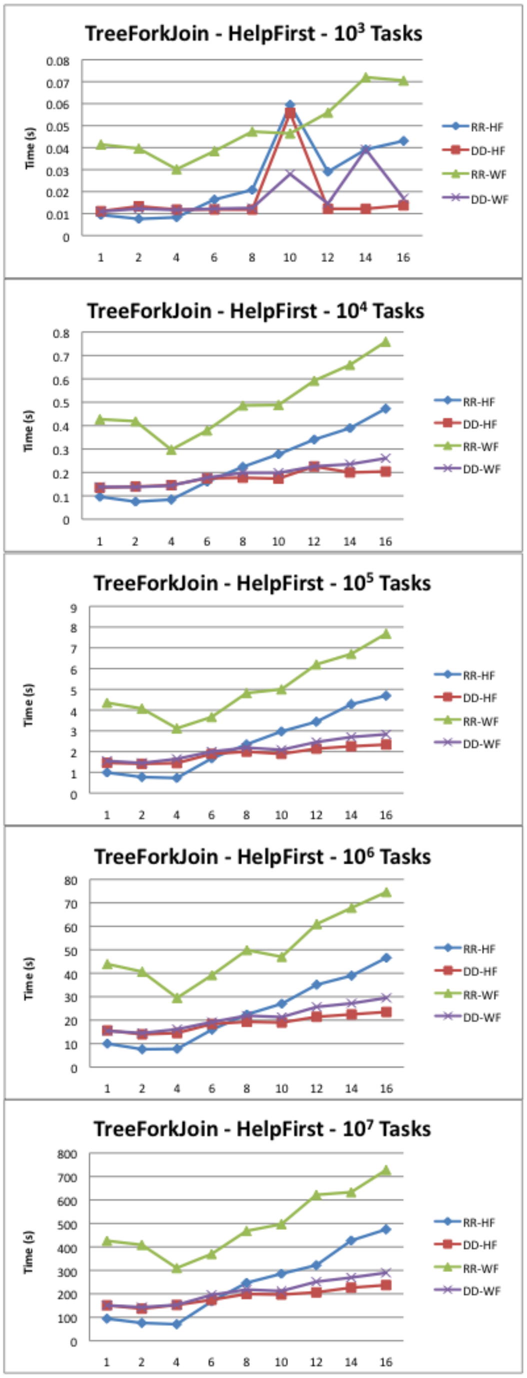


Figure 7.5 : Performance comparison between the Restart and Replay and Data Driven Runtimes for TreeForkJoin benchmark, with little work, 10 failing Gets, for 1-16 cores.

for each compute step, while in the Restart and Replay runtime these compute steps were eagerly executed and aborted after failing to find the first item. The second part took considerably more time for the Restart and Replay runtime, as, once the first item became available for a compute step, it was again executed eagerly and failed on the next item. In contrast, in the Data Driven runtime, the Put only triggered a check in the dependency list created previously.

We notice that for the Data Driven runtime, this approach made the two policies (work first and help first) have very similar results, because we have eliminated the work-stealing overhead (steal conflicts) through tree-like spawning. In this micro-benchmark, the short coming of the work-first policy is only visible in the Restart and Replay runtime because of the way tasks are respawned in the runtime, which is eagerly for each item put. In contrast, in the Data Driven runtime, even though the tasks are also analysed in a for loop, only those with all data available are spawned using async. Thus the overhead we see for the RR-WF line in figure 7.5 can be attributed to work-stealing overhead caused by the way tasks are spawned. This can be removed either by using a different data structure in CnC for keeping the enqueued steps, such as a concurrent skip-list or by creating a "parallel_for" construct in HabaneroC such as Cilk's "cilk_for" [50]. We note that in the HF policy this problem does not arise, yet the results show that Data Driven still outperforms Restart and Replay in a scenario such as the one presented above.

7.1.2 When the Restart and Replay runtime is the better choice

We chose to make a second variant of the ForkJoin micro benchmark to model a situation when using the Restart and Replay runtime would be preferable. In this implementation we create N tasks that need to get 100 input items that are available before execution begins. Figure 7.6 shows the results for this case: the first column of graphs presents the absolute times for the the two runtimes, for the Work-First and Help-First policies when creating $10^3 - 10^7$ tasks, while the second column of graphs shows the time difference between the two runtimes.

We notice that the execution time increases with the number of cores for the Data Driven runtime, the reason being that it loses parallelism by checking dependencies serially and adding more cores only increases the overhead of work-stealing (for 10^7 tasks, running on 1 core causes the test machine to swap, yielding the spike displayed in the graph). In contrast, the Restart and Replay runtime scales well in the beginning, but for more than 8 cores the time it takes is about constant. This is due to the fact that the tasks are created sequentially and take little time to execute, so after a certain threshold any core added will be idle as there isn't enough work available for it. The tests could be further tuned to include larger amounts of work which would show good scaling for the Restart and Replay runtime, however, this synthetic case was created to show how checking dependencies beforehand can lower performance and we have shown this already.

We conclude that under heavy load of small parallel tasks, that are likely to fail on retrieving the inputs needed, the Data Driven runtime is a preferable choice to the Restart and Replay as it avoids the task-creating overhead. In contrast, the Restart and Replay Runtime performs better for a heavy number of successful Get calls as it eagerly executes tasks, thus obtaining more parallelism than the "safe" approach. These two situations are the extreme cases and recognizing how an application is modelled will help choose the best runtime for that application; for regular applications, however, the two runtimes can be used interchangeably.

