DEPARTMENT OF COMPUTER SCIENCE

Task Data-flow Execution on Many-core Systems

Andreas Diavastos

A Dissertation Submitted to the University of Cyprus in Partial Fulfillment of the Requirements for the Degree of Doctor of Philosophy

November, 2017

VALIDATION PAGE

Doctoral Candidate: Andreas Diavastos

Doctoral Dissertation Title: Task Data-flow Execution on Many-core Systems

The present Doctoral Dissertation was submitted in partial fulfillment of the requirements for the Degree of Doctor of Philosophy at the Department of Computer Science and was approved on the ***November 13, 2017*** *by the members of the Examination Committee.*

Examination Committee:

Research Supervisor: ______________________
Associate Professor Pedro Trancoso

Committee Member: ______________________
Professor Paraskevas Evripidou

Committee Member: ______________________
Associate Professor Yanos Sazeides

Committee Member: ______________________
Professor Leonel Sousa

Committee Member: ______________________
Professor Dionisios N. Pnevmatikatos

DECLARATION OF DOCTORAL CANDIDATE

The present Doctoral Dissertation was submitted in partial fulfillment of the requirements for the degree of Doctor of Philosophy of the University of Cyprus. It is a product of original work of my own, unless otherwise mentioned through references, notes, or any other statements.

.... [Full Name of Doctoral Candidate]

.... [Signature of Doctoral Candidate]

Περίληψη

Η αύξηση της επίδοσης με ταυτόχρονη μείωση της κατανάλωσης ενέργειας επιτυγχάνεται με την παράλληλη επεξεργασία. Προκειμένου να αξιοποιηθεί ο παραλληλισμός μέσα από τις εφαρμογές και να βελτιστοποιηθεί η ενεργειακή απόδοση μεγάλων συστημάτων, η τάση είναι να αυξάνωνται οι πυρήνες μέσα στους επεξεργαστές. Ωστόσο, η αύξηση του αριθμού των πυρήνων από μόνη της δεν έχει ως αποτέλεσμα την βελτιωμένη επίδοση των εφαρμογών. Από την άποψη του λογισμικού, αυτό δημιουργεί νέες προκλήσεις καθώς χρειαζόμαστε ένα πλαίσιο που να μπορεί να εκμεταλλεύεται αποτελεσματικά τον παραλληλισμό των εφαρμογών στους διαθέσιμους πόρους που προσφέρει το υλικού.

Η αύξηση της επίδοσης στα μελλοντικά συστήματα πολλαπλών πυρήνων θα επηρεαστεί από τους ακόλουθους παράγοντες: τον βαθμό παραλληλισμού στις εφαρμογές, τον τρόπο προγραμματισμού, τα χαμηλού κόστους συστήματα εκτέλεσης, την εκτέλεση των εργασιών με γνώμονα την τοποθεσία των δεδομένων, την αποτελεσματική χρήση των διαθέσιμων πόρων και τα επεκτάσιμα σχέδια αρχιτεκτονικής επεξεργαστών.

Σε αυτήν την Διατριβή, το μοντέλο εργασιών (Task model) χρησιμοποιείται ως υλοποίηση του μοντέλου ροής δεδομένων (Data-flow). Το μοντέλο ροης δεδομένων είναι το καταλληλότερο μοντέλο για την εκμετάλλευση μεγάλων ποσοτήτων παραλληλισμού, ενώ η υλοποίηση με τη χρήση εργασιών μειώνει το κόστος της εκτέλεσης και προσαρμόζεται εύκολα σε διαφορετικές εφαρμογές. Σε αυτή τη εργασία, η Συναλλακτική Μνήμη (Transactional Memory) ενσωματώνεται στο μοντελο ροής δεδομένων για να μειώσει την αυστηρότητα του, διερευνώντας τον εικαστικό παραλληλισμό όταν οι εξαρτήσεις μεταξύ εργασιών είναι πολύ περίπλοκες για να εφαρμοστούν ή ακόμα και όταν δεν ισχύουν.

Αυτή η Διατριβή παρουσιάζει την πρώτη εφαρμογή σε μεγάλους πολυ-επεξεργαστές του μοντέλου πολλαπλών σπειρωμάτων (DDM), μια υλοποίηση του μοντέλου ροής δεδομένων. Το μοντελο πολλαπλών σπειρωμάτων επανασχεδιάζεται για να υποστηρίξει τον πρώτο επεξεργαστή πολλαπλών πυρήνων από την Intel(Single-chip Cloud Computer) που παρέχει ένα ενιαίο χώρο διευθύνσεων χωρίς υποστήριξη υλικού για συνεκτικότητα της κρυφής μνήμης (cache). Τα αποτελέσματα αυτής της εργασίας οδήγησαν στο σχεδιασμό και την ανάπτυξη ενός νέου πιο αποδοτικού, συστήματος εκτέλεσης που μπορεί να επεκταθεί σε ακόμα μεγαλύτερους επεξεργαστές.

Το προτεινόμενο σύστημα (SWITCHES) περιλαμβάνεται σε μια πλατφόμρα προγραμματισμού και εκτέλεσης εφαρμογών. Το SWITCHES είναι λογισμικό που υλοποιεί το μοντέλο ροής δεδομένων που βασίζεται σε εργασίες για επεξεργαστές πολλών πυρήνων. Απαιτεί ενιάιο χώρο διευθύνσεων αλλά όχι απαραίτητα μηχανισμούς συνεκτικότητας της κρυφής μνήμης που θα μπορούσαν να περιορίσουν την επεκτασιμότητα του επεξεργαστή. Το SWITCHES υλοποιεί ένα ελαφρύ στατικό κατανεμημένο σύστημα ενεργοποίησης για την επίλυση εξαρτήσεων κατά τη διάρκεια εκτέλεσης εργασιών. Υποστηρίζει μηχανισμούς κατανομής των πόρων του συστήματος και ενσωματώνει τεχνικές εκμάθησης μηχανών (machine-learning) για την αποτελεσματική αξιοποίηση τους. Για τη εύκολη υλοποίηση προγραμμάτων, το πλαίσιο υλοποιεί το πιο πρόσφατο πρότυπο από το (OpenMP) (v4.5) και το επεκτείνει για να υποστηρίζει εργασίες σε βρόχους με εξαρτήσεις σε διαφορετικούς βρόχους. Παρέχει ένα εργαλείο (Translator) που παράγει αυτόματα κώδικα βασισμένο σε νήματα, ο οποίος μπορεί να μεταγλωττιστεί από οποιονδήποτε μεταγλωττιστή C/C++, εφαρμόζοντας όλες τις υπάρχουσες βελτιστοποιήσεις.

Χρησιμοποιώντας το σύστημα αυτό, η επίδοση εφαρμογών με διαφορετικά χαρακτηριστικα σε μια μηχανή με Intel Xeon Phi επεξεργαστή ξεπερνά την επίδοση του OpenMP κατά μέσο όρο κατά 32% και βελτιώνει την αποδοτικότητα της πετυχαίνωντας μέγιστη επίδοση χρησιμοποιώντας 30% λιγότερους πυρήνες σε εφαρμογές με σύνθετες εξαρτήσεις.

Abstract

Power-performance efficiency in High Performance Computing (HPC) systems is currently achieved by exploring parallel processing. Consequently, in order to exploit application parallelism and optimize energy efficiency, the trend is to include more cores in the processors. From the hardware perspective, many small and simple cores will be added in processor architectures, leading towards many-core chips with hundreds of cores. Nevertheless, scaling the number of cores alone does not result in improved application performance. From the software perspective, this creates new challenges as we need a framework that can efficiently exploit application parallelism on the available hardware resources.

Overall, performance scalability in future many-core systems will be affected by the following factors: the degree of parallelism, programmability, low-overhead runtime systems, locality-aware execution, efficient use of the available resources and scalable architecture designs.

In this thesis, the Task model is used as an implementation of the Data-flow paradigm. Data-flow is the most appropriate model for exploiting large amounts of software parallelism, while a task-based implementation reduces runtime overheads and easily adapts to different applications. In this work, Transactional Memory is integrated in Data-flow to reduce the strictness of the latter by exploring speculative parallelism when task dependences are too complex to apply or even when not applicable.

This thesis presents the first many-core implementation of the Data-Driven Multi-threading (DDM) model, a task-based implementation of Data-flow. DDM is redesigned to support the first single-chip many-core processor from Intel (the Single-chip Cloud Computer) that provides a single address space with no hardware support for cache-coherence. The results from this work led to the design and development of a new more efficient, lightweight runtime system that is able to scale to larger many-core processors.

The proposed runtime system (called SWITCHES) is included in a complete programming and execution framework. SWITCHES is a software implementation of the task-based Data-flow model for many-core processors. It requires global address space but not necessarily a hardware cache-coherence mechanisms that could limit the scalability of the architecture. SWITCHES implements a lightweight distributed triggering system for runtime task dependence resolution and uses static scheduling and compile time assignment policies to reduce overheads. It supports explicit task resource allocation mechanisms and incorporates machine-learning techniques within the framework to efficiently utilize the underlying resources. To maintain high-levels of programming productivity, the framework implements the latest API standard from OpenMP (v4.5) and extends it to support variable granularity loop-tasks with dependences across different loops as to favor data-locality in loops with inter-dependences. It provides a source-to-source tool that automatically produces thread-based code that can be compiled by any off-the-shelf C/C++ compiler, applying all existing optimizations.

Performance evaluation of applications with different characteristics on an Intel Xeon Phi system shows good scalability that surpasses the state-of-the-art by an average of 32% and resource utilization is increased with maximum performance achieved using 30% fewer cores for applications with complex dependences.

Acknowledgments

First of all, I would like to my advisor, Pedro Trancoso, for his continuous support and guidance throughout the years we worked together. Thank you for teaching me how to understand, how to review and how to do research. Thank you for teaching me how to behave and how to react to criticism and how to use it to become a better researcher and a better person. Thank you for being my mentor.

I would like to thank my parents, Yiannakis and Lenia and my sister Anthi for their unconditional love and support, through this long process. For understanding and withstanding, I dedicate this thesis to you.

A big thank you to all my friends for being there for me, whenever needed and for whatever needed. Thank you for bearing and tolerating me all these years and for being there in the good and the not so good times.

I would of course like to thank all the Casper group colleagues that I have had the pleasure to work with and produce research with, Kyriakos, Demos, Marios, Panayiotis, Constantinos and Giannos. Also, thank you to all the colleagues at the Department for all the conversations and discussions, work- or and work-related.

A special thank you to Constantinos Costa for the endless discussions during lunches and dinners at work. We 've come a long way and learned a lot since the day we started. Now it's your turn.

To everybody at the Department of Computer Science, thank you for the excellent cooperation all these years. Special thanks to Melina and Savvoula for taking care of us. Big thanks to the IT team, Savvas, Maria, Andry, Andreas and Loizos for providing all the support and even more.

To Giannis Koutsou and the operations team of the Cyprus Institute, George, Andreas and Thekla - a Big thank you for supporting the Phi cluster. Most of my experiments were done on the Phi cluster and with their support.

"I expect nothing and I accept everything."

"Treat others as you would want to be treated."

Contributions of this Thesis

Journal publications:

1. **A. Diavastos** and P. Trancoso. *"SWITCHES: A Lightweight Runtime for Dataflow Execution of Tasks on Many-Cores"*, ACM Transactions on Architecture and Code Optimization (TACO) 14, 3, Article 31, pp. 1-23, August 2017.

2. **A. Diavastos**, P. Trancoso, M. Luján, I. Watson, *"Integrating Transactions into the Data-Driven Multi-threading Model using the TFlux Platform"*, International Journal of Parallel Processing (IJPP) 44(2): 257-277, April 2016.

Conference and workshop proceedings:

3. **A. Diavastos** and P. Trancoso, *"Auto-tuning Static Schedules for Task Data-flow Applications"*, in Proceedings of the 1st ACM Workshop on AutotuniNg and aDaptivity AppRoaches for Energy efficient HPC Systems, (ANDARE), co-located with Parallel Architectures and Compilation Techniques (PACT), pp. 1-6, Portland, Oregon, USA, September 2017.

4. **A. Diavastos**, G. Stylianou, P. Trancoso, *"TFluxSCC: Exploiting Performance on Future Many-Core Systems through Data-Flow"*, in Proceedings of the 23rd Euromicro International Conference on Parallel, Distributed and Network-based Processing (PDP), pp. 190-198, Turku, Finland, March 2015.

5. **A. Diavastos**, P. Trancoso, M. Luján and I. Watson, *"Integrating Transactions into the Data-Driven Multi-threading Model using the TFlux Platform"* in Proceedings of the Data-Flow Execution Models for Extreme Scale Computing (DFM) Workshop, pp. 19-27, Galveston, Texas, U.S.A., October 2011.

Technical reports:

6. **A. Diavastos** and P. Trancoso, *"Unified Data-Flow Platform for General Purpose Many-core Systems"*, Department of Computer Science, University of Cyprus, Nicosia, Cyprus, Technical Report UCY-CS-TR-17-2, pp. 1-22, September 2017.

7. **A. Diavastos**, G. Matheou, P. Evripidou and P. Trancoso, *"Data-Driven Multi-threading Programming Tool-chain"*, Department of Computer Science, University of Cyprus, Nicosia, Cyprus, Technical Report UCY-CS-TR-17-3, pp. 1-18, September 2017.

Other Contributions beyond the scope of this thesis:

8. **A. Diavastos**, P. Petrides, G. Falcao and P. Trancoso, *"LDPC Decoding on the Intel SCC"* in Proceedings of 20th Euromicro International Conference on Parallel, Distributed and Network-Based Computing (PDP), pp. 57-65, Garching, Germany, February 2012.

Contents

List of Figures

List of Tables

Glossary

Data-flow The Data-flow Model of Execution

Thread A software unit of execution (*i.e.* a pthread).

Task A unit of work to be executed (*e.g.* a loop iteration).

Performance How fast a task finishes in relation to execution time.

Runtime Runs during execution of an application.

Many-core A processor with 100s of hardware execution units.

Scalability Increase in relation to the number of hardware units.

Asynchronous Tasks execute freely without interactions with each other.

Granularity The scale or size of a task.

Off-the-shelf Taken from existing/commodity systems.

Source-to-source Takes as input and produces as output a source code.

Auto-tuning Automatically tunes the execution without programmer intervention.

TFluxHard The hardware implementation of the TFlux runtime system.

TFluxSoft The software implementation of the TFlux runtime system.

TFluxTM The implementation of the TFlux Platform that includes support for Transactional Memory.

TFluxSCC The implementation of the TFlux Platform for the Intel SCC processor.

SWITCHES The system that we propose and includes a runtime scheduler and a programming platform.

HPC High Performance Computing

GPU Graphic Processing Unit

MIC Many Integrated Core

DDM Data-Driven Multithreading

TM Transactional Memory

API Application Programming Interface

SCC Single-chip Cloud Computer

NLA Numerical Linear Algebra

SG Synchronization Graph

TSU Thread Scheduling Unit

PPE Power Processor Element

SPE Synergistic Processor Element

DMA Direct Memory Access

DDMVM Data-Driven Multithreading Virtual Machine

FPGA Field-Programmable Gate Array

DDMCPP Data-Driven Multithreading C Pre-Processor

SMCC Software Managed Cache Coherence

LDPC Low-Density Parity Check

FEB Full/Empty Bits

OCR Open Community Runtime

MPB Message Passing Buffer

MIU Message Interface Unit

LUT Lookup Table

MC Memory Controller

SPMD Singe Program Multiple Data

KNC Knights Corner

GA Genetic Algorithm

NSGA Non-dominated Sorting Genetic Algorithm

GAC Genetic Algorithm Component

TSX Transactional Synchronization Extentions

Chapter 1

Introduction

Improving application performance in an efficient and effective way is a joint task between both the hardware and the software. Performance scaling is determined by a combination of the following factors: the degree of parallelism, programmability, low-overhead runtime systems, locality-aware execution, efficient use of the available resources and scalable architecture designs. In this thesis we focus on the Task Data-flow model as the most appropriate for exploiting large amounts of parallelism and explore ways to address the above mentioned factors in order to scale application performance on commodity hardware, from conventional multi-cores to future many-cores with hundreds of cores. Our efforts brought the implementation of parallel programming and execution platform that is publicly available for download in [1].

1.1 Motivation

Performance improvement is the principal driver in High Performance Computing (HPC). This improvement was achieved by increasingly adding more nodes to the large-scale systems. Nevertheless, as suggested by M. Resch [2], the number of nodes in future exascale systems may not change dramatically compared to those of today because of the prohibitive cost. It is the number of cores in a single node that will increase with the use of single-chip many-core processors leading to large-scale HPC machines with millions of processing units. Most HPC facilities today use many-core processors such as the Graphic Processing Units (GPUs) [3] and the Intel Many Integrated Core (MIC) processors [4] as to achieve high-performance within a certain power budget. Nevertheless, programing a million units is an extremely difficult task as can be seen in HPC centers today. Message-passing models are most commonly used but they will become a bottleneck for intra-node performance as they are designed for only a few thousands of processing units and their programming is already a challenging task. Therefore, new parallel programming frameworks and runtime systems are needed to support the future HPC large-scale machines [2].

1.2 Problem Statement

There has recently been a renewed interest in the Data-flow model as a way to efficiently exploit large-scale parallel computation. The work in this thesis is based on the task-based Data-flow model of execution that is the most appropriate for exploiting large amounts of parallelism. Nevertheless, existing runtime systems offer limited performance scalability (see Chapter 2). The main factors that affect the scaling of application performance on large-scale many-core systems are: the degree of parallelism, programmability, low-overhead runtime systems, locality-aware execution, efficient use of the available resources and scalable architecture designs. To address these factors and achieve performance that scales in the many-core era it is essential to answer the questions presented in the following sections.

1.2.1 Application Parallelism

The Data-flow model is an asynchronous (non-blocking) model capable of exploiting large amounts of parallelism as the execution follows the path of the data, thus a natural paradigm for expressing parallelism. Whenever the input operands of an instruction in a Data-flow program is produced, the instruction can be executed. Despite the simplicity of the idea, extracting producer/consumer relationships in applications described by complex algorithms is proven to be a difficult task, even for highly experienced programmers. Partly, this is because of the strictness of the model as it doesn't allow shared state in Data-flow programs, even in cases where this is a fundamental operation. Therefore, a different approach must be followed in order to keep the benefits of the Data-flow model and at the same time not burden the users with the difficult task of extracting all dependences from complex algorithms.

Research Question 1: *Is it possible to relax the definition of dependences and increase the parallelism exploited, without loosing the benefits of Data-flow?*

1.2.2 Programmability

Message-passing models that are most commonly used in HPC systems will become a bottleneck for intra-node performance as they are designed and optimized for large-scale distributed systems [2, 5]. But even so, programing a million units is an impossible task as can be seen in HPC centers today [2]. Although HPC users are highly experienced programmers, there is a lack of compelling motivation for switching programming environments unless the performance gains are high and the development effort is low [5]. Therefore, existing parallel execution frameworks need to be extended for large-scale many-core systems and provide user-friendly programming tools, already familiar to the HPC community.

Research Question 2: *Is it possible to provide the programming community with tools that can exploit large degrees of parallelism using well known language constructs based on highly adopted environments?*

1.2.3 Scalable Architectures

Shared-memory multi-core processors with powerful cores have been holding the largest share in the HPC industry the past decade. The architectural design of such processors is guided (among others) by its ability to produce more performance with legacy software. Parallel processing has become the de-facto standard for increasing application performance and in order to continue and optimize power-performance efficiency in such systems the trend today is to include more cores in a single processor [2]. From the hardware design perspective, many simple cores will be added, leading towards single-chip large-scale many-core processors. But, as the number of cores in a processor gradually increases, simpler designs must be explored by the vendors in order to reduce hardware costs and improve energy-efficiency [6]. Therefore, lightweight cores will be included and expensive mechanisms such as cache-coherence might be removed from future many-core designs as to allow the scalability of the architectures to continue [7, 8].

Research Question 3: *How do low-cost scalable architectures with many lightweight cores impact runtime systems and application performance?*

1.2.4 Scalable Runtime Systems

Scaling the number of cores alone does not result in reduced execution time. From the software perspective, many-core processors create new challenges as there is a need for scheduling systems that can efficiently exploit parallelism without incurring runtime overheads and are able to increase speedup regardless of the number of cores. Most Data-flow runtime systems that exist today (presented in Chapter 2) follow a centralized approach, creating a single-point of communication for parallel tasks that will become a bottleneck on large-scale processors. A centralized runtime implementation implies sharing of scheduling data structures that requires locking and coherence mechanisms in order to ensure correct execution. As previously explained in Section 1.2.3, the use of such techniques results in large performance overheads and limits the scalability of both the software and the hardware [6]. Also, almost all current runtime systems use dynamic scheduling that provides adaptability to irregular executions, but results in large runtime overheads, especially in applications with complex dependences on large-scale many-cores.

RESEARCH QUESTION 4: *Is it possible to develop a low-overhead runtime system that scales regardless of the number of cores and requires minimum hardware support?*

1.2.5 Efficient Utilization of Resources

It is particularly common in large-scale systems to have under-utilized hardware resources that result in wasted performance and energy [5]. Efficiently mapping tasks from a parallel application with irregular data access patterns and complex dependences to a large-scale system is an extremely difficult task, even for highly experienced programmers. A runtime system with a dynamic scheduling policy could be a possible solution to such a problem but as explained earlier, finding a better scheduling at runtime incurs in significant overheads that will not scale on future many-core processors. Therefore, new scheduling techniques and tools are needed that can utilize large number of hardware resources and provide optimized power-performance efficiency.

RESEARCH QUESTION 5: *Is it possible to develop a tool that automatically provides scheduling policies, able to efficiently utilize the underlying resources?*

1.3 Thesis Statement

In order to achieve application performance scalability for processors with large number of cores (many-cores) you need to collectively address parallelism, programmability, runtime system, resource utilization, and architecture design limitations.

1.4 Objectives and Contributions

1.4.1 Goal

The main goal of this thesis is to address the factors that affect the scalability of application performance on large-scale many-core processors. To achieve this goal, the task-based Data-flow model is used and in particular the Data-Driven Multi-threading (DDM) [9] model, a higher-granularity execution model of the Data-flow paradigm. In order to hide scheduling and communication latencies, DDM applies dependences across threads (collection of instructions) instead of single instructions as in the original implementations of Data-flow. To present our findings we evaluate our implementations with applications from different domains that exhibit the characteristics we want to show for each objective. The objectives and contributions of this Thesis are stated below.

1.4.2 Objective 1: Exploit more application parallelism

To answer *Research Question 1* (Section 1.2.1), the Data-flow model is to be extended to support speculative parallelism by incorporating Transactional Memory (TM). More precisely, explore shared mutable data by integrating transactions in Data-flow in order to execute more tasks in parallel that would otherwise be serialized. Therefore, further parallelism can be exploited while mutable shared state can be introduced to the Data-flow model in a composable way.

CONTRIBUTION: This work is the first software integration of transactions into a task-based Data-flow implementation as a way to introduce shared state in the Data-flow model. It offers software support for developing Data-flow applications with TM support [10], while it provides an overhead analysis of the integration of the two models [11]. More details on the implementation and results of this work are presented in Chapter 3.

1.4.3 Objective 2: Programmability

In order to address *Research Question 2* (Section 1.2.2) and maintain high-levels of programming productivity, a unified Data-flow platform is to be developed that will support different DDM runtime systems and different types of hardware processors [12,13] (see Chapter 2). To provide for easy to program environments, the most commonly used Application Programming Interface (API), the OpenMP v4.5, must be implemented.

CONTRIBUTION: This work extends the OpenMP v4.5 API to support explicit task resource allocation mechanisms and variable loop task granularity to increase data-locality even for loop tasks with inter-dependences. It provides a source-to-source tool that automatically produces thread-based code that can be compiled by any off-the-shelf C/C++ compiler, applying all existing optimizations [14]. Details on the API supported and the extensions implemented are presented in Chapter 5.

1.4.4 Objective 3: Support for Scalable Architectures

To address *Research Question 3* (Section 1.2.3), the DDM model [9], is to be ported on the Intel Single-chip Cloud Computer (SCC) many-core system [15]. Intel SCC is an experimental processor for many-core software development that does not provide support for cache-coherence in order to scale the architecture design to a large number of cores.

CONTRIBUTION: The work presented in Chapter 4 is the first DDM implementation for a many-core processor [16]. It provides low-overhead software support for shared-memory execution without requiring hardware cache-coherence mechanisms as to avoid factors that can limit the scalability of the architectures [6–8].

1.4.5 Objective 4: Low-overhead Runtime

To answer *Research Question 4* (Section 1.2.4) a new runtime system that implements the task-based Data-flow model, is to be designed and developed in order to exploit large degrees of parallelism and at the same time addresses the trade-offs of *Research Question 3* (Section 1.2.3) as to be more efficient, lightweight and scale regardless of the number of cores used.

CONTRIBUTION: This work proposes a software runtime system for many-core processors that supports global address space without the need for hardware cache-coherence mechanisms [14]. It implements a lightweight distributed triggering system for task runtime dependence resolution and uses static scheduling and compile time assignment policies to reduce overheads. More details on the implementation of the runtime system are presented in Chapter 5.

1.4.6 Objective 5: Efficient Utilization of resources

To address the issue of under-utilized execution presented in *Research Question 5* (Section 1.2.5), we will employ a machine-learning technique to automatically map tasks to the underlying hardware. Using machine-learning, the scheduling policy can be automatically optimized for execution time, power consumption, temperature or any combination of the three. At the same time, the hardware can be utilized to achieve maximum performance using fewer resources than what common scheduling policies use.

CONTRIBUTION: The main contribution of this work is the integration of a machine-learning algorithm in a real parallel framework that produces an auto-tuning scheduling tool for task-based parallel applications [17]. Details on the implemented auto-tuning tool are presented in Chapter 6.

Chapter 2

Background

The work presented in this thesis is based on the Task model, where tasks are scheduled in a Data-flow fashion. More precisely, we use the DDM model of execution [9] as a baseline for this work. In the following sections the background of this work that is related to the Data-flow model is presented along with the software runtime systems that implement it (Section 2.1), the DDM model and its implementations to date (Section 2.2), the TFlux Platform [18] that is the framework used as a starting point for this work (Section 2.3) and finally software frameworks that exist today for various HPC-based many-core systems (Section 2.5). The DDM model is also implemented as a hardware runtime unit [18, 19] but these implementations are out of the scope of this work as we focus on a software solution that can execute on commodity hardware.

2.1 The Data-flow Model

The original Data-flow model was proposed by Jack Dennis in the early 1970s [20, 21] as an alternative to the Control-flow (von Neumann) model. Instructions in a Data-flow program are executed when all their input operands are available, creating an asynchronous (non-blocking) execution. The availability of the operands is expressed using data dependences, that define a Data-flow graph representing the order of the execution. Using the Data-flow graph and the input operands required by each instruction, one can expose parallelism in a program. Many systems today try to explore fine-grain parallelism by using Data-flow-like models as a way to achieve high performance and utilization on large-scale many-core systems with hundreds to thousands of cores [22–25].

The main advantage of the Data-flow model is the ability to exploit maximum parallelism from an application by exposing fine-grain tasks. This large degree of parallelism can be exploited to hide the latency of memory accesses. Unlike other models, Data-flow does not require synchronization mechanisms as the correctness of the execution is assured by enforcing the data dependences. Nevertheless, exploiting fine-grain parallelism was also the limiting factor in the success of this model in past implementations, mostly due to the overheads in enforcing the data dependences at the instruction level. More recent attempts managed to overcome these overheads by adopting the model at a coarser granularity (*e.g.* tasks), consequently achieving high performance [9, 22–24].

Another relevant factor towards using Data-flow for increasingly large systems is its disciplined access to shared data. Assuming a task-based implementation of Data-flow, it is ensured by the model that no concurrent tasks will be modifying the same data, as this would result in a data dependence violation [20, 21, 26][1]. Therefore, in a shared-memory system the Data-flow model doesn't require a hardware implementation of a cache-coherence protocol as access on shared data may be coordinated by the model itself. Correctness of the application may be assured simply by updating cached data to and from main memory on completion of a task, *i.e.* flushing updated values to memory and invalidating cached copies in other cores. The fact that hardware cache-coherence is not required by the Data-flow model, allows for increasing performance scalability on many-cores as shown in [16], reducing hardware costs and improving energy-efficiency [6].

2.1.1 Data-flow Runtime Implementations

Recently, there has been renewed interest in the Data-flow approach to computation that was pioneered in late 70s and 1980s [27–31]. These projects demonstrated that it was feasible to express sufficiently large amounts of parallelism but a significant problem was how to throttle and schedule it. Subsequent projects, like DDM [9], OmpSs [22], etc. showed that coarsening granularity (from instructions to tasks) could result in more a controllable and efficient parallel execution.

Numerical Linear Algebra (NLA) is one area where Data-flow ideas have recently

[1]This is according to the strict Data-flow definition while in Chapter 3 a relaxed version that uses Transactional Memory to handle the use of shared context between tasks is explored.

Table 2.1: *Data-flow implementations on multi-cores from the literature.*

Description	*OmpSs* [22]	*Triggered Instructions* [32]	*Serialization Sets* [33]	*OpenDF* [34]
Implementation	Software	Hardware	Software	Hardware
Scheduling	Dynamic	Dynamic	Dynamic	-
Memory Model	Shared	Shared	Shared	-
Cache-coherence	Yes	No	Yes	-
Dependences	Pragma Directives	Inserted triggers	Writable and read-only variables	-
Parallelism	Task-based	Spatial	Serialization sets of dependent operations	Data-flow Instructions

Description	*DTT/CDTT* [35]	*SEED* [36]	*Statically Sequential* [37]	*WaveScalar* [38]
Implementation	Software	Hardware	Software	Hardware
Scheduling	Dynamic	Dynamic	Dynamic	Dynamic
Memory Model	Shared	Shared	Shared	Shared
Cache-coherence	-	-	Yes	Yes
Dependences	Macro-based triggers	-	Functions, shared objects, read/write sets, sequential segments	tokens/tags
Parallelism	Data-flow	Hybrid Data-flow + von-Neumann	Statically sequential programs	Data-flow instructions

been adopted. This is apparent in both LAPACK and BLAS functionality (PLASMA project [39]), for sparse matrices [40]. NLA constitutes one of the main kernels for scientific computing and their main next challenge is how to scale to petaflop-scale high performance systems. The new generation of NLA algorithms are moving towards expressing parallelism but leaving the scheduling to the runtime trying to harness the available combination of resources (multi-cores, many-cores, clusters, GPUs). It has been demonstrated that a Data-flow execution using Parallel Linear Algebra for Scalable Multi-core Architectures (PLASMA) can easily generate millions of tasks. Convolutional networks in computer vision [41–43] have been using GPUs for performance exploitation but it has recently been investigated for parallelism exploitation on single shared memory CPU machines as the task dependency graph implies linear speedup in within the PRAM model of parallel computation [44].