Another useful approach may be to alternate policies, e.g., eagerly spawn a task

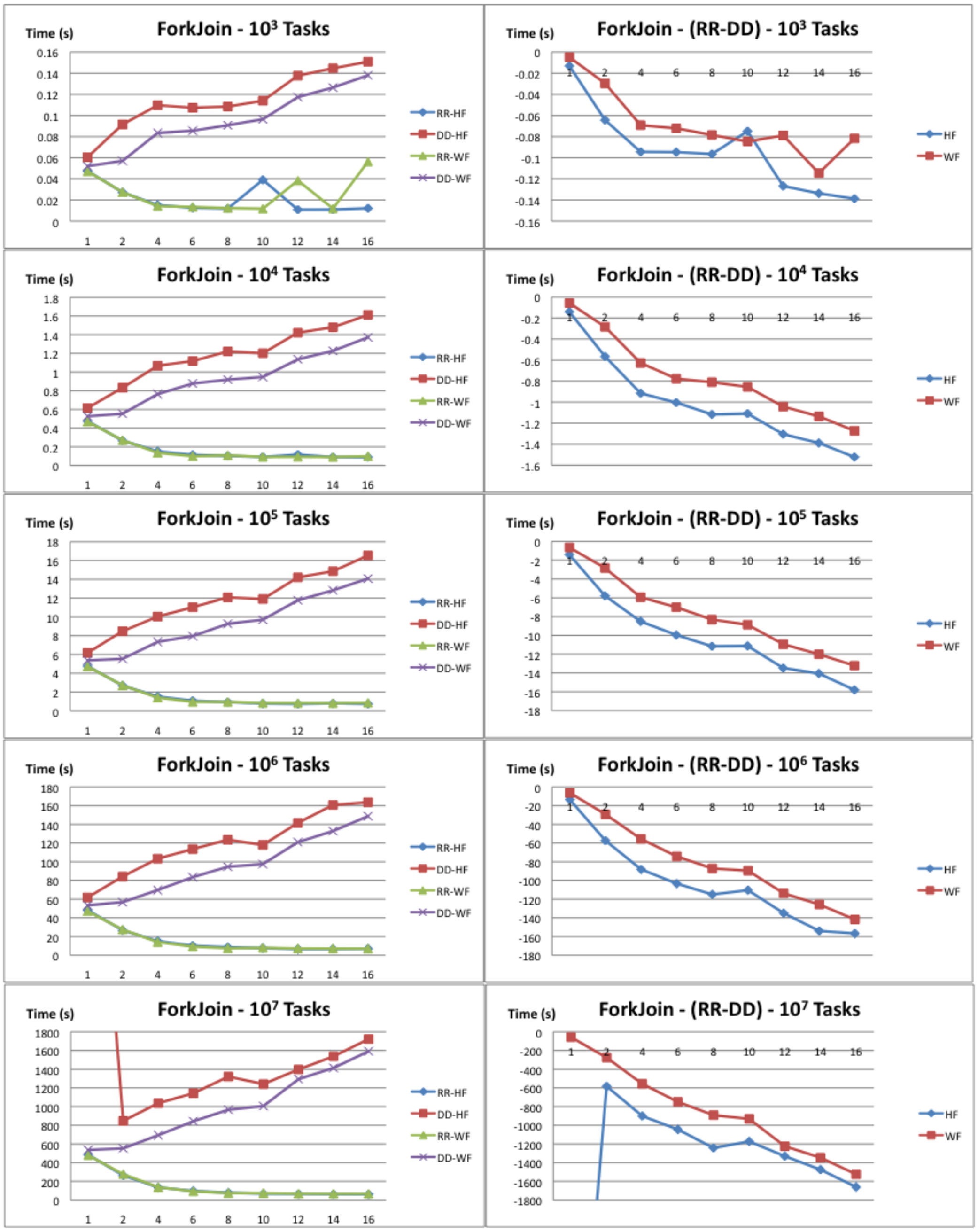


Figure 7.6 : Performance comparison between the Restart and Replay and Data Driven Runtimes for a variant of the ForkJoin benchmark when each task needs 100 items that are already available; 1-16 cores. The graphs on the right indicate that the Restart and Replay runtime performs better than the Data Driven runtime when getting the data would be successful on a first attempt.

a number of times to exploit parallelism but, after a threshold, continue the checks sequentially if all previous spawns failed. Researching such a runtime is a subject for future work.

In the remainder of this chapter we will report results for the Data Driven runtime, as testing showed no noticeable difference between the times collected for the two runtimes on the benchmarks we present further.

7.2 CPU testing and benchmarks

We consider three benchmarks for comparing the HabaneroC implementation of CnC with either a previous implementation in Intel's CnC or a reference OpenMP benchmark.

The first benchmark chosen is Cholesky factorization, which takes as an input a positive-definite matrix and decomposes it into a lower triangular matrix and its conjugate transpose. The well known algorithm, which computes the resulting triangular matrix, will update the columns starting with the first, use these values to update each of the rows (the value on column 1 and row i is used for the remaining columns on row i), then repeat the process for the remaining columns using the values computed in the previous iterations. A tiled version of the algorithm will perform the computation serially for a matrix of size tile*tile, using adjacent tiles as inputs.

Figure 7.7 shows the results for the implementations of CnC-HC with work-first and help-first policies and Intel's CnC. The runs are done on 1-16 cores using an input matrix of 2000 by 2000 elements and the computation is tiled with tile sizes of 50, 100, 125 and 200. The performance differences when varying the tile size are due to cache effects, and previous work [52], [53] has shown that the best times are obtained for a tile size of 125 on this machine.