There is a clear relation between Data-flow computations and parallelization of functional languages. Another highly prominent use of functional programming techniques can be observed in the MapReduce frameworks [45]. Provided the map and reduce operations are side-effect free, we can automatically parallelize their execution using a Data-flow approach.

Tables 2.1 and 2.2 present recent work from the literature that use Data-flow concepts in programming multi-core systems. The tables present the type of each implementation (software or hardware), the scheduling policy used by each runtime system, the underlying memory model, whether it requires hardware support for cache-coherence, the way dependences are expressed and the type of parallelism exploited in each system. We focus our attention on OmpSs [22], SWARM [24] and Intel TBB [23], as they are the closest to the model we use and the runtime we implement in this thesis.

OmpSs is a software task-based programming model based on the OpenMP [46] and the StarSs [47]. Its target is heterogeneous multi-core architectures, thus it incorporates the use of OpenCL and CUDA kernels. OmpSs outperforms OpenCL or OpenMP in some applications for the same platforms while it offers a more flexible programming environment to exploit multiple accelerators. The basic differences of OmpSs with what is proposed in this work is that the dependences of the parallel tasks are resolved at runtime and the scheduling is decided by the scheduling unit also at runtime. OmpSs is also built as a shared memory model, thus it needs a hardware cache-coherence mechanism in order to provide correct execution.

Intel TBB is another software implementations that uses Data-flow concepts for scheduling parallel tasks. TBB is following a dynamic scheduling policy and is using a task stealing approach to distribute the tasks to the available cores. The programming model of TBB is using macro-statements in a C++ environment that makes the programming non-trivial compared to other systems with the more programmer-friendly compiler directives.

Finally, SWARM is an implementation of a Data-flow system for both shared and distributed memory systems. It uses a dynamic scheduling policy with a runtime system design that doesn't require hardware support for cache-coherence. The programming of SWARM is done using C-based macros that require from the programmer to re-write the entire application to be compatible with the runtime system

Table 2.2: *Data-flow implementations on multi-cores from the literature (cont.).*

Description	*SWARM* [24]	*Intel TBB* [23]	*CnC* [48]	*Maxeler* [49]
Implementation	Software	Software	Software	Hardware
Scheduling	Dynamic	Dynamic	Dynamic	Static
Memory Model	Hybrid	Shared	Shared	-
Cache-coherence	No	Yes	Yes	-
Dependences	Codelet macros	Explicit task dependence macros	Input/Output declaration	Input/Output variables
Parallelism	Codelets	Concurrent Containers	Data-flow and Stream-processing	Data-flow, Spatial and Stream

of SWARM.

TERAFLUX [25] was a project funded by the European Union aiming to solve the challenges of programmability, manageable architecture design and reliability of a 1000+ core chips by using the Data-flow principles. The idea was to develop new programming models, compiler analysis and optimization technologies in order to build a scalable architecture based mostly on off-the-shelf components while simplifying the design of such Tera-device systems. TERAFLUX used the TFlux platform as a programming and execution system for DDM programs on the proposed machine.

2.2 The Data-Driven Multi-threading model

DDM is a Data-flow model where the granularity of the Data-flow code is a thread and the synchronization part of the program is separated from the communication part as to overcome the synchronization and communication overheads imposed by the dynamic scheduling process [9, 50] on multi-core execution. DDM programs are composed of Data-Driven Threads (DThreads) that contain an arbitrary number of instructions. Within a DThread the instruction execution follows the classic control-flow model, thus allowing any other runtime or compile-time optimizations to be performed. The programming of the DDM model is done explicitly by the programmer by defining the DThreads in a program and the dependences between them, either by declaring a direct dependence on other DThreads or by declaring the

Table 2.3: *Data-Driven Multithreading implementations.*

Description	D^2NOW	$DDMVM_c$	$DDMVM_s$	$DDMVM_d$	$DDMVM_{FPGA}$
Date Introduced	1999	2011	2011	2013	2014
Implementation	Hardware	Software	Software	Software	Hardware
Scheduling	Static	Static and Dynamic	Static and Dynamic	Static	Static and Dynamic
Memory Model	Distributed	Distributed	Shared	Shared and Distributed	Shared
Cache-coherence	-	No	Yes	Yes	No
Dependences	Macros	Macros	Directives	Macros	Directives

inputs and outputs of the DThreads. The dependences and the inputs/outputs form a producer/consumer relationship between the DThreads in a program and their combination creates the Synchronization Graph (SG) of the DDM program. The parallel execution of the all DThreads in a DDM program is managed by a centralized Thread Scheduling Unit (TSU) that uses the SG in order to correctly synchronize the firing of ready DThreads and the update of waiting DThreads.

Table 2.3 presents all DDM systems implemented to date. D^2NOW [51] is a hardware implementation for a distributed system of single-core nodes. D^2NOW is the first DDM simulated hardware distributed implementation that also incorporated the CacheFlow technique, a deterministic data prefetching scheme using data-driven caching policies [52]. DDMVM$_c$ is also a DDM implementation for a heterogeneous processor (the IBM CELL/BE). DDMVM$_c$ uses a centralized TSU that is executed on the Power Processor Element (PPE) of the processor, while the Synergistic Processing elements (SPEs) were used for executing the application threads. The communication between the TSU and the application threads is done explicitly using Direct Memory Access (DMA) commands. DDMVM$_c$ was the first to implement a software version of the CacheFlow technique for moving data close to the cores prior to their reference. DDMVM$_c$ is also the first implementation of DDM that uses a dynamic scheduling policy for assigning the parallel threads to the cores of the system. The same implementation was also ported to SMP processors with shared memory and was called DDMVM$_s$ [53]. As DDMVM$_s$ is using shared memory to store its scheduling data structures and requires hardware cache-coherence support and simultaneous access protection mechanisms (locks) to protect the shared data structures from simultaneous from parallel DThreads. DDMVM was implemented

Table 2.4: TFlux DDM implementations.

Description	*TFluxSoft*	*TFluxHard*
Implementation	Software	Hardware
Scheduling	Static	Static
Memory Model	Shared	Shared
Cache-coherence	Yes	Yes
Dependences	Directives	Directives

in software for a distributed system (DDMVM$_d$ [54]) and became the first software distributed implementation of DDM. The scheduling unit was hierarchically distributed to the nodes with one TSU instance on each participating node. To measure the actual overheads and communication costs DDM was implemented on an FPGA unit. DDMVM$_{FPGA}$ [19] is the first implementation of DDM on real hardware with a TSU connected on the bus of the processor for communicating with the cores of the system. DDMVM$_{FPGA}$ also uses shared memory for storing data but as the entire system is implemented in hardware, no cache-coherence mechanism is needed.

2.3 The TFlux Platform

The TFlux Platform [18] is another system implementation of the DDM model that was developed with both software (TFluxSoft) and hardware (TFluxHard) implementations for the scheduling unit. It uses the shared-memory model and follows a static scheduling policy for assigning DThreads to the execution kernels. TFlux is the first DDM implementation of a SMP system on commodity hardware and the first to implement a more user-friendly programming interface with compiler pragma-directives. As shown in Table 2.4, both TFlux implementations use static assignment of DThreads, while they dynamically monitor dependences resolution and release ready DThreads for execution. Both use a centralized scheduling unit and require hardware cache-coherence in order to provide a safe execution environment.

TFlux is a complete platform that includes a programming environment for DDM applications using compiler directives, a source-to-source preprocessor that translates the application augmented with the directives into DDM parallel code and a TSU that handles the Data-flow execution of the DThreads at runtime. An important advantage of TFlux is that it is not built for a specific machine but rather

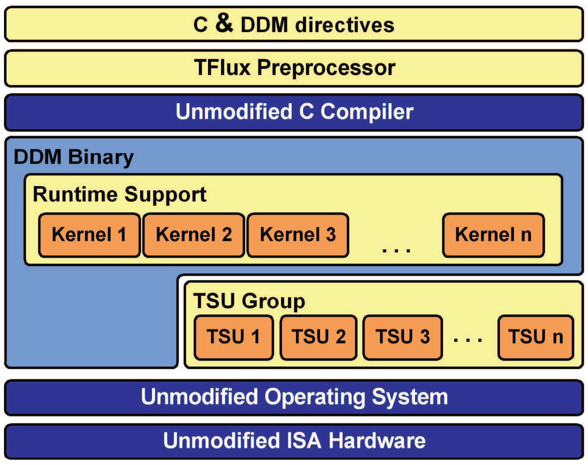


Figure 2.1: The layered design of the TFlux system [18].

works as a virtualization platform for DDM program execution on a variety of computing systems. Figure 2.1 shows the layered design of the TFlux system. To abstract details of the underlying hardware, the programmer uses only the top layer to develop DDM applications.

TFlux also requires a TSU to enforce the Data-flow execution of the DThreads. The TSU loads the SG of a DDM application, initiates and schedules the execution of all DThreads in a Data-flow manner, according to the dependences described by the SG. In the first implementation of DDM, the *D^2NOW* [9], each processor needed to have its own private TSU since the execution nodes were independent machines. In the TFlux implementation the TSUs were unified in a single unit named the *TSU Group*. This unit is logically split in *n+1* parts. One part per core (totaling *n*) for the core's own TSU operations and one common part which is located on a dedicated core and manages the common operations of the TSU for all cores.

In addition it is necessary to define the inputs and outputs of a thread or the producer and consumer relationships between the DThreads in order for the TFlux preprocessor to produce the SG of the application and print the parallel source. The TSU also manages the counters that control the firing of threads. Each time a producer thread terminates its execution, the consumer's counter is decremented by one. When the counter reaches zero, all needed results have been produced and thus the thread is ready for execution. All these operations are part of the TFlux runtime system, which includes all data structures required to manage the thread scheduling as well as the scheduling code itself. In TFluxHard, the TSU is implemented as a separate unit attached to the processor, while in TFluxSoft the execution of the TSU

Table 2.5: TFlux DDM pragma directives [55].

DDM Pragma Directives	Description

`#pragma ddm startprogram`
`#pragma ddm endprogram`
	Define the start and the end of a DDM program

`#pragma ddm block` *ID* `#pragma ddm endblock`	Define the start and the end of a block of threads with identifier *ID*

`#pragma ddm thread` *ID* `kernel` *NUMBER* `import(VAR : FROM) export(type :  VAR)`
`#pragma ddm endthread`
	Define the boundaries of a DDM thread with identifier *ID* and the kernel *NUMBER*

`#pragma ddm for thread` *ID* `depends(ID)`
`#pragma ddm endfor`
	Define the boundaries of a DDM loop thread with identifier *ID*

`#pragma ddm kernel` *NUMBER*	Declare the number of kernels to be used
`#pragma ddm var` *TYPE NAME*	Declare a shared variable with *NAME* and *TYPE*
`#pragma ddm private var` *TYPE NAME*	Declare a private variable with *NAME* and *TYPE*

is explicitly handled by one core of the multi-core system.

2.3.1 TFlux Programming Tool-chain

DDM applications in TFlux are developed using ANSI-C with pragma compiler directives [55]. The directives are used to define the code of the DThreads and to express the dependences between them. This section presents the programming style for developing DDM application with TFlux. This is supported by the DDM C Pre-Processor (DDMCPP) [55] that is included in the TFlux Platform. The main objective of the directives is to allow the programmer to define the boundaries, the type and the dependences of all DThreads in a DDM application.

Table 2.5 shows the most relevant directives used to write a DDM program. A simple thread is defined by enclosing its code in a `#pragma ddm thread` and a `#pragma ddm endthread`. These directives mark the start and the end of a thread and also define its unique identifier. Because TFlux supports a static scheduling technique, the programmer must also define the `kernel` that each thread will execute on. The definition of loops is supported in a similar way. By enclosing the code of a for loop in `#pragma ddm for` and `#pragma ddm endfor` directives, all iterations of the loop will be executed in parallel. The `kernel` construct is not necessary in loop threads as iterations will be evenly distributed to all participating kernels.

In order for the runtime system to know when a thread is ready for execution a counter variable is kept that denotes the number of consumers a thread is waiting upon being scheduled for execution. This counter is called *readycount*. For TFlux implementations (TFluxSoft and TFluxHard) this field is not mandatory, as the number of consumers will be automatically inferred by the declaration of the

Table 2.6: DDMVM pragma directives extensions [13].

$DDMVM_s$ Constructs	Description
`kernel(SCHED_POLICY:SCHED_VALUE)`	Define the scheduling policy that the specified thread will follow
`arity NUMBER`	Define the number of nested loops that a thread represents
`readycount NUMBER`	Explicitly define the number of consumers of thread
`update(THREAD_ID:START:END)`	Update the ending threads' consumers
`import(address:size:flag:expression:reference_variable)`	Define the variables that will be imported in a thread
`export(address:size:flag:expression:reference_variable)`	Define the variables that will be exported by a thread

explicit dependences.

To define the producer/consumer relationships between threads in a DDM application the data that the threads consume and produce must be considered. Using the `import` and `export` statements on a thread directive the preprocessor will know which variables each thread will consume and produce. Thus, it will automatically create a dependence between the thread that produced the declared variable and the thread that will consume it. In some cases expressing the data dependences through the produced and consumed data is not possible, such as arrays elements. To explicitly define dependences between threads the `depends` construct is used. This construct denotes the producers of a thread. Note that for a loop thread only the `depends` statement can be used to define the producers of that loop.

Any type of thread in a DDM program must be enclosed in a DDM block at all times. This can be done by enclosing the definitions of threads in a set of `#pragma ddm block` and `#pragma ddm endblock` directives. Any number of blocks can be used. The threads within a block will be executed in parallel as long as the dependences among them allow it but blocks are strictly executed sequentially between them. To define a complete DDM program all DDM blocks must be enclosed in a pair of `#pragma ddm startprogram` and `#pragma ddm endprogram` directives. Finally, before executing the preprocessor to create the DDM application, the user must also define the number of kernels that will be used in the execution using the `#pragma ddm kernel` directive.

Support for DDMVM systems

Table 2.6 shows the extensions developed in the DDMCPP [55] in order to support the $DDMVM_s$ and the $DDMVM_{FPGA}$ systems. In the DDMVM implementations the `kernel` parameter is a pair of numbers that defines the scheduling policy the programmer wants to use on the specific thread and a value that might be needed, depending on the scheduling policy used. The `kernel` statement in DDMVM is also used in a loop thread as to define the scheduling policy that the user wishes to apply on the execution of the loop iterations. The `arity` parameter describes the depth of the nested loops that are to be parallelized. Currently this is used on the declaration of a simple thread while the user removes the `for` loop statements from the code and only keeps the body of the loop. This parameter is used to expand the parallelism to the internal loops in the case of nested loops.

In contrast to TFlux implementations, in DDMVM the user declares the consumers of a thread (either simple or loop). This is done using the `update` statement on the `#pragma ddm endthread` and the `#pragma ddm endfor` directives. The `START` and `END` parameters are used to update multiple iterations of the consumer loop. The `import` and `export` directives have the same meaning as in TFlux, with the difference that in DDMVM you are also allowed to use expressions as to whether you will import or export a specific variable.

More details on the DDMVM$_s$ and DDMVM$_{FPGA}$ runtime systems and application examples can be found in [19,53].

2.4 Transactional Memory

Transactional Memory (TM) [56] is a model for manipulating mutable shared data which attempts to reduce complexity by eliminating the need for explicit synchronization. It works by allowing the programmer to specify that certain sections of a program must be executed atomically but without the need to consider any of the synchronized control that might be required. Execution of atomic sections takes place optimistically, that is with an assumption that any shared data within the section will not be changed by any other concurrent execution. If such a conflict does occur, the underlying runtime system ensures that only one execution succeeds while others are transparently re-executed. This leads to the important property of isolation. A thread always proceeds as though it has exclusive access to any shared data within an atomic section. All synchronization complexity is removed and it is unnecessary to serialize accesses to achieve correct execution. Although, in practice, some serialization may occur due to the resolution of conflicts, the optimistic nature of the model ensures that maximal parallelism is achieved. Although TM was originally proposed to reduce the complexity in the context of conventional threaded programming languages, the isolation property makes it an ideal way of introducing mutable state into Data-flow or functional approaches.

A common example used for motivating TM illustrates the need for mutable shared state [57]. Consider a computation which is trying to perform concurrent credit/debits between bank accounts. Firstly, the state is fundamental to the problem. The account balances must be globally accessible variables which can be updated and persist. The credit/debit operations must be atomic to preserve the overall correctness of the balances. Assuming that we do not know the identity of the accounts when specifying the problem, it is clear that the accounts might overlap and conflicts could occur. A conventional locking approach would need to deal with the cases where overlap might occur by taking explicit locks and dealing with complex interactions such as deadlock. The problem could, of course, be greatly simplified by serializing all the operations but this would defeat the desire to exploit parallelism. However, in many cases, there will be no overlap and an optimistic approach can proceed with maximal parallelism. We can envisage a Data-flow solution where threads have been generated to perform calculations on each account using a purely functional approach and then invoking a transaction to perform a balance transfer. Any number of such threads can be generated to operate in parallel without any need to consider how they interact.

2.5 Many-core Hardware

The inclusion of multiple cores in the same chip has become the de-facto standard for the processor architecture. This multi-core approach has resulted as a solution to the power- and complexity-walls of previous monolithic single core processors. The need to optimize performance per watt combined with the continuous advances in technology results in an increase of the need for more cores in processor chips [15]. To retain the power-performance efficiency to an acceptable level we are currently exploring large-scale parallel processing as the way to scale performance. Consequently, in order to exploit the software parallelism and optimize the performance per watt, the trend today is to include more cores in a single die resulting in what is known as many-core processors [15].

Today we observe two major trends of hardware designs for general purpose many-core processors: cache-coherent, shared-memory and distributed-memory, clustered architectures. Each architecture has different characteristics that target different application domains. There are also various parallel programming systems today that are used to develop parallel applications, both shared- and distributed-memory. The latter has proven to be a good solution for large clustered systems but memory-demanding process-based execution, non-trivial message-passing programming style and high cost for fine-grain parallelism prohibit their use on single-chip many-core processors. Shared-memory programming systems achieve good performance results on multi-cores but little evaluation exists on a many-core processor. In addition, most software systems require cache-coherence to work correctly but some works [7, 8] show that scaling hardware coherence on future many-core processors may be an issue for performance and network traffic. It also requires a significant amount of chip area to implement and will increase the complexity of the hardware to levels that the time for validation is increased substantially [58].

The industry is moving closer to the development of many-core processors with Intel proposing recently the Many Integrated Cores (MIC) Architecture [4] and introducing the 48-core SCC processor [59] as an experimental processor that adopts the clustered architecture with simple hardware design. Intel SCC avoids the hardware-based cache-coherence and introduces a software-oriented message passing based architecture instead. A software cache-coherence implementation for the SCC system can act as another potential solution for creating simpler many-core architectures, free of complex hardware. As X. Zhou et al. propose in [6] Software Managed Cache Coherence (SMCC) shows a comparable performance to hardware coherence while offering the possibility of having dynamically reconfigurable coherence domains on the chip. The unnecessary complex hardware support for applications with little sharing and the inability to support heterogeneous platforms make the SMCC achieve better use of silicon with significant reduction of hardware budget.

Totoni et al. in [60] use CHARM++ and MPI message passing paradigms to implement parallel applications with different characteristics in order to evaluate the Intel SCC processor in matters of performance. They get speedup results up to 32.7x for 48 cores and they propose more sophisticated cores for the future many-cores in order to increase the performance, mainly for the sequential execution. RCKMPI [61] by Intel and SCC-MPICH [62] by RWTH Aachen University implement customized MPI libraries aiming to improve the message passing model with respect to the SCC many-core architecture. These two implementations use an efficient mix of Message

Passing Buffer (MPB) and DDR3 shared memory for low-level communication in order to achieve higher bandwidth and lower latency. In [63] the authors examine various performance aspects of the SCC using a stream benchmark and the NAS Parallel Benchmarks BT and LU [64]. Their findings show that for these benchmarks the data exchange based on message passing is faster than shared memory data exchange and in order to improve the memory access behavior you must increase both the clock frequency of the mesh network and the memory controllers.

Xeon Phi [65] is a brand name given to a series of many-core processors designed, manufactured, marketed, and sold by Intel, targeted at supercomputing, enterprise, and high-end workstation markets. It incorporates more than 60 cores that can work both as stand-alone processing unit as well as a coprocessor assisting the execution of a multi-core host. The Xeon Phi architecture offers a fully cache-coherent environment across all cores. For this reason it supports standard programming models like OpenMP [46], POSIX threads [66] or MPI [67] without the immediate need to rewrite an application. It provides high performance in a low power envelope, by utilizing four-way simultaneous multi-threading cores with its 512-wide vector units. It targets high performance computing centers with some already starting to develop systems with multiple Xeon Phi units [68].

In addition, other types of many-core architectures exist like the Graphics Processing Units (GPUs) [3] that have been around for some time now and support parallel execution with hundreds of cores. GPUs offer large computing power with low cost but the programming of such engines is not trivial and the programmer must have sufficient programming skills and all the information of the underlying hardware in order to achieve good performance. GPUs are special purpose hardware units and the range of applications that offer significant performance increase is limited to data parallel applications.

Several projects currently, target the exploitation of parallelism for many-core architectures. In [69] three different programming strategies are used to test the scalability of a many-core clustered architecture that supports both shared- and distributed-memory (the 48-core Intel Single-chip Cloud Computer (SCC) [15, 59]). Low-Density Parity-Check (LDPC) error correcting codes [70, 71] were chosen as the baseline applications of this research because LDPC decoding is computationally demanding and requires irregular access to memory which make it a suitable candidate applications for many-core processors. LDPC codes are used today in communication standards such as DVB-S2 and WiMAX to transmit data inside noisy channels with high error probability. LDPC decoding is performed on the Intel SCC using three strategies: the Distributed Parallel Decoder, the Shared Parallel Decoder and the Parallel Multi-codeword Decoder. Results show that the distributed memory model can't scale due to the large number of messages exchanged by the parallel kernels, while the shared memory model provides limited scalability due to the overhead added by the uncacheable shared memory. On the other hand, the multi-codeword implementation scales almost linearly achieving a relative throughput of 28 for 32 cores [69].

Another important challenge of many-core processors is memory contention and serving the large amount of cores fast enough with data, so that performance can be increased without delays. In another work ([72]) we stressed the Intel SCC with three queries from the standard DSS benchmark TPC-H [73]. To reduce memory overheads we use a prefetching technique to bring data close the cores before they are needed. The simplicity of the Intel SCC architecture and the absence of hardware mechanism for cache-coherence allowed for adding more on-chip memory. This extra on-chip memory is user-accessible and in this work was used to store data close to the cores. Results show that depending on the complexity of the query, the performance improvement of such scheme can reach up to 5× [72].

2.6 Many-core Software

Several software solutions exist today that target the wide exploitation of parallelism on commodity general-purpose many-cores. Many of them show a renewed interest in the Data-flow computation approach that was pioneered in the late 1970s and 1980s [27–31, 74]. These past projects demonstrated that it was feasible to express large amounts of parallelism but a significant problem was how to throttle and schedule it efficiently. Subsequent projects show that coarsening the granularity of parallelism (from instructions to tasks) results in efficient execution. Many hardware solutions have been proposed as well but the focus of this section is on software systems that use unmodified existing hardware and can be directly compared to what is proposed in this thesis.

Swan [75] implements a unified scheduler on top of the Cilk [76] language to support recursive and Task Data-flow with more than one levels of parallelism. Like the previous systems, it features dynamic scheduling and assignment of tasks and implements shared *tickets* for resolving task dependences. To avoid simultaneous access on shared *tickets* the runtime requires atomic access and locking protection that can become a bottleneck when scaling to larger systems. SWARM [24] is another Data-flow system implemented for both shared and distributed memory systems. Its runtime system is built in such a way that no hardware cache-coherence mechanism is needed and it uses a dynamic scheduling policy to assign tasks to cores.

Qthreads [77] was developed to provide a portable abstraction for lightweight thread control and synchronization primitives. Although Qthreads is not a Data-flow implementation, it has many similarities. Qthreads is based on the full/empty bits (FEBs) technique developed by the Denelcor HEP [78] and used by the Cray XMT [79] and PIM [80–82], designs. This technique marks each word in memory with a *full* or *empty* state, allows programs to wait for either state, and makes the state change atomically with the word's contents. Although this technique can be emulated in software, it is clearly stated by the authors in [77] that without hardware support for lightweight synchronization, FEB locks will be a bottleneck.

The Legion [83] runtime system attempts to express locality and independence of program data and tasks using logical regions that name a set of objects (data). The programmer is responsible for explicitly grouping these objects to regions and this requires significant source code modifications. Similarly to OmpSs [22], Legion also uses a dynamic detection and enforcement of dependences on parallel tasks that increases the work of the runtime system and potentially increases overheads.

The Open Community Runtime (OCR) [84] is a recent effort that aims to provide a runtime system for extreme-scale computing. OCR proposed a shared globally unique name space without hardware cache coherence support and uses a dynamically generated task graph to express parallelism that could potentially produce significant runtime overheads (depending on the application and the hardware used). An implementation of OCR using the TBB task scheduler for the Intel Xeon Phi showed that it doesn't scale that well to such a high number of threads and significant optimizations are required in order to achieve performance comparable to OpenMP as shown in [85].

The Codelet [86] execution model aims to develop a methodology for exploiting parallelism in future exascale machines. A codelet is a collection of instructions that can be scheduled atomically as a unit of computation. Codelets are more fine grained than traditional threads and can be seen as a small chunk of work belonging to a larger task. The authors in [86] believe that the overhead of context switching (used in traditional task-based models) is too large, therefore system software and hardware will require significant optimizations in order to produce performance improvements.

OpenCL [87] represents a parallel programming standard especially for heterogeneous computing systems. SnuCL [88] is an OpenCL framework for heterogeneous CPU/GPU clusters that provides ease of programing for such systems. This framework achieved high performance on a cluster architecture with a designated, single host node and many compute nodes equipped with multi-core CPUs and multiple GPUs. The scalability though refers only to medium-scale clusters, since large-scale clusters may lead to performance degradation due to centralized task scheduling model followed. Lee et al. in [89] presents a new OpenCL framework, this time for homogeneous many-cores with no hardware cache-coherence, such as the Intel SCC. The framework includes a compiler and an OpenCL runtime which together with the dynamic memory mapping mechanism preserve coherence and consistency between CPU cores on the SCC architecture with a small overhead.

All these solutions are software implementations for shared-memory systems with each one providing a task-based runtime that dynamically schedules parallel tasks to processor cores based on data availability. Each system also comes with its own programming interface that requires extra effort from the programmer and all except for SWARM and OCR require support of hardware cache-coherence in order to provide correct execution. More specifically, the work described in this thesis differs from other solutions in the following ways: (1) it implements a distributed triggering system that avoids all synchronization primitives, such as locks and barriers, (2) it implements a *static* scheduler that minimizes runtime overheads, (3) it requires no hardware support for cache-coherence that provides for portability of the implementation in future processor that may not support hardware cache-coherence (also shown in a previous work [16]) and (4) it maintains programming productivity by extending a widely used programming API (OpenMP v4.5 [90]).

Chapter 3

Speculative Parallelism in Data-flow

Data-flow is generally associated with functional styles of programming which do not handle well shared mutable state. There have been a number of attempts to introduce state into Data-flow models and functional languages but none have proved able to maintain the simplicity and efficiency of pure Data-flow parallelism. Transactional Memory is a concurrency control mechanism that simplifies sharing data when developing parallel applications while at the same time promises to deliver affordable performance by exploiting undiscovered parallelism. This work reports the experience of integrating Transactional Memory and Data-flow within the TFlux Platform. The ability of the Data-flow model to expose large amounts of parallelism is maintained while Transactional Memory provides simplified sharing of mutable data in those circumstances where it is important to the expression of the program. The isolation property of transactions ensures that the exploitation of Data-flow parallelism is not compromised but instead is enhanced with speculatively executing tasks in parallel that would otherwise be serialized. This work extends the TFlux platform, a Data-Driven Multi-threading implementation, to support transactions (DDM+TM), in a way to extend the application coverage of the Data-flow model. To achieve this, new pragmas are proposed that allow the programmer to specify transactions. In addition the runtime functionality is extended by integrating a software transactional memory library with TFlux. To test DDM+TM, two applications are ported that require transactional memory: Random Counter and Labyrinth an implementation of Lee's parallel routing algorithm. Results show good opportunities for performance scalability when using the integration of the two models.

3.1 Motivation

As technology delivers higher integration of devices into processors, the multi-core design has become the de-facto standard for processor architecture. It promises to deliver high performance whilst maintaining an acceptable complexity and power budget. The trends show a continuous increase in the number of cores and it is expected that by 2020 processors will include 1000s of cores [25]. This will lead to new challenges, one of them being the programmability of such large-scale systems. If their power is to be harnessed on the solution of a wide range of problems it will be necessary to develop new parallel programming models which are both efficient and easy to use.

There has recently been a resurgence of interest in the Data-flow model as a way to efficiently exploit large-scale parallelism. Even though the original implementations of Data-flow were not efficient, more recent developments have overcome this [50,91, 92]. However, the models are suited largely to the implementation of programming styles which are essentially purely functional. Indeed it is the absence of side effects in functional models which permits easy parallelization. Unfortunately, there are many cases in real programs where the use of shared mutable state is either necessary for efficiency or is a fundamental part of the problem being solved. In these circumstances, functional approaches are unsuitable.

This limitation has long been recognized and there have been a number of attempts to integrate state into both Data-flow models and functional languages. The early work on M-structures in Id [93] added implicit locking to data items to avoid the explicit manipulation of synchronization. However, this merely hid the complexity of the synchronization rather than removed it and, in many ways, made the writing of shared state programs more error prone. Functional languages such as SML [94] and F# [95] have introduced mutable variables in an attempt to extend their practicality. Unfortunately this state can rapidly destroy both the mathematical cleanliness of the language and the ability to exploit parallelism with Data-flow like execution models. Haskell originally introduced state in a more disciplined way by the use of MONADS [96]. However, although this enables the isolation of state via the type system and hence preserves mathematical properties, the state manipulation is serialized and thus does not address the problems of writing parallel programs.