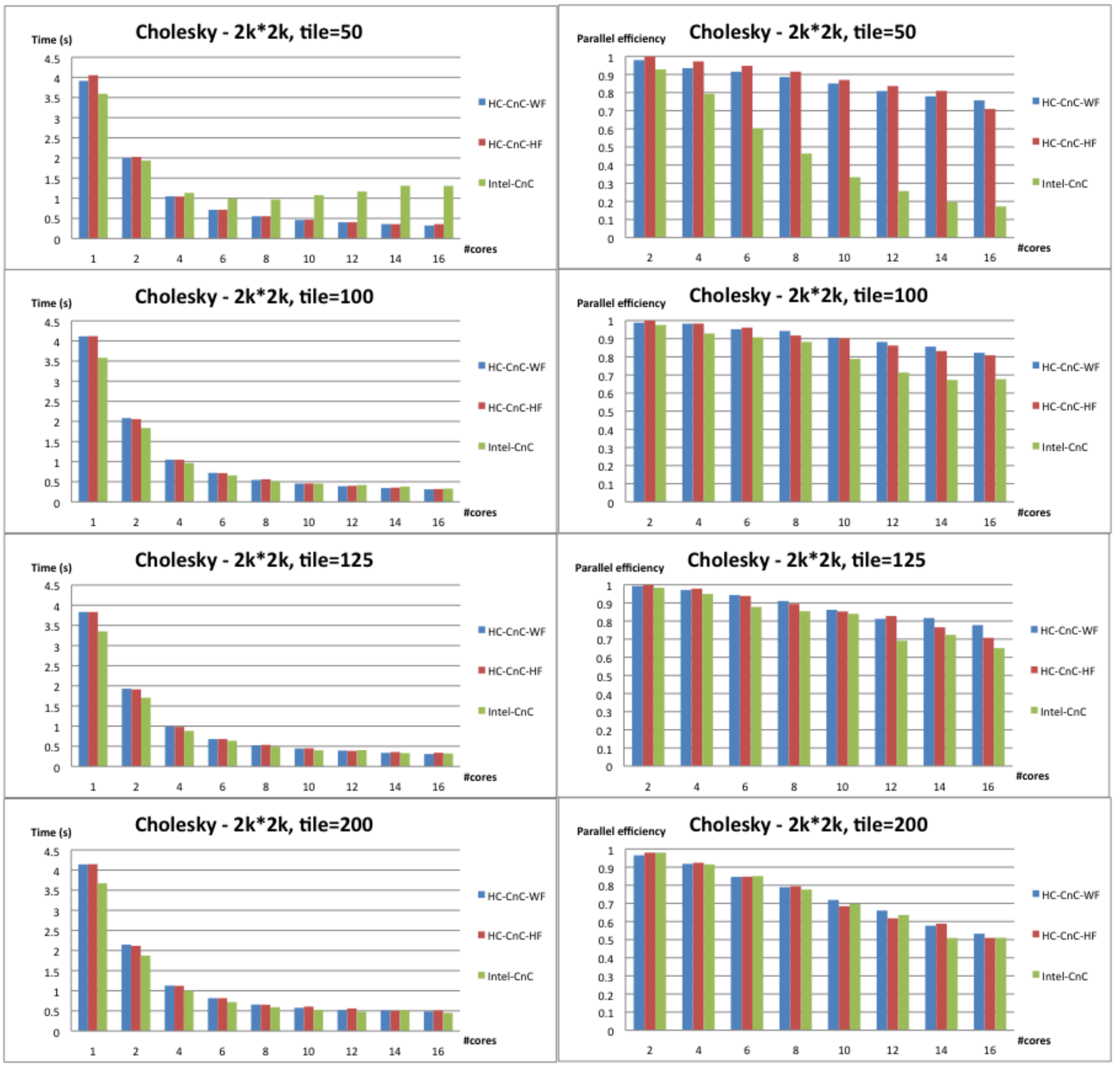


Figure 7.7 : Performance comparison (time and parallel efficiency) between Intel's CnC and CnC-HC for the Cholesky factorization, on 1-16 cores

Analysing the results we can draw quite a few conclusions. First, we note that Intel's CnC always performs better for execution on small number of cores (1,2), which shows that the overhead of spawning tasks can be improved in our implementation. Secondly, our results show that if the computation is too fine grained (tile size = 50), Intel's CnC begins to perform poorly and fails to scale beyond 4 cores, whereas the CnC-HC still offers good scalability and close to optimal performance. Thirdly, the best times are obtained for both version - as expected - for tile size equal to 125, for which the CnC-HC outperforms Intel's CnC implementation on 16 cores by 4.2%. For other tile sizes around the "sweet-spot", we notice CnC-HC offers better times for smaller tasks (5.4-9.8%) while Intel's CnC does better for coarser granularity (0.4-9.3%)

The second benchmark, taken from the Parsec benchmark suite [54], is option pricing with Black-Scholes partial differential equation. This is a embarrassingly parallel benchmark which computes the price using the Black-Scholes equation for one million items, over one hundred trials. Figure 7.8 shows a performance comparison between the OpenMP implementation available from [54] and CnC-HC with work-first and help-first policies. We tested the CnC version for two cases, varying the amount of work for the whole that is attributed to a core at any iteration. The differences in performance were negligible, which for such a benchmark is encouraging as it shows our model can handle both complex graphs and parallelism of the type "parallel-for" with no noticeable overhead.

The third benchmark, taken from the Rodinia benchmark suite, is Heart Wall Tracking. This is the final stage of an application that tracks the movement of a mouse's heart over a sequence of 104 609x590 ultrasound images [55]. By using sample points to observe the changes in the sequence of images, it aims to detect

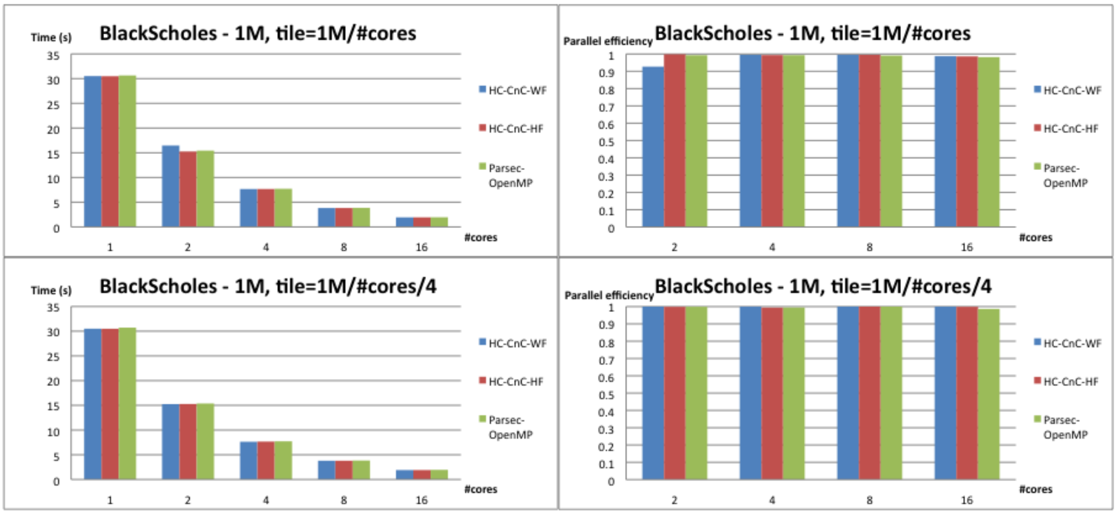


Figure 7.8 : Performance comparison Parsec's OpenMP and CnC-HC for BlackScholes, on 1-16 cores

the movement of the heart walls. The processing is done frame by frame, in a serial way due to the dependencies between adjacent frames. For each frame a number of sample points are processed with no dependencies between each other; these points are retrieved from a previous edge detection processing of the sequence of images. The processing for each point is based on a template, updated every 10 frames; the initial template is based on the first frame. The movement of the heart walls is detected by comparing the frame currently being processed to the template. Processing a single point for a certain frame: (point, frame), involves a large number of small steps, dependent only on each other, point, frame and the template updated after every batch of frames. However, according to our analysis of the source code, processing (point, frame) will only update its share of the template, while (point, frame+1) will only use the template updated in the previous step, the same share of the template. The Rodinia OpenMP implementation uses *parallel for* to execute in parallel the processing of all sample points points for a frame. Since the parallel for pragma has an implicit barrier after each step, the full template will be updated before the next frame starts to be processed. We created two versions of the benchmark in CnC-HC and compared them with the OpenMP version. The results are shown in figure 7.9. In the first CnC implementation, we created each step with just one dependency, that of the previous frame processed. In this manner, we relaxed the barrier and exposed more parallelism for the HC work-stealing scheduler to handle load balancing of the tasks. This yielded a 13.35% improvement relative to the OpenMP version. In the second version, we analysed the computation performed for each step (identified by a (frame, point) pair) and created a more finer grained graph (composed of 10 steps). However, since each of the steps was decomposed into a single entry single exit graph, the parallelism obtained from within a step was not significant and didn't yield much performance improvement, but a slowdown compared to the first CnC version and a 10.5% improvement compared to the OpenMP implementation.

7.3 Hybrid testing and benchmarks

The first benchmark we implemented for hybrid execution was *crypt* from the Java Grade Benchmark suite [51]. This application performs encryption on a large amount of data, then decrypts the same amount of data to obtain the initial information and performs an equality comparison to check correctness. Typical sizes for the data for the JGB are 10^4 (SizeA), 10^5 (SizeB), 10^6 (SizeC), however we tested for larger sizes: $2 * 10^8$, $5 * 10^8$ and 10^9, to show how an accelerator can significantly increase performance for such loads. We note that for hybrid execution, a thread will be assigned to interfacing with the device, thus we will have N-1 working threads for a machine with N cores. The results can be seen in figure 7.10.

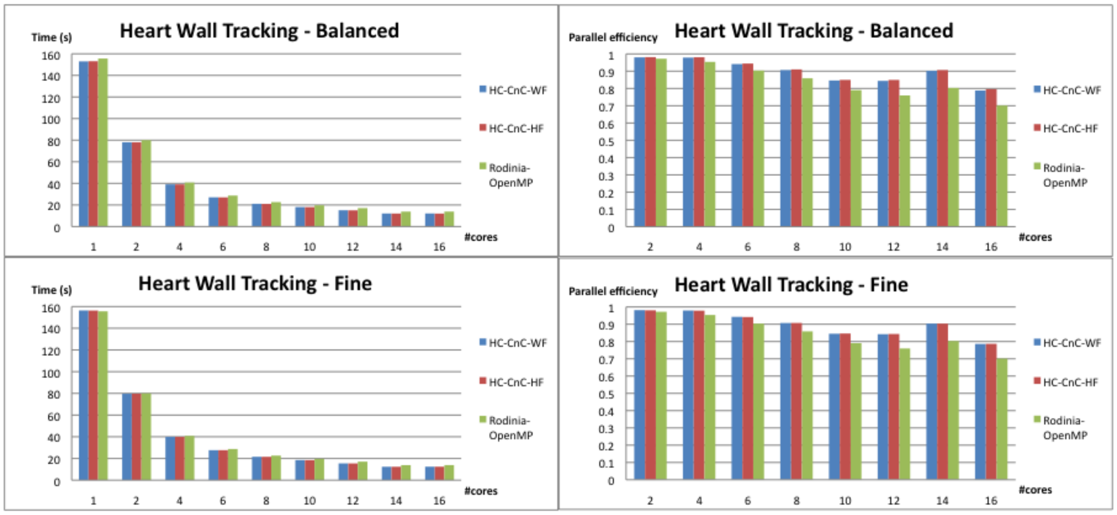


Figure 7.9 : Performance comparison Rodinia's OpenMP and CnC-HC for Heart Wall Tracking, on 1-16 cores

We used two machines to test this benchmark, running a percentage of the computation on the GPU and the rest on the available CPU cores. The first machine has 8 CPU cores and an older GPU model (an NVIDIA Quadro FX 580). For $2 * 10^8$ data size we obtain the best performance if we offload 30% of the computation to the GPU. For $5 * 10^8$ elements, the optimal case is when running 40% on the device, but we run out of device memory when attempting to execute 60% of the computation on the GPU. Similarly for 10^9 data size, we can only run up to 30% of the computation on the device. This limitation is due lack of data size knowledge from the current graph specification, but there is potential research opportunity in chunking of data and launching multiple device kernels in order to get past this obstacle.