The Transactional Memory (TM) model facilitates sharing data in a manner which isolates individual sharers from the complexities of synchronization. It was originally proposed as a way of simplifying parallel programming in conventional languages but has been shown to provide a clean and simple way to add the sharing of state to a functional language [57]. Transactional Haskell uses the MONAD approach to allow the expression of explicit threads within Haskell programs by defining transactional variables which are manipulated serially within a thread but interact in parallel across threads. It has been shown that parallel state based programs can be specified while maintaining much of the purity of functional programming. It is our belief that a more general and more usable programming model can be produced by adding transactions to Data-flow using more pragmatic programming approaches and avoiding the complexity of MONADS.

This work reports the first experience of integrating transactions into a thread based Data-flow model. The TFlux [18] platform is used as the Data Driven Multi-threading (DDM) implementation and TinySTM [97] for transactional support. This merged implementation will be referred to as DDM+TM [10,11], hereafter. Through this work additional TFlux directives are proposed for defining transactional threads and variables. The possibility of programing applications with the combined model and present a preliminary performance evaluation study. The overheads of the proposed model are also analyzed through an evaluation process using a synthetic application.

3.2 Data-flow and Transactional Memory

This Section describes the two individual parallel programming models, emphasizing on the strengths and weaknesses of each one separately. The combination of the models as proposed in this work is also discussed.

3.2.1 Data-flow

Data-flow is known to be the model that is able to exploit the most parallelism in an application. In the recent years it has been revisited as the solution for scaling the performance of applications. Data-flow is a computation model that does not follow the classical program counter model but instead relies the execution of each

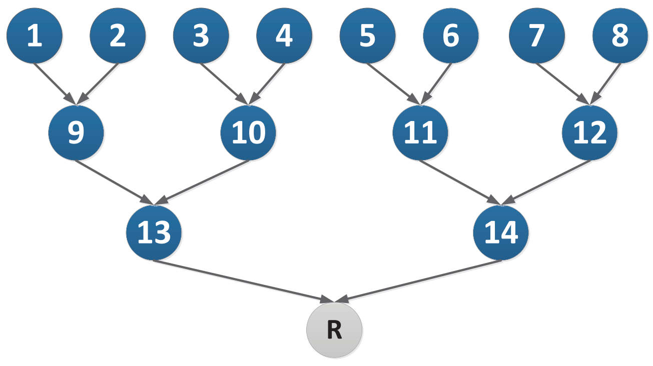


Figure 3.1: A Data-flow graph for calculating a reduction operation.

operation on the availability of its input data. Each operation can be considered as a Data-flow node and each node can be executed independently of all other nodes.

A Data-flow program can be described using a directed acyclic graph (Data-flow Graph), where each node represents an operation and every edge the data dependences between nodes. Figure 3.1 shows a Data-flow graph for a reduction operation. The first level of nodes can start execution immediately as there are no input dependences, while the rest of the nodes must wait for their input to arrive from previous nodes as depicted by the directed edges of the graph.

Strengths: The Data-flow model has the ability to exploit the maximum available parallelism in an application while avoiding the high synchronization overheads and memory latencies of other parallel implementation (e.g. locks, barriers). Data-flow is a side-effect free model as each node depends only upon its inputs and can be independently executed. A Data-flow program can be easily implemented as a distributed program as the communication between nodes is explicit and done only at the end of each nodes execution.

Weaknesses: The Data-flow model lacks the ability of providing shared state in programs where this is a fundamental operation. To overcome this limitation it is important to be able to provide shared mutable state in a Data-flow program. However, it is also essential that this is done without introducing explicit synchronization in the program as this will lead to unnecessary serialization of the execution that will eventually eliminate the Data-flow principles from the execution.

3.2.2 Transactional Memory

Originally proposed as a way of simplifying parallel programming in conventional languages, Transactional Memory [56] attempts to reduce complexity of manipulating mutable shared data by eliminating explicit synchronization. It allows the programmer to specify that certain sections of a program will be executed atomically, without the need of considering any synchronization control. Atomic sections are executed fully parallel and if a conflict occurs on shared data, the underlying runtime will ensure that only one transaction succeeds, while others are re-scheduled for execution. Using Transactional Memory, parallel state programs can be specified while maintaining much of the purity of functional programming.

Strengths: The atomicity offered by TM will allow concurrent access to shared data structures, consequently allowing parallel execution of transactions that share these data. By monitoring these data structures the TM runtime system will detect any conflicts that may arise during the execution. In such a case the conflict transactions will abort and the runtime system will reschedule them.

Weaknesses: The overheads imposed by existing software implementations of TM are significant. These overheads come from monitoring data structures, aborting and rescheduling conflicting transactions. Although TM is good at synchronizing and monitoring memory access to shared data, it does not offer any mechanism to prevent conflicts. These conflicts alone impose a large overhead as the runtime system will need to reschedule them, thus re-executing them all over again. Some of the overheads of a software TM implementation can be resolved when using a hardware TM system. The focus in this work is on a software implementation as to keep it portable, as this is a first work of combining the two models and will provide us with a platform for debugging and experimenting with the system as much as possible.

3.2.3 Data-flow and Transactional Memory Combined

Our proposed combined model allows both approaches to be integrated in one system. Data-flow is used to parallelize a program at the highest degree possible, thus retaining the Data-flow principles in a parallel application. Where shared state is needed, or to explore even more parallelism in the parallel application, the TM system is introduced.

Strengths: Combining the two models results in several benefits. The strengths of both models are retained and some of their weaknesses are minimized. Introducing mutable state improves the Data-flow model by allowing data structures to be easily shared among Data-flow threads, without having to explicitly merge them as you would in a purely Data-flow program. The TM model also improves by reducing the number of conflicts. By using both Data-flow threads and transactions in the same program the total number of transactions in a parallel program is reduced. Eventually this means that the possibility of transactions to conflict will also be reduced. Overall, this will reduce the overhead produced by a software TM implementation. The Data-flow model cannot explore parallelism in sections that share common data structures. Integrating transactions into the Data-flow model adds the ability of handling shared state, thus increasing the level of parallelism that can be explored. Data-flow threads are stateless and can be re-executed since they have no side effects. This reduces the work that has to be done by a TM system upon recovery in a conflict situation, by simply re-executing the conflicting Data-flow threads.

Weaknesses: To provide mutable shared state on data structures for the Data-flow model the shared data must be monitored from the TM runtime system. This will provide atomicity, isolation and consistency for the shared data and the execution itself. To do so, the data structures that are to be monitored must be specified. In the proposed model this task is left to the programmer. To ease the programmers' effort though a more efficient way was implemented to express these dependences on the shared data structures between threads as described in Section 3.3.

3.3 Proposed DDM+TM Implementation

3.3.1 TFlux System

This study uses the Data-Driven Multi-threading model and in particular its TFlux implementation [18]. An important advantage of TFlux is that it is not build for a specific machine but rather works as a virtualization platform for DDM program execution on a variety of computing systems. Also, TFlux parallel source code is in ANSI-C which complies with the supported programming languages of most systems. More details on the TFlux Platform can be found in Chapter 2, Section 2.3.

3.3.2 TinySTM Software Library

In order to support the transactional execution, *i.e.* the monitoring of the updates to variables, the conflict detection and the restarting of the execution in case of abort, the TFlux runtime must be extended. Rather than developing from scratch the TFlux runtime system was extended with an existing software TM implementation. For this purpose, TinySTM [97] was chosen as it appeared to provide a simple approach to the integration. However, other software TM systems could be used such as TL2 [98] or RSTM [99].

3.3.3 DDM+TM Implementation

A program in DDM+TM consists of both Data-flow threads and transactions. Each of the two models though has it's own runtime system for scheduling either threads or transactions, thus when combining the two responsibilities must be set for each runtime. For this work, the two runtime systems are kept independent from one another but a hierarchical structure is enforced for the execution of Data-flow threads and transactions. The TFlux runtime system (the TSU) runs on top of the TM runtime library and is responsible only for scheduling Data-flow threads, while the TM runtime is only responsible for aborting/committing transactions. The scheduling of a Data-flow thread may impose the start of a transaction but the TSU will not interfere with the monitoring of the transactions. When a transaction starts execution the TM runtime will take over and monitor for conflicts. If such a conflict occurs, the TM runtime will reschedule the transaction without the TSU noticing any changes. As soon as a transaction commits and the Data-flow thread is finished, the TSU will take over again and schedule the next ready thread.

In this first attempt at adding TM to TFlux, no changes were made to the TSU in order to support transactional behavior. However, the possibility of offloading the re-scheduling of an aborted transactional thread to the TSU instead of the TinySTM system is investigated, as previous work on TM [100,101] has shown that controlling the scheduling using information about transactions can improve the performance and reduce wasted work due to aborts.

When adding support for transactions to TFlux an important decision concerned the granularity of transactions. The simplest approach would be to declare a whole thread as a transaction. With this option the system is enhanced by providing the

Table 3.1: TFlux DDM+TM pragma directives.

DDM+TM Pragma Directives	Description

`#pragma ddm atomic thread` *ID* `tvar(`*NAME : READ/WRITE/READ_WRITE*`)`
`#pragma ddm atomic endthread`
	DDM+TM thread boundaries with identifier *ID* and the atomic variables to monitor for either *READ* or *WRITE*

`#pragma ddm atomic for thread` *ID* `tvar(`*NAME : READ/WRITE/READ_WRITE*`)`
`#pragma ddm atomic endfor`
	DDM+TM loop thread boundaries with identifier *ID* and the atomic variables to monitor for either *READ* or *WRITE*

`#pragma ddm atomic transaction tvar(`*NAME : READ/WRITE/READ_WRITE*`)`
`#pragma ddm atomic endtransaction`
	DDM+TM boundaries of a transaction that is smaller than a thread and the atomic variables to monitor for either *READ* or *WRITE*

`#pragma ddm atomic tvar(`*NAME : READ/WRITE/READ_WRITE*`)`	Declare an atomic variable to monitor either for *READ* or *WRITE*
`#pragma ddm atomic abort`	Manually abort a transaction

programmer with two types of threads: pure Data-flow threads or transactional threads. However, a thread may contain code that needs to be transactional but combined with non-transactional code. Furthermore, it may be appropriate to specify several atomic regions within a thread. This could lead to potentially wasteful aborts when either a transaction is only a small portion of the thread or multiple atomic regions need to be aborted together. Therefore, support for defining the beginning and the end of transactional sections within threads is provided.

The responsibility of finding the dependences in a Data-flow program and the variables to be monitored in a TM program fall upon the programmer. In DDM+TM the programmer must decide for both the dependences of Data-flow threads and the variables to be monitored within transactions. To ease the effort of the programmer in developing DDM+TM programs new pragma directives are introduced that allow the declaration of Data-flow threads along with their dependences and the declaration of transactions with the variables to be monitored. These new TFlux directives are presented in Table 3.1.

Another design issue is how transactional variables are identified. These variables will require that their read and write operations are observed to form the read-set and write-set during a speculative execution of the transaction. These sets are used to detect conflicts. For all these transactional variables the results also need to be versioned to allow a clean restart of the transaction if necessary. One option is to monitor every memory access that is performed within a transaction. However, this is not necessary for unshared variables, for example those which are thread local. Therefore, explicit declaration of transactional variables is used, as in other TM approaches. The directive

```
#pragma ddm atomic tvar(NAME : READ/WRITE)
```

offers such functionality. Note that each transactional variable is associated within

Table 3.2: TFlux DDM+TM pragma directives correspondence with TinySTM runtime calls.

DDM+TM Pragma Directives	TinySTM runtime support
`#pragma ddm atomic thread ID tvar(NAME : READ/WRITE/READ_WRITE)`	`stm_init_thread()`
`#pragma ddm atomic endthread`	`stm_exit_thread()`
`#pragma ddm atomic for thread ID tvar(NAME : READ/WRITE/READ_WRITE)`	`stm_init_thread()`
`#pragma ddm atomic endfor`	`stm_exit_thread()`
`#pragma ddm atomic transaction tvar(NAME : READ/WRITE/READ_WRITE)`	`sigjmp_buf *_e = stm_start(&_a)      if (_e != NULL) sigsetjmp(*_e, 0)`
`#pragma ddm atomic endtransaction`	`stm_commit()`
`#pragma ddm atomic tvar(NAME : READ/WRITE/READ_WRITE)`	`int temp = (int) stm_load((stm_word_t *)&NAME) OR` `stm_store((stm_word_t *)&NAME, (stm_word_t *)temp)`
`#pragma ddm atomic abort`	`stm_abort()`

a thread with a `READ`, `WRITE` or `READ_WRITE` qualifier. This qualifier provides information on the use of the variable within the thread which can be used by the TM implementation to optimize the execution.

For DDM+TM, there is a complete separation between transactional and non-transactional variables. Transactional variables must always be accessed within a transaction. Non-transactional variables are normally private to a thread during execution and thus cannot generate conflicts. Other non-transactional threads will only be allowed to access a non-transactional variable if the scheduling can guarantee independence. With this decision weak isolation problems are avoided. Note that by imposing this decision the DDM is not modified in any way. DDM+TM could be implemented without speculation by performing a scheduling where the transactional variables are treated as inputs and outputs of the threads that are read and written. This will result in the sequential execution of the code in case of threads accessing shared data where the dependences can't be determined prior to the execution.

Pragmas that define transactional variables within the declaration of a transactional thread are also provided. This is required to support the monitoring of variables that may have more than one alias (e.g. parameter variables inside the code of a function). The monitoring of these variables is specified as a parameter in the thread declaration (see Table 3.1).

As TFlux has two pragmas for declaring threads, the table contains

`#pragma ddm atomic thread ID` and

`#pragma ddm atomic for thread ID`

declaring a transactional thread and a transactional loop thread, respectively. The `tvar(NAME : READ/WRITE)` extension defines the thread variables that are transactional.

The last proposed directive

```
#pragma ddm atomic for thread 1 tvar(counters : READ_WRITE)
   for(cv00 = 0; cv00 < THREADS; cv00++)
   {
      for (j = 0; j < numOfCounters; j++)
         indexTM[j] = rand() % arraySize;
         for (j = 0; j < numOfCounters; j++)
            counters[indexTM[j]]++;
   }
#pragma ddm atomic endfor
```

Figure 3.2: Random Counters implementation with TFlux DDM+TM pragma directives.

`#pragma ddm atomic transaction`

allows the declaration of a transaction as a portion of a thread. For certain applications such as those considered in this work, this offers better performance (see Section 3.6).

These directives are a subset of the possible ones which have been defined for TM [56]. However, they are enough to implement the applications described in this work and considered as the core directives. Extra transactional functionality can be added by declaring

`#pragma ddm atomic abort`

in the case the programmer wants to manually abort a transaction. Table 3.2 shows the correspondence of the proposed pragmas with the calls to the TinySTM runtime that will automatically be generated by the preprocessor.