The second machine we ran the crypt benchmark on, has a better GPU (an NVIDIA Tesla C1060) but only 4 CPU cores. The results in this situation show that the best results (a speedup of 80%) are obtained when 90% of the computation is

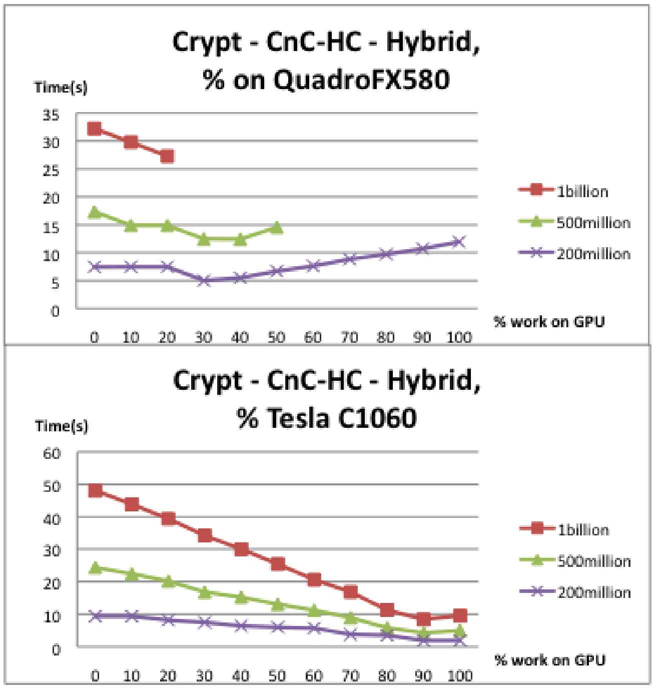


Figure 7.10 : Performance on two GPU models of the crypt benchmark when a percentage of the work is offloaded to the GPU

performed on the GPU. We need to mention that the difference in speedup between the two situations is also due to the difference in the number of cores available in each of the machines. Comparing the results in these two situations, we note that further investigation and testing on the latest GPU models with a larger number of cores would perhaps reveal other optimal points for executing similar applications. Our current results show that it is a great feature to have the option to use the GPU card in a laptop computer to speed up computation - e.g., on the first machine the computation was sped up by 32.4% just by running 30% of the computation on the GPU.

The second benchmark we tackled was a medical imaging pipeline, composed of three stages: denoising, registration and segmentation. This implementation was developed as part of the CDSC project [13]. The first stage, denoising, performs an image noise reduction to facilitate further image analysis. The second stage, registration, compares the image to previous samples in order to detect possible changes. Since we are dealing with medical images, this pipeline is mainly used for detecting tumours or aneurysms and recording their progress in time through of subsequent studies from different MRIs. The third and final stage, segmentation, has the role of emphasising a location in the image where a change has occurred, like the growth or appearance of a tumour. We ran a pipeline containing 10 runs of denoising, registration and segmentation (simulating 10 images run in parallel), each processing an image of 256*256*256 voxels, where denoising had 3 iterations, registration 100 iteration and segmentation 50 iteration (the reason for the multiple iterations is to control the convergence of the algorithms).

First, we considered a CPU only implementation for all stages in the pipeline. We them improved on this by providing GPU implementations for all stages and retesting the benchmark. Further, we merged the two, to obtain a hybrid approach where some steps were run on the GPU and some on the CPU. We used the affinity metric we presented in previous chapters to mark which steps are better suited for the CPU and which for the GPU. Finally, we tested the benchmark on a hybrid architecture encompassing a 4 core CPU, a GPU and an FPGA by also providing an FPGA implementation for the registration step. While it is possible to run multiple kernels on the FPGA, in practice that may involve a large reconfiguration overhead. In our experiments, we configured the FPGA to accelerate the registration kernel only. We took two approaches here: one involved a static mapping, similar to what a programmer would write in the absence of a cross-device work-stealing runtime, and one with the extended HC runtime we presented in section 6.4. In the static

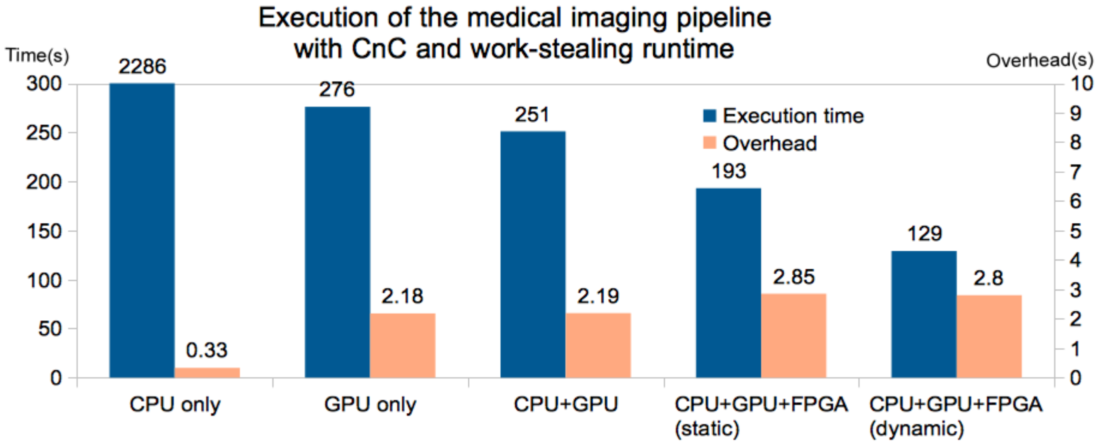


Figure 7.11 : Evaluating performance for the medical imaging pipeline on a Convey HC-1ex with 2 CPU threads, 1 GPU and 1 FPGA

approach we executed the denoise step on the CPU, the registration on the FPGA and the segmentation on the GPU, as this mapping yielded the best performance for each step in turn and for the overall application. In the dynamic approach, we allowed steps to run on multiple architectures as presented in listing 6.1.

Figures 7.11 and 7.12 show the results obtained on a Convey HC-1ex, on which a C1060 GPU was added; these results are also detailed in the paper [56] submitted for publication and were obtained in collaboration with the co-authors from University of California Los Angeles (UCLA). In both figures we plot the time for the runs on various architecture choices and notice an increase in performance of 8.28× just from switching from CPU to GPU only, while, when using all devices, we get 11.84× for the static scheduling and 17.72× for the dynamic one. As we assign each device a CPU thread, for the CPU+GPU results only 3 cores are doing computation, while for the CPU+GPU+FPGA results only 2 CPU cores do actual work.