3.4 Workloads

For this proposal of transaction integration with the Data-flow model two applications were tested. The Random Counters and the Labyrinth implementation of Lee's algorithm from the STAMP Benchmark suite [102]. These were originally implemented in a conventional language using TM and are not naturally Data-flow applications. However, for this study the focus is mainly on the functionalities and overheads of the implementation rather than the added benefits of the integration such as exploiting implicit parallelism.

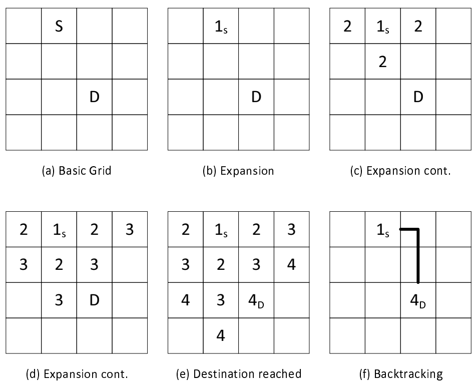


Figure 3.3: Lee's algorithm example.

3.4.1 Random Counters

In the Random Counters application multiple threads are used to increment the values of random array positions. All elements of the array are created and initialized to zero. Multiple threads are then spawned to execute loop thread code. Each loop iteration is executed in parallel and each thread will do the same work: generate a random sequence of numbers - that represent index positions in the array and increment by 1 the values located in those positions. Figure 3.2 shows the code of the implementation of this algorithm, where the

`#pragma ddm atomic for` and

`#pragma ddm atomic endfor`

directives define the DDM+TM thread boundaries, showing that all the code of the thread is considered to be transactional. The purpose of this algorithm is to create concurrent accesses to memory. This will in turn create memory conflicts between threads trying to access the same memory locations. In an unsynchronized parallel implementation this execution would create a false result.

In the DDM model this code would have to be executed sequentially since the dependences cannot be defined prior to the execution (due to the random memory accesses). By using TM the values of the shared variables are protected, therefore correct parallel execution is ensured with a correct output.

3.4.2 Labyrinth implementation of Lee's Algorithm

Lee's Algorithm is used in the process of producing an automated interconnection of electronic components. It guarantees to find a shortest interconnection between two points using the Expansion-Backtracking technique shown in Figure 3.3.

```
#pragma ddm atomic for thread 1
    for(cv00 = 0; cv00 < THREADS; cv00++)
    {
        #pragma ddm atomic transaction tvar(queue : READ_WRITE)
            [1] Get a (S, D) pair from the work queue
        #pragma ddm atomic endtransaction

        #pragma ddm atomic transaction tvar(global_grid : READ)
            [2] Copy global grid to threads local grids
        #pragma ddm atomic endtransaction

        [3] Expansion stage
        [4] Backtracking stage

        #pragma ddm atomic transaction tvar(global_grid : WRITE)
            [5] Write selected route back to global grid
        #pragma ddm atomic endtransaction
    }
#pragma ddm atomic endfor
```

Figure 3.4: Lee's Algorithm pseudo-code with TFlux DDM+TM pragma directives.

Starting from the source point *S*, the grid points are numbered by expanding a wavefront until the destination is reached (Figure 3.3(a)-(e)). At each phase during the expansion stage each grid point in the wavefront marks its unnumbered neighbors with an increment of its value. Once the destination *D* is reached, a route is traced back to the source by following any decreasing sequence of numbered grid points that are not used by others. Once the shortest route has been determined, the grid points reserved for this route cannot be used by others [103].

For the purpose of this work, the Labyrinth TM implementation of Lee's algorithm from STAMP [102] was used as a guideline and ported it to DDM+TM. Figure 3.4 presents the integration of TFlux DDM+TM pragmas into the application. The

`#pragma ddm atomic for` and

`#pragma ddm atomic endfor`

directives are used to declare the boundaries of a DDM+TM thread. In step 1 each thread will take a *(S, D)* pair from the global work queue as an atomic action. In step 2 each thread will copy the global grid to its local memory space, and in steps 3-4 each thread will execute the algorithm's expansion and backtracking phase locally. Finally in step 5 each thread will write the selected route back to the global grid as an

Table 3.3: *Experimental workloads problem sizes.*

Benchmark	**Problem size**		
Random Counters	10 updates	100 updates	1000 updates
Labyrinth	256x256x3-n256	256x256x5-n256	512x512x7-n512

atomic action. In order for steps 1, 2 and 5 to be executed correctly no data conflict must be ensured on the work queue or global grid. Declaring steps 1, 2 and 5 as transactional code with the

`#pragma ddm atomic transaction` and

`#pragma ddm atomic end transaction`

atomic execution is provided for those steps ensuring that the final result will be correct. As for steps 3 and 4 the execution is done locally using only data local to each thread. This does not require this part of the code to be transactional and lets it execute in parallel within the boundaries of the thread, thus avoiding the need to re-execute non-transactional code in the event of a conflict. This is therefore an example where not all the code of a thread is transactional and hence need not be declared with the transactional version of the pragma.

3.5 Experimental Setup

The integration of TM with the Data-flow model is evaluated using the Random Counters and Lee's algorithm described in Section 3.4. A conventional multi-threaded TM implementation is used as a baseline for our implementations of Random Counters for the DDM+TM version of the application. In this preliminary work the overheads of the DDM+TM model over the TM implementation is studied rather than detailed performance. For Lee's algorithm the Labyrinth implementation was used from [102] that uses TM calls from TinySTM and integrated the TFlux directives to create the DDM+TM version. The results are for the parallel DDM+TM implementation of the applications on a 12-core machine with 2 6-core AMD Opteron(tm) Processors 2427 running at 2200MHz. The available memory of the system is 31GB of main memory, 512KB of L2 and 6144KB of L3 cache. The system is running an Ubuntu SMP x86-64 operating system.

The porting of the applications for the Data-flow model was performed using the

Table 3.4: *Random Counter Commits/Aborts for TM and DDM+TM implementations.*

Number of Threads	Commits	Aborts	
		TM	DDM+TM
2	200	38	47
4	400	157	479
8	800	920	1283

pragma directives from [55] of the TFlux Soft system [18] version 1.5.0. To integrate the transactions in the applications the TinySTM library was used. The execution time measurements were collected using the *gettimeofday* system call to measure the execution time of the section of code that has been parallelized. The input data sizes used for each application are depicted in Table 3.3. All the results collected are for the Data-flow model implementing transactions over the baseline execution. The baseline execution was considered to be the sequential execution of the applications implementing transactions with the TinySTM library. To avoid any statistical errors the average of 10 executions was used to produce the results, removing the largest and the smallest execution time.

3.6 Experimental Results

3.6.1 DDM+TM Applications Evaluation

Table 3.4 shows the number of commits for executions with a different number of threads and the average number of aborts for 10 executions for the Random Counter application both for TM and DDM+TM implementations. For each execution the results are different due to the use of *random* in the application. Since each execution of the application is random the commits and aborts will follow a uniform random distribution. Consequently, the results demonstrate that the integration of TM with DDM behaves similarly to a normal TM implementation but cannot be used to extract conclusions about detailed performance.

Figure 3.5 shows the difference in the execution time of the Labyrinth benchmark when reducing the size of the transactional code inside a thread. From now on the execution of a whole transactional thread will be referenced as *large* and the execution

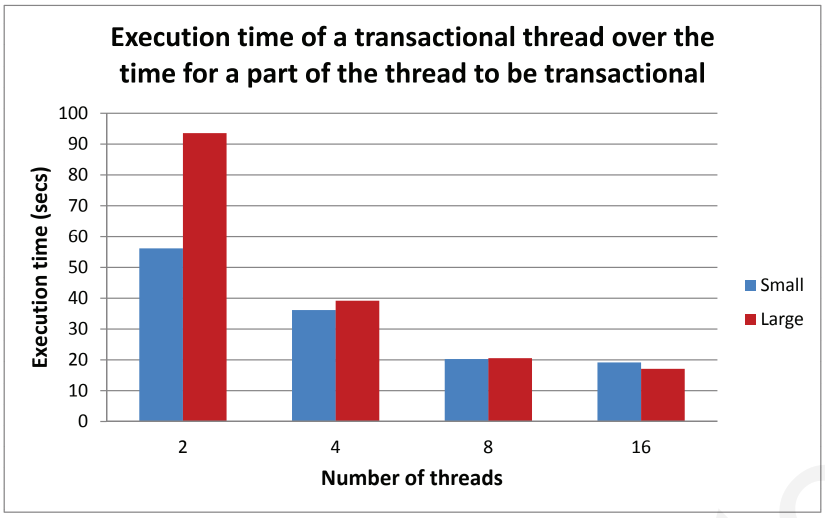


Figure 3.5: Complete vs partially transactional threads in the Labyrinth implementation.

of a part of the thread as transactional as *small*. On average the execution time is smaller when only a part of the thread is declared as transactional. As the number of threads increases, this difference is decreasing. The correlating of these results with the numerical results of the previous executions from Table 3.5, shows that when the number of aborts is the same for the two executions the *small* implementation is faster and when the number of aborts for the *small* implementation exceeds the number of aborts of the *large* implementation the execution times are almost the same. This correlation concludes that the overhead of rescheduling a whole transactional thread is higher than rescheduling a small part of the thread. This is because the rescheduled code to be executed will be more in the former case.

Table 3.5: Labyrinth Commits/Aborts for TM and DDM+TM implementations.

Number of Threads	**Commits**	**Aborts**	
		Small	**Large**
2	1028	18	18
4	1032	47	46
8	1040	101	99
16	1056	182	175

Finally, the experimental results that indicate the speedup for the DDM+TM implementation of the Labyrinth application over the TM sequential implementation are depicted in Figure 3.6. From these results it is easy to observe that for the two smaller input files (256x256x3-n256 and 256x256x5-n256) the application scales well

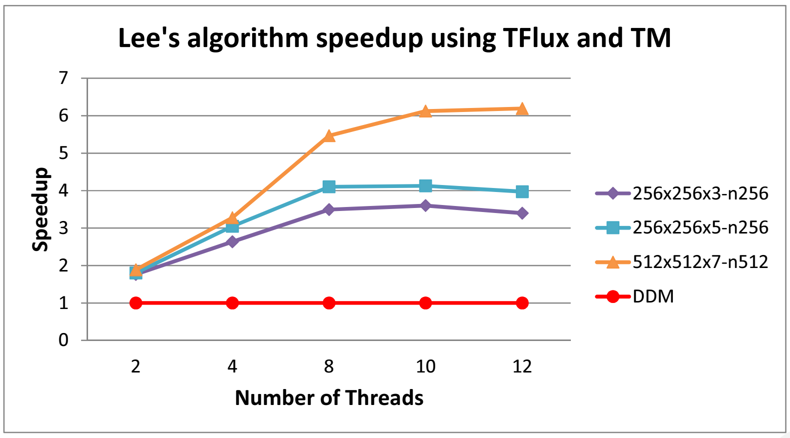


Figure 3.6: Labyrinth's TM routing algorithm - scalability of TFlux extended with TM.

up to 10 threads and then starts degrading. For the largest input file it scales beyond 10 threads with a maximum speedup of 6.2x over the sequential implementation with TM at 12 threads. Although these results are on a 12-core machine, the reason for this degradation in the speedup for more than 10 threads is that the TSU and the OS is running simultaneously with the application and occupying one core each [18]. As seen from Figure 3.6 the scalability of the DDM+TM model is good.

3.6.2 DDM+TM Overheads Analysis

To get a clear view of the integration of transactions into the Data-flow model, simple scenarios were created to test the TM runtime system and record the overheads produced by monitoring shared mutable data, as well as the overheads of aborting and rescheduling conflicting transactions. Synthetic applications were created with multiple threads that interact with shared data structures in order to record the overheads produced.

In the first scenario a parallel application was executed that uses shared data but will not create any conflicts during the execution. The data of this application are shared only for read operations, so no conflicts that require aborting of parallel transactions will arise. The purpose of this scenario is to record the overheads when monitoring only one such shared data structure, as opposed to monitoring all the shared data structures of the application. Monitoring a large number of shared data structures not only requires more hardware resources (e.g. memory) but, as shown in Figure 3.7, it also results in a large overhead in execution time. The more data

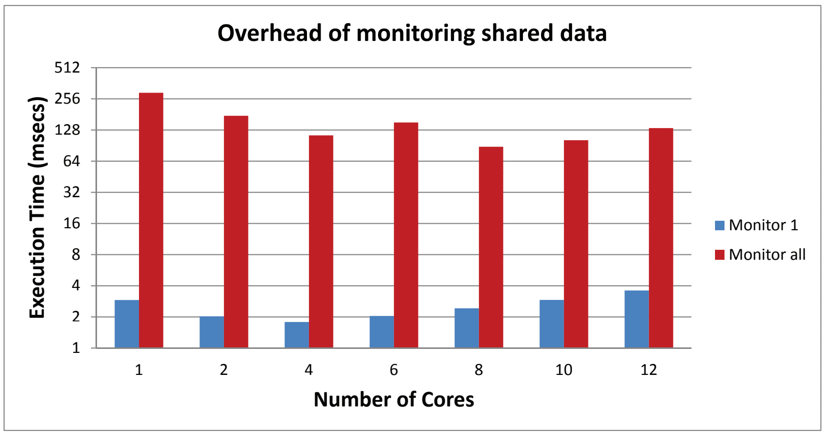


Figure 3.7: Overheads of monitoring one shared variable versus monitoring all shared variables of the parallel application.

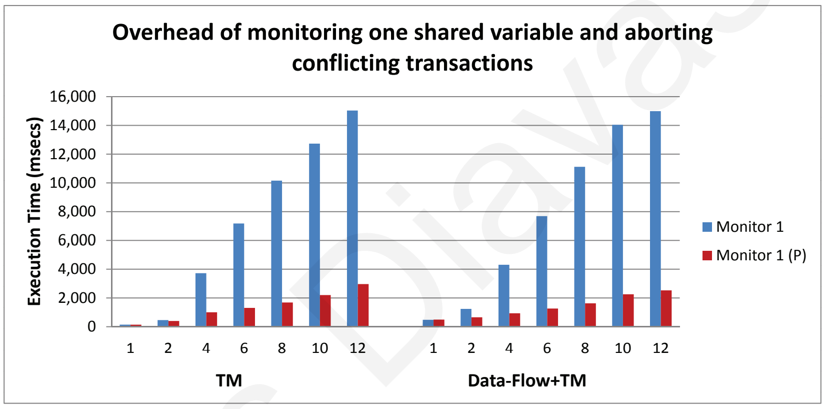


Figure 3.8: Overheads of aborting conflicting transactions while monitoring one shared variable when a whole Data-flow thread is declared as a transaction versus a part of a Data-flow thread to be declared as a transaction.

structures monitored, the more work a TM runtime system will have to do during the execution. Although there are no conflicts during the execution the runtime system will still consume resources (mainly CPU time) in order to monitor the execution and check for possible conflicting transactions.

In another scenario the granularity of a transaction relative to a Data-flow thread was studied. Figure 3.8 shows the overhead produced when monitoring one shared variable for two granularity levels. The first case considers that a whole Data-flow thread will be transaction. This means that in case of a conflict the whole Data-flow thread will be aborted and re-executed. The second case, declares as a transaction only the portion of the Data-flow thread that actually uses the shared data structure for a write operation. This is the only part of the Data-flow thread that a conflict may

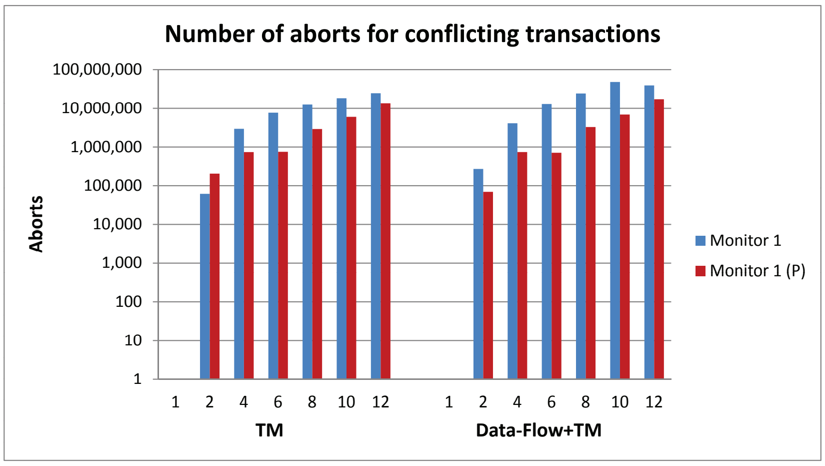


Figure 3.9: Average number of aborts of conflicting transactions while monitoring one shared variable when a whole Data-flow thread is declared as a transaction versus a part of a Data-flow thread to be declared as a transaction.

arise. In this case, if a conflict occurs during the execution only that portion of the thread will be aborted and rescheduled for execution, retaining any operations done previously in the Data-flow thread. This avoids re-executing any operations done previously and did not affect the shared data monitored in any way. The results in Figure 3.8 show significant reduction of the overhead imposed when rescheduling only the portion of the thread that uses the shared data, instead of rescheduling the whole Data-flow thread.

When the granularity of a transaction is reduced, the scope of a conflict detection is reduced by the TM runtime system. This means that as soon as an operation on shared data structures is executed the transaction will commit, allowing other transactions to execute correctly and without a conflict. The average number of aborts for these two test cases are presented in Figure 3.9. The results clearly state that when reducing the scope of a conflict detection (i.e. the granularity of a transaction) the number of conflicts and consequently the number of aborted transaction will reduce significantly.

In the previous two figures results were presented for both the TM implementation and the proposed implementation of combining the Data-flow model with transactions. The small variance in the results show an instability of the execution that is affected mainly by the TM runtime system. The nature of a TM implementation is that every execution is different from any other. The important thing that should be taken from this is that adding Data-flow to a software TM implementation does not add any significant extra overhead to the performance of the TM system used. This means that by combining the two models, their overheads are not necessarily combined to the new proposed model.

3.7 Conclusions and Contributions

This chapter reports the lessons learned from integrating, DDM with TM. DDM offers the benefit of avoiding sharing of mutable data while expressing large amounts of parallelism. However, it is not always possible to avoid sharing mutable state and that reduces parallelism in the DDM model. By adding TM to DDM (DDM+TM), further parallelism is exploited while mutable shared state is introduced in a composable way. To test DDM+TM, two applications are ported that require transactional memory: Random Counter and Labyrinth an implementation of Lee's parallel routing algorithm. Results show good opportunities for performance scalability when using the integration of the two models.

The experiments evaluated indicate that, from the runtime integration point of view, existing software implementations can be integrated without major problems. The biggest interaction and point for further exploration comes with the scheduling. When a complete DDM thread is a single transaction, the TSU could take ownership of the restart mechanism and reschedule the thread.

The major contributions of this work are:

- The first software integration of transactions into a thread based Data-flow implementation as a way to introduce shared state in the Data-flow model.
- The DDM runtime system of TFlux is extended to support transactions by integrating a software transactional memory library (DDM+TM [10, 11]);
- Analysis of the overheads created by the integration of the two models.

DDM+TM extends the TFlux directives to develop Data-flow applications with new pragmas for TM. The extended TFlux runtime system with functionality provided by a software TM was also described. The new runtime system was tested by developing two applications that require TM: Random Counter and an implementation of Lee's parallel routing algorithm.

Chapter 4

Data-Driven Multi-threading on Many-core Systems

The development of software that is able to exploit large amount of hardware resources is one of the biggest challenges the community faces. In this work a Data-flow based system is proposed that can be used to exploit the parallelism in large-scale many-core processors. The proposed system - TFluxSCC - is an extension of the TFlux DDM, which evolved to exploit the parallelism of the 48-core Intel SCC processor. With TFluxSCC scalable performance is achieved using a global address space without the need of hardware support for cache-coherence. The scalability study shows that application's performance can scale, with speedup results reaching up to 48x for 48 cores. The findings of this work provide insight towards what a Data-flow implementation requires and what not from a many-core architecture in order to scale the performance.

4.1 Motivation

Scaling the performance of an application can be achieved by either improving the hardware, or developing more efficient software to solve the particular problem. To retain the power-performance efficiency to an acceptable level we are currently exploring parallel processing as the way to scale performance. Consequently, the trend today is to include more and more cores into the processor resulting in what is known as a multi- and many-core processor. From the hardware design perspective, the more parallel units offered for execution, the higher the performance that can be achieved. From the software design perspective though, this new trend creates new challenges. To achieve scalable performance in these new systems, programmers are required to think parallel. This essentially means learning new programming models and developing new algorithms that exploit parallelism. Therefore, the solution to the scalability of the performance depends on scalable hardware with increasing number of computational units along with efficient programming and parallel execution models that hide the hardware complexity from the programmer.

The objective of this work is to show how a software implementation of the DDM model of execution can offer performance scalability on a many-core architecture by efficiently exploiting the parallelism and at the same time relieve the programmer from the hardware details, such as data communication. The complete software platform presented in this work is based on the TFlux platform [18]. It offers an environment for parallel execution on many-core architectures that is able to scale without major hardware requirements or programming effort. The experimental processor Intel SCC [104] was used to test the implementation. The SCC was developed by Intel for many-core software research, as a representative for future many-core processors. The proposed system, called TFluxSCC [16], includes a source-to-source preprocessor that takes as input programs in C, augmented with directives that specify the threads and their dependences, as well as, a runtime system to handle the scheduling of the threads in a Data-flow manner. The evaluation is performed on a real Intel SCC system and the model is implemented in software as a library that is linked to the application.

4.2 Intel SCC Architecture

The need to optimize performance per watt has resulted in the increase of the number of cores in processor chips [59]. Many light-weight cores will be replacing the few sophisticated cores we have currently in multi-core chips, creating the many-core chips with hundreds or more cores. This work examines the Intel SCC experimental processor as a 48-core 'concept vehicle' created by Intel Labs as a platform for many-core software research [104].

The Intel SCC processor consists of 24 dual-core tiles interconnected by a 2D-grid network as illustrated in Figure 4.1. These tiles are organized in a 6x4 mesh with each one containing two cores with dedicated L1 and L2 caches of 16KB and 256KB respectively, 16KB Message Passing Buffer (MPB) for message storing travelling

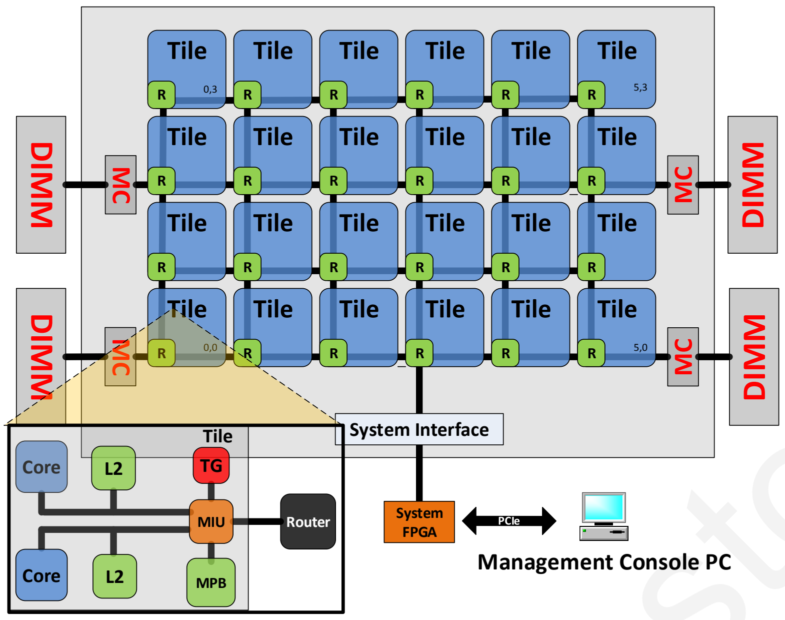


Figure 4.1: Intel SCC top-level and tile top-level architecture.

through the network, a Traffic Generator (TG) for testing the performance of the on-chip network, a Mesh Interface Unit (MIU) connecting the tile to its' network router and two test-and-set registers.

The maximum main memory the current system can support is 64GB and the 32-bit memory addresses of the cores are translated into system addresses by the MIU through a lookup table (LUT). The main memory of the system is located outside the chip and the access to it is achieved through four on-chip DDR3 Memory Controllers (MC). The SCC supports both distributed and shared memory models. The system memory is composed of four regions. Each cores' private main memory (*Private off-chip memory*), the systems' global address space (*Shared off-chip memory*), the Message Passing Buffer (MPB) used to store messages to be sent through the network (*Shared on-chip memory*) and the *L2 cache* of each core.

The Intel SCC is equipped with a large number of light-weight processing units with low power consumption and with multiple memory controllers serving them. Based on the characteristics of the Intel SCC, it is understood that future many-core processors will focus on energy-efficient designs. Expensive, in matters of design- and operating-cost, hardware support units will be avoided and other solutions must be exploited. One example in the Intel SCC is the lack of a cache-coherence mechanism [105], where caching is only supported for data allocated on the private address space. Techniques like this though, have an impact on the performance and the programming of new architectures.

Being an experimental processor, the SCC comes with a concept Programming API called RCCE that supports the Single Program Multiple Data (SPMD) model of execution. It supports C/C++ programming languages with RCCE API extensions to implement the message passing communication of the system. More information on the RCCE communication environment and all the operations supported can be found in [15].

4.3 TFluxSCC Implementation

The goal of this work is to achieve performance scalability of the DDM model using the TFlux Platform on the Intel SCC processor. It is also important to show that using the DDM model of execution on the Intel SCC it is possible to exploit the performance even for simple scalable hardware. TFlux uses compiler directives as to define DThreads and the dependences amongst them. Given that TFlux is an existing platform, the programming style and syntax for TFluxSCC were maintained. The global address space of the SCC was used for storing application data. Thus, no communication mechanism is necessary to exchange data between cores as the hardware system itself will take care of this. This way the programming directives of TFlux remained unchanged as no addition information was needed. Therefore, this allows the portability of the already existing applications.

Although the Intel SCC supports global address space, there is no cache-coherence protocol. In order to ensure correctness of the data coming from the global address space, the SCC does not allow the use of the cache for storing application data coming from the global address space. The DDM model though, and consequently the TFluxSCC implementation, doesn't require cache-coherence as it doesn't allow simultaneous access on shared data. In a DDM program, data is shared through the input and output of the threads as they are defined by the dependences declared by using the pragma directives. Consequently, at each moment only one thread can access a specific data structure on the global address space. Thus, caching global address space data is possible when using TFluxSCC. To ensure correctness, the cache is flushed after the thread completes its execution, as to guarantee that the shared data is saved back to memory.

The SCC configuration was modified (using a modified Linux image, provided

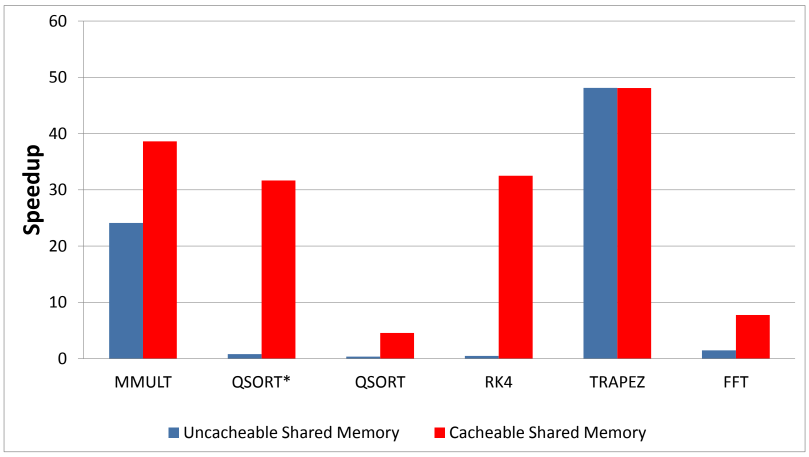


Figure 4.2: Comparing speedup results of uncacheable *against* cacheable *shared memory on the Intel SCC.*

by the SCC platform) as to allow the caching of data coming from the global address space. The TFluxSCC runtime was also modified as to flush the cores caches after writing data to the global address space, using the `DCMflush()` routine. To measure the impact on the performance imposed by this technique both scenarios were tested and Figure 4.2 presents the speedup results for 48-cores. In the first scenario (*Uncacheable Shared Memory*) all the applications tested were implemented in TFluxSCC without caching application data. The second scenario (*Cacheable Shared Memory*) shows the results when allowing the caching of the application data coming from the global address space.

Conventional shared-memory parallel programming models require more sophisticated hardware components, that in systems of 100s -1000s of cores would be both power and performance inefficient due to the extra hardware and the overheads imposed by the coherence protocol [106]. TFluxSCC guarantees correct execution without major impact on the performance while at the same time maintain hardware simple.

4.3.1 TSU Implementations

The TSU implementation in the TFlux Platform is a semi-centralized implementation. This means that except from the TSU thread, part of the scheduling operations are executed by the application threads. This technique was used as an optimization that reduces the overheads of the TSU functionalities on a multi-core system [18]. Since there is only one instance of the TSU structures in TFlux this needs monitoring

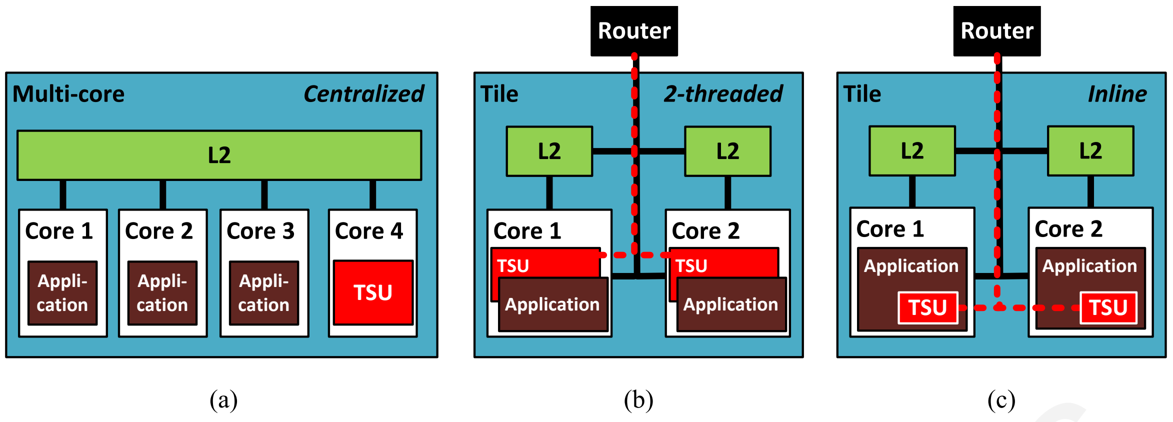


Figure 4.3: TSU Evolution: (a) Original TFlux TSU, (b) 2-threaded *for TFluxSCC and (c)* Inline *for TFluxSCC.*

and locking of shared structures. Using shared structures on systems with a large number of cores will eventually create contention among the cores and locking can possibly result in starvation. For this reason no locking scheme is supported by the SCC.