In figure 7.11 we also show the measured overhead of launching steps on different

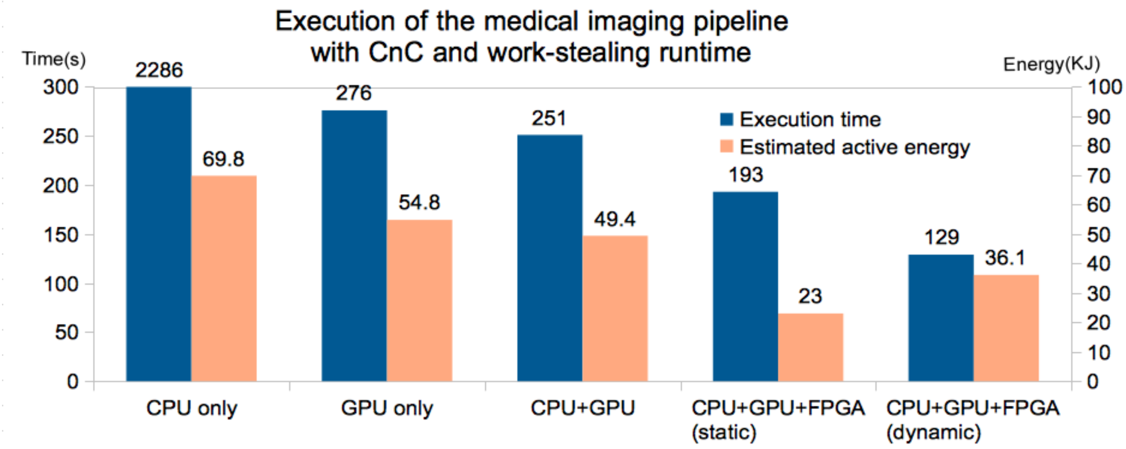


Figure 7.12 : Evaluating performance for the medical imaging pipeline on a Convey HC-1ex with 2 CPU threads, 1 GPU and 1 FPGA

devices (creating HabaneroC frames for CPU tasks and copying of data to and from the device for GPU tasks). We note that even though the overhead increases significantly when using multiple devices, the increase in performance more than makes up for it.

In figure 7.12 we plot the estimated energy consumption which is computed by summing up the energy spent by each device that contributes to the computation, assuming 10W per core for the CPU, 200W for the GPU and 94W for the FPGA, ignoring idle power. One of the reasons why the static binding has a lower energy estimate than dynamic binding is because idle power is ignored. If the idle power and the power of other system components are considered, we expect that dynamic binding will have a lower energy cost because of its shorter execution time. The data in figure 7.12 also suggests that different mapping policies may be needed to optimize the energy consumption and/or the overall performance.

In this chapter we have presented our experimental results. We first made a study on the two runtimes presented in this thesis and argued when each can outperform the other basing our deductions on analysis of the implementation and confirming through experimental results. We then presented a series of CPU benchmarks showing multiple aspects: our approach outperforms Intel's CnC as showed by the Cholesky benchmark, it does not add notable overhead to embarrassingly parallel computations compared to OpenMP as showed by the BlackScholes benchmark and using the CnC model can outperform OpenMP as it can express arbitrary task graphs thus exploiting more parallelism if this exists, as shown by the Heart Wall Tracking benchmark. In addition, porting these applications took little time due to the introduction of tag functions and having a significant amount of code auto-generated, especially the calls reading the inputs (Gets). Finally, we showed two approaches to tackle mapping on heterogeneous architectures: we introduced a way to specify GPU steps thus enabling auto-generation of much of the glue code between CPU and GPU (such as data copies, kernel launches), and we extended the HabaneroC work-stealing runtime to support cross-device work-stealing, which coupled with the introduction of affinities enables the definition of multi-versioned computation steps and allows load balancing, thus obtaining increased performance and reduction in energy consumption.

Chapter 8

Conclusions and future work

In this work we have shown how a general data flow model can be extended for developing and running programs on hybrid architectures. In the process we retained the dataflow model's most important guarantees: determinism, race and live-lock freedom while obtaining good speedup when using accelerators like GPUs and FPGAs.

First, we presented the changes made to the user specification language and the need for additional information introduced by tag functions and ranges. We showed how tag functions specify the relation between the tag that uniquely identifies a step and the tags identifying the data items that the step is either reading or writing, while ranges define contiguous sets of items read or written by a step. We used these extensions to enable the compiler to automatically generate high level operations for inter-task communication, more precisely, the auto-generation of the calls to the functions that read the data items (Get functions) within a computational step. In addition we introduced the generation of function calls that write data items (Put functions) and that spawn of other steps as helper code to improve usability. The addition of tag functions also makes the code more analysable, opening the door for future optimizations. Some of these are: the possibility to signal errors or problems in the specification through static analysis (such as defined steps not being started by any other step or the environment, items never being computed or computed items never being used) or the opportunity to help with scheduling choices and decisions on memory management.

Second, we described a novel, data-driven runtime for the CnC model, developed in a C framework, using HabaneroC (HC). We also created an implementation of the previous runtime approach and conducted a study between the two runtimes to determine the situations when each performs best. We determined that there are situations when either runtime can outperform the other. Under heavy load of small parallel tasks, that are likely to fail on retrieving the inputs needed, the Data Driven runtime is a preferable choice to the Restart and Replay as it avoids the task-creating overhead. In contrast, the Restart and Replay Runtime performs better for a heavy number of successful Get calls as it eagerly executes tasks, thus obtaining more parallelism than the "safe" approach. A useful approach may be to alternate policies, e.g., eagerly spawn a task a number of times to exploit parallelism but continue the checks sequentially if the spawn failed every time. Researching such a runtime is a subject for future work.

The third aspect we tackled is the extension of CnC-HC for heterogeneous architectures. We introduce a way to specify steps that are data parallel and thus are fit to execute on the GPU, and also the notion of task *affinity*, a tuning annotation which offers hints on where a computational step should be run. Affinities are used by the runtime during scheduling and can be fine-tuned based on application needs to achieve better (faster, lower power, etc.) results. Further, we showed how we expanded the HabaneroC dynamic work-stealing runtime to allow cross-device stealing based on task affinity, in order to achieve load balancing across heterogeneous platforms for improved performance. We are currently exploring the possibility of extending the notion of affinity to encompass locality in space and time in a hierarchical definition of the steps in the CnC graph. Moreover, further work can enable affinities to be determined and adapted automatically by combining static analysis and runtime profiling. Another extension, would be the introduction of a means to specify data sizes in the graph specification, so as to fully automate the generation of data copying code between the host and devices. In addition, this would also enable the splitting of groups of data such that multiple GPU kernels could be launched when there are memory constraints. On that subject there is also the hard problem of how best to exploit different storage classes on a GPU device.

Finally, in our results we started by making a comparison study between the two runtimes that we implemented, in order to assess how the theoretical differences affect actual programs. Next, we used a series of CPU benchmarks in order to evaluate the performance of our CnC-HC implementation. We compared to Intel's CnC runtime (based on C++) and with OpenMP and concluded CnC-HC did just as well or better in terms of performance. Further, we used the extensions we made for hybrid execution to evaluate two benchmarks on systems encompassing CPUs, GPUs and FPGAs and proved that this approach can yield a significant increase in performance and decrease in power consumption, for applications with steps that can exploit such hardware, when using CnC-HC for a hybrid CPU/GPU/FPGA execution. To evaluate how a larger variety of applications could take advantage of this model, more applications need to be ported, as part of future work.

We conclude with our claim sustained by practical results, that a high level and easy to use data flow model can be extended for execution on hybrid architectures, while yielding very good performance, low power consumption and retaining high programmability.

Bibliography

[1] M. Grossman, A. Simion-Sbirlea, Z. Budimlic, and V. Sarkar, “CnC-CUDA: Declarative Programming for GPUs,” in *Languages and Compilers for Parallel Computing: 23rd International Workshop*, (Heidelberg, Dordrecht, London, New York), pp. 230–246, Springer, 2010.

[2] “Convey Computer Corporation.” http://www.conveycomputer.com.

[3] J. Dean and S. Ghemawat, “MapReduce: simplified data processing on large clusters,” *Commun. ACM*, vol. 51, no. 1, pp. 107–113, 2008.

[4] B. He, W. Fang, Q. Luo, N. K. Govindaraju, and T. Wang, “Mars: a MapReduce framework on graphics processors,” in *Proceedings of the 17th international conference on Parallel architectures and compilation techniques*, PACT ’08, pp. 260–269, 2008.

[5] J. Yeung, C. Tsang, K. Tsoi, B. Kwan, C. Cheung, A. Chan, and P. Leong, “Map-Reduce as a Programming Model for Custom Computing Machines,” in *Proceedings of the 16th International Symposium on Field-Programmable Custom Computing Machines*, FCCM ’08, pp. 149 –159, Apr. 2008.

[6] J. Nickolls, I. Buck, M. Garland, Nvidia, and K. Skadron, “Scalable Parallel Programming with CUDA,” *ACM Queue*, vol. 6, no. 2, pp. 40–53, 2008.

[7] Khronos OpenCL Working Group, “The OpenCL Specification - Version 1.0,”

tech. rep., The Khronos Group, 2009. http://www.khronos.org/opencl.

[8] K. Knobe and C. D. Offner, "TStreams: A Model of Parallel Computation (Preliminary Report)," Tech. Rep. HPL-2004-78, HP Labs, 2004.

[9] 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.

[10] R. Barik, Z. Budimlić, V. Cavé, S. Chatterjee, Y. Guo, D. Peixotto, R. Raman, J. Shirako, S. Taşırlar, Y. Yan, Y. Zhao, and V. Sarkar, "The Habanero multicore software research project," in *Proceeding of the 24th ACM SIGPLAN conference companion on Object oriented programming systems languages and applications*, OOPSLA '09, (New York, NY, USA), pp. 735–736, ACM, 2009.

[11] V. Cavé, J. Zhao, J. Shirako, and V. Sarkar, "Habanero-Java: the New Adventures of Old X10," in *9th International Conference on the Principles and Practice of Programming in Java (PPPJ)*, August 2011.

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

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

[14] M. Snir, S. Otto, S. Huss-Lederman, D. Walker, and J. Dongarra, *MPI: The Complete Reference.* MIT Press Cambridge, 1995.

[15] A. Wollrath, R. Riggs, and J. Waldo, "A Distributed Object Model for the Java System," in *Proceedings of the USENIX 1996 Conference on Object-Oriented*

Technologies, 1996.

[16] Blaise Barney, Lawrence Livermore National Laboratory, "POSIX Threads Programming." https://computing.llnl.gov/tutorials/pthreads.

[17] R. Chandra, L. Dagum, D. Kohr, D. Maydan, J. McDonald, and R. Menon, *Programming in OpenMP*. Academic Press, 2001.

[18] OpenMP Architecture Review Board, "The OpenMP API specification for parallel programming." http://openmp.org/wp.

[19] Blaise Barney, Lawrence Livermore National Laboratory, "OpenMP." https://computing.llnl.gov/tutorials/openMP.

[20] H. Richardson, "High Performance Fortran: history, overview and current developments," tech. rep., April 1996.

[21] G. Blelloch and P. R. Model, "Nesl: A Nested Data-Parallel Language," tech. rep., 1990.

[22] 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.

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

[24] "The Chapel Language Specification," tech. rep., February 2005.

[25] E. Allen, D. Chase, V. Luchangco, J.-W. Maessen, S. Ryu, G. L. S. Jr., and S. Tobin-Hochstadt, *The Fortress Language Specification.* Sun Microsystems, Inc., May 2006. http://projectfortress.java.net.

[26] L. Meadows, "Openmp 3.0 — a preview of the upcoming standard," in *Proceedings of the 3rd international conference on High Performance Computing and Communications*, HPCC '07, (Berlin, Heidelberg), pp. 4–4, Springer-Verlag, 2007.

[27] 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," in *Proceedings of the 20th annual ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, OOPSLA '05, (New York, NY, USA), pp. 519–538, ACM, 2005.

[28] "The Habanero C parallel programming language.." https://wiki.rice.edu/confluence/display/HABANERO/Habanero-C.

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

[30] Y. Guo, J. Zhao, V. Cavé, and V. Sarkar, "SLAW: a Scalable Locality-aware Adaptive Work-stealing Scheduler," in *24th IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, April 2010.