In the original TFlux TSU (Figure 4.3(a)), the operation that handles the update messages involving all application threads is centralized. This means that the update messages that notify a thread that is ready to execute is sent explicitly by the TSU thread. In large-scale systems and in applications with a large number of update messages this can become a bottleneck to the performance due to contention in the network.

TFluxSCC proposes a non-centralized runtime system that is able to scale and consume as few resources as possible, thus reducing the overheads to a minimum. In TFluxSCC the TSU functionalities is distributed to each core in order to achieve scalability regardless of the number of cores used. Also, only the TSU is allowed to take control of the execution unit at the time needed and only for as long as it is needed, as to reduce the runtime overhead to the minimum. Figure 4.3 shows the evolution process of the TSU from the original TFlux implementation (Figure 4.3(a)) to what is proposed for TFluxSCC (Figure 4.3(c)).

In a first attempt to de-centralize the runtime system a TSU thread was created on every core (Figure 4.3(b)): the *2-threaded* implementation. This implementation leads to one extra thread per core. As the SCC cores do not support hardware multi-threading, depending on the scheduling policy, the OS will switch the execution from the application to the TSU thread allowing the TSU to hold the execution unit from

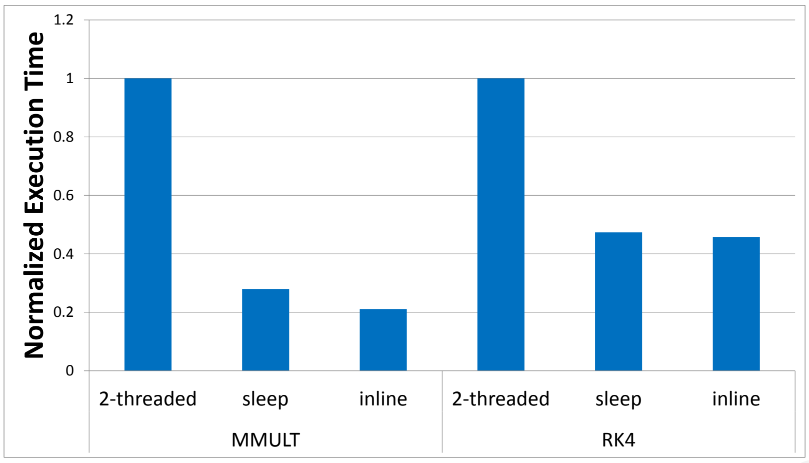


Figure 4.4: Execution time of the implementations of the TSU for 48 cores normalized to the 2-threaded.

the application thread. The TSU implements a busy wait loop in the background until an update message is ready to be send. Thus, in many cases the TSU will take control of the execution unit without having any useful work to do that will help the application progress. To minimize the possible cost of this context switching, the time that the TSU thread uses the execution unit is controlled. At runtime, the TSU thread is deactivated for a certain amount of time by adding a *sleep* call. This way the application thread is allowed to take control of the execution unit for a longer period of time and also reduce the number of times that the OS switches the execution from the application thread to the TSU thread. By limiting the time that the TSU thread uses the execution unit (*sleep* implementation) the total execution time is reduced compared to the baseline implementation (*2-threaded*) as shown in Figure 4.4. Although the *sleep* is effective, it can not be considered for an implementation as the best sleep time varies with the applications and can not be determined in advance. Any runtime mechanism used to determine the best value will impose significant overhead to the execution. While the absence of multi-threading on the Intel SCC is a limitation, it is also a factor of scalability of the hardware. The simpler the cores are, the more that can be integrated on the same processor.

To avoid the above restrictions and limitations the *Inline* implementation is proposed. In this implementation the TSU functionality is integrated with the application thread as shown in Figure 4.3(c). The busy wait loop is removed from the TSU, with its operations being called at the end of the execution of an application thread. This is the only time that the TSU will have real work to do (send update messages

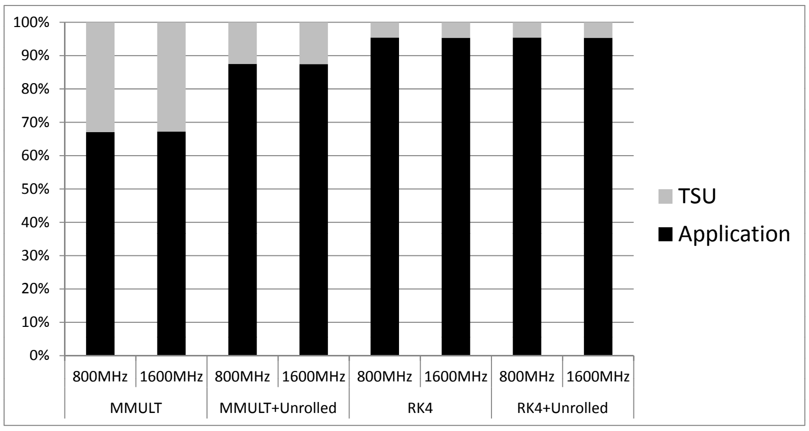


Figure 4.5: Application and TSU execution time breakdown for different frequencies.

to consumers). This solution allows us to utilize the execution unit of the core to the maximum. Figure 4.4 shows the execution time of the three approaches, normalized to the *2-threaded* implementation, for two applications. The results observed verify what was discussed earlier.

Figure 4.5 shows the execution time breakdown into the time for the application. *i.e.* the time spent to execute pure application code, and the time for the *Inline* implementation of the TSU, *i.e.* the time spent to execute the different scheduling procedures, such as the update of the local TSU structures after a thread execution, the exchange of update messages with consumer threads on different cores. Two different network frequencies are also tested to find out whether the message exchange process of the TSU can be improved by the current hardware. In the baseline scenario, the SCC network was operating at 800MHz. The network frequency is then increased to the maximum, which is 1600MHz and observe that the TSU time remains constant. This means that the time spent by the TSU for message exchanging in the *Inline* implementation is not significant, as this process is not network bandwidth dependent. This Figure also shows that the overall TSU overhead is small and is reduced as unrolling is applied to the applications. This happens because in DDM every loop iteration is considered a separate thread, thus the more iterations, the more threads to be scheduled. Using unrolling, the total number of threads to schedule is reduced, therefore reducing the total time consumed by the TSU procedures.

Table 4.1: *Experimental workload description and problem sizes.*

Benchmark	Source	Description	Type	Unroll Factor	Problem Size		
					Small	Medium	Large
MMULT	kernel	Matrix multiply	Compute+Memory	16	64x64	128x128	256x256
QSORT	MiBench	Total Array sorting	Memory	N/A	100K	200K	400K
QSORT*	MiBench	Partial Array sorting	Memory	N/A	100K	200K	400K
RK4	kernel	Differential equations	Compute	1	1024	2048	4096
TRAPEZ	kernel	Trapezoidal rule for integration	Compute	256	30	31	32
FFT	NAS	FFT on a matrix of complex numbers	Compute	1	32	64	N/A

4.3.2 Compilation Toolchain

As described in Section 4.2, Intel SCC processor comes with it's own programming API, called RCCE. Therefore, the RCCE API is integrated into the TFluxSCC preprocessor. The DDM C source-to-source Preprocessor [55] is updated in order to be able to generate code for the Intel SCC Processor. The basic operations added on top of the previous implementations of TFlux are the initialization of the platforms' API, the support for memory allocation on the global address space and most importantly, the flushing of a cores' cache after writing data in the global address space. This later operation is necessary for the model to synchronize explicitly the cache contents to memory as to ensure correctness, given that no hardware cache-coherence is available.

Although the intended platform for the implementation is different, TFluxSCC can be programmed in the exact same way as the original TFlux. This means that the interface of TFlux (*i.e.* the directives in Table 2.5) is kept the same in the TFluxSCC implementation as well. This way, the updated preprocessor provides backward compatibility with the previous implementations of the TFlux Platform. Using a predefined command line flag (*scc*), the preprocessor is signaled to produce code for the Intel SCC platform. Thus, no change is needed to the previously implemented applications.

4.4 Experimental Setup

TFluxSCC was evaluated using six different benchmarks. Three of them are kernels that represent common scientific operations [107], two belong to the *MiBench* [108] suite and one to the NAS [64] suite. QSORT* is a subset of QSORT and represents the partial sorting of an array. In QSORT* the reduction operation is removed and only

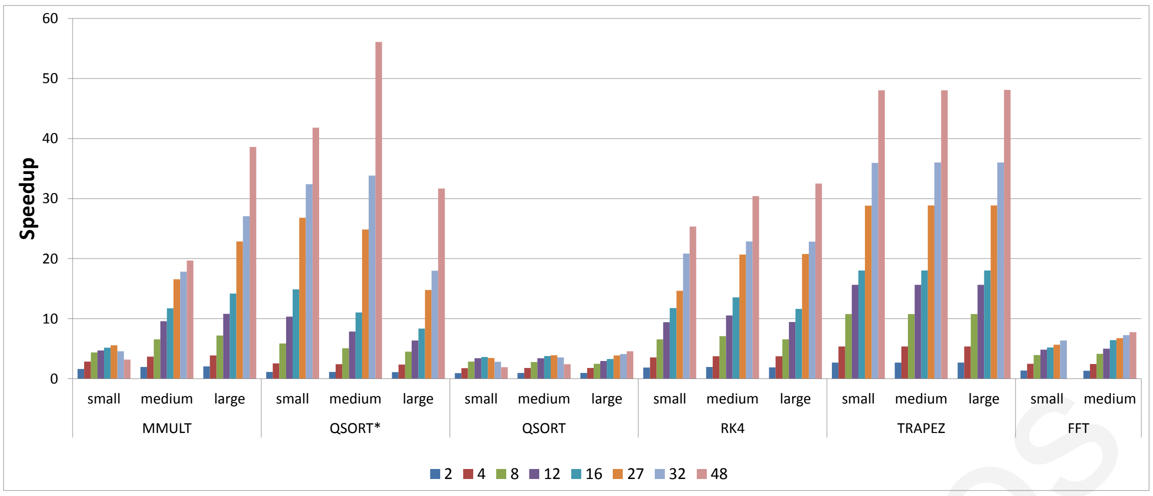


Figure 4.6: Performance scalability of TFluxSCC for different number of cores and input sizes.

the fully parallel part of QSORT is measured, where each core sorts its' own portion of the input array. The benchmarks are briefly described along with their different problem sizes in Table 4.1. For these experiments three different input problem sizes were used: *Small*, *Medium* and *Large*. The execution time was measured and in order to minimize the statistical error, each experiment was executed ten times in order to satisfy a 5% standard deviation. The results shown are the arithmetic average of the measurements after excluding the outliers. The baseline execution for every scenario is the best sequential execution of the benchmarks on a single SCC core.

The hardware setup was an Intel SCC experimental processor, RockyLake version. The system has a total of 32GB of main memory and a balanced frequency setting was used for the experiments of 800MHz for the tile, the mesh interconnection network and the DDR3 memory and Memory Controllers. The operating system used for the Intel SCC cores was the Linux_dcm kernel provided by Intel SCC Communities repository that supports caching the data coming from the off-chip shared-memory to L2 cache. To cross compile the benchmarks for the SCC the GCC v.3.4.5 compiler was used with the optimization flag O3. For porting and executing the applications on the SCC the RCCE v1.4.0 tool-chain [15] was used. Finally, since this study emphasizes on the scalability of the DDM model, all results are reported as *Speedup* compared to the baseline execution.

4.5 Experimental Results

In this work a scalability study of the performance is performed for applications with different characteristics. The experimental workloads include applications that are embarrassingly parallel like QSORT*, applications that are compute-bound like TRAPEZ and others that have a combination of memory- and compute-bound nature, as well as more complex dependences among the different parallel threads. The applications were executed using up to 48 cores and with 3 different input sizes and present in Figure 4.6 the speedup results compared to the best sequential execution.

The results in Figure 4.6 show a large speedup for most applications. The application with the largest overall speedup is TRAPEZ, which is an application that is compute-bound and suffers no memory overheads. RK4, which has a considerable number of threads and dependences achieves also a good speedup and thus it shows that the execution of the TSU code does not incur in a large overhead for the execution of the application. QSORT* shows very impressive speedup, especially for the *medium* input size. The reason that the speedup in this case exceeds 48 is that the input size of QSORT* fits in the cache thus, creating a super-linear speedup phenomenon. MMULT, that is both a compute- and memory-bound application, shows smaller but still large speedup. Finally, FFT and QSORT show the smallest speedup of all applications. QSORT is split into two phases. The first one is like QSORT* and thus has linear speedup. The following phase combines the results of all sorted parts as to build the complete sorted vector. This is done as a reduction using the merge sort algorithm. For this implementation, a binary reduction was too costly so a more optimized n-way reduction was implemented, where at each step one thread combines the results of 4 sorted portions and managed to get better performance.

To see if the data set size affects the performance results the same applications were also tested with 3 different data set sizes. For MMULT and RK4, as the size of the input data increases the speedup increases as expected for compute-bound applications. TRAPEZ achieves linear speedup for the all input sizes. FFT could not scale to the large input size as it required more memory space than what is available from the global address space of the SCC. For QSORT*, which is a memory intensive application, a different behavior was observed as for the larger set the speedup is smaller than for the medium set. This is due to the fact that while for the medium set the portions of data to sort for the 48-core case are still fitting in the cache, for the larger set this is not valid any longer. In QSORT a small difference in the performance was observed while increasing the input size but it is not able to scale more as the reduction phase is a bottleneck for the performance.

4.6 Conclusions and Contributions

In this Chapter the Data-flow model is exploited on a many-core system. Specifically, the TFluxSCC platform is introduced that supports the programming and execution of DDM applications on the Intel SCC processor. Using a set of applications with different characteristics we were able to show that the performance scales well and good speedup is observed for most applications. It is relevant to notice that these are executions of real applications on a real many-core processor and TFluxSCC is the first software implementation of the DDM model for a many-core processor.

TFluxSCC proves that a software implementation of the Data-flow model can produce significant performance benefits on future generations of many-core processors. Being a software system makes it easily configurable as it has limited demands on hardware support that allows for a simpler design with a larger number of execution units and thus increases the parallelism offered by the hardware. The implementation only requires a global address space for storing application data and a selective data-cache flush policy for data that were written and came from the global address space. Hardware support for cache-coherence is not a requirement and this leads to using simpler, more scalable hardware.

The main contributions of this work are as follows:

- TFluxSCC [16]: The first DDM implementation for a many-core processor;
- Evaluation of TFluxSCC on a real 48-core SCC system, achieving up to 48x speedup;

Overall, the results show that TFluxSCC scales well with different applications on a real many-core system. The fact that it gets 48x speedup for 48 cores, for compute-bound applications shows that the runtime does not add any overhead in applications that don't have a performance bottleneck in the algorithm.

Chapter 5

A Scalable Framework for Task-based Data-Flow Execution

In this Chapter we propose a task-based Data-flow