[31] Y. Yan, J. Zhao, Y. Guo, and V. Sarkar, "Hierarchical place trees: A portable abstraction for task parallelism and data movement," in *Proceedings of the 22nd Workshop on Languages and Compilers for Parallel Computing (LCPC)*, pp. 172–187, October 2009.

[32] Intel Corporation, Intel Software Network, “What If Experimental Software.” http://software.intel.com/en-us/whatif/.

[33] Shams, Imam and Habanero Research Programming Group, “CnC-Python.” http://cnc-python.rice.edu/.

[34] Imam, Shams and Sarkar, Vivek, “CnC-Python: Multicore Programming with High Productivity.” submitted, January 2012.

[35] “NVIDIA Corporation. The CUDA Zone – The resource for CUDA developers.” http://www.nvidia.com/cuda.

[36] Y. Yan, M. Grossman, and V. Sarkar, “JCUDA: A Programmer-Friendly Interface for Accelerating Java Programs with CUDA,” in *Euro-Par ’09: Proceedings of the 15th International Euro-Par Conference on Parallel Processing*, (Berlin, Heidelberg), pp. 887–899, Springer-Verlag, 2009.

[37] D. Sbirlea, J. Shirako, R. Newton, and V. Sarkar, “SCnC: Efficient Unification of Streaming with Dynamic Task Parallelism,” in *Data-Flow Execution Models for Extreme Scale Computing*, 2011.

[38] E. Ayguadé, R. M. Badia, F. D. Igual, J. Labarta, R. Mayo, and E. S. Quintana-Ortí, “An Extension of the StarSs Programming Model for Platforms with Multiple GPUs,” in *Proceedings of the 15th International Euro-Par Conference on Parallel Processing*, Euro-Par ’09, pp. 851–862, Springer-Verlag, 2009.

[39] C. Augonnet, S. Thibault, R. Namyst, and P.-A. Wacrenier, “StarPU: A Unified Platform for Task Scheduling on Heterogeneous Multicore Architectures,” in *Proceedings of the 15th International Euro-Par Conference on Parallel Processing*, Euro-Par ’09, (Berlin, Heidelberg), pp. 863–874, Springer-Verlag, 2009.

[40] C. Augonnet, S. Thibault, R. Namyst, and P. A. Wacrenier, “StarPU: a unified platform for task scheduling on heterogeneous multicore architectures,” *Concurrency and Computation: Practice and Experience*, vol. 23, pp. 187–198, February 2011.

[41] A. R. Putnam, D. Bennett, E. Dellinger, J. Mason, and P. Sundararajan, “CHiMPS: a high-level compilation flow for hybrid CPU-FPGA architectures,” in *Proceedings of the 16th international ACM/SIGDA symposium on Field programmable gate arrays*, FPGA ’08, (New York, NY, USA), pp. 261–261, ACM, 2008.

[42] M. D. Linderman, J. D. Collins, H. Wang, and T. H. Meng, “Merge: a programming model for heterogeneous multi-core systems,” *SIGPLAN Not.*, vol. 43, pp. 287–296, Mar. 2008.

[43] P. H. Wang, J. D. Collins, G. N. Chinya, H. Jiang, X. Tian, M. Girkar, N. Y. Yang, G.-Y. Lueh, and H. Wang, “EXOCHI: architecture and programming environment for a heterogeneous multi-core multithreaded system,” *SIGPLAN Not.*, vol. 42, pp. 156–166, June 2007.

[44] K. J. Brown, A. K. Sujeeth, H. Lee, T. Rompf, H. Chafi, M. Odersky, and K. Olukotun, “A Heterogeneous Parallel Framework for Domain-Specific Languages,” in *PACT ’11: 20th International Conference on Parallel Architectures and Compilation Techniques, October 2011*, 2011.

[45] H. Chafi, A. K. Sujeeth, K. J. Brown, H. Lee, A. R. Atreya, and K. Olukotun, “A domain-specific approach to heterogeneous parallelism,” in *Proceedings of the 16th ACM symposium on Principles and practice of parallel programming*,

PPoPP '11, (New York, NY, USA), pp. 35–46, ACM, 2011.

[46] 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.

[47] 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.

[48] M. Bro-nielsen and C. Gramkow, "Fast Fluid Registration of Medical Images," pp. 267–276, Springer-Verlag, 1996.

[49] K. Knobe and M. G. Burke, "The Tuning Language for Concurrent Collections," in *16th Workshop on Compilers for Parallel Computing*, 2012.

[50] R. D. B. et al, "CILK: An efficient multithreaded runtime system," *Proceedings of Symposium on Principles and Practice of Parallel Programming (PPoPP'95)*, pp. 207–216, July 1995.

[51] "The Java Grande Forum Benchmark Suite." http://www2.epcc.ed.ac.uk/computing/research_activities/java_grande.

[52] 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.

[53] A. Chandramowlishwaran, K. Knobe, and R. Vuduc, “Applying the Concurrent Collections Programming Model to Asynchronous Parallel Dense Linear Algebra,” in *Proceedings of the 15th ACM SIGPLAN Annual Symposium on Principles and Practice of Parallel Programming (PPoPP)*, (Bangalore, India), 2010.

[54] C. Bienia, S. Kumar, J. P. Singh, and K. Li, “The PARSEC benchmark suite: characterization and architectural implications,” in *Proceedings of the 17th international conference on Parallel architectures and compilation techniques*, PACT ’08, (New York, NY, USA), pp. 72–81, ACM, 2008.

[55] S. Che, M. Boyer, J. Meng, D. Tarjan, J. W. Sheaffer, S.-H. Lee, and K. Skadron, “Rodinia: A benchmark suite for heterogeneous computing,” in *Proceedings of the 2009 IEEE International Symposium on Workload Characterization (IISWC)*, IISWC ’09, (Washington, DC, USA), pp. 44–54, IEEE Computer Society, 2009.

[56] A. Sbirlea, Y. Zou, Z. Budimlic, J. Cong, and V. Sarkar, “Mapping a Dataflow Programming Model onto Heterogeneous Architectures.” submitted, January 2012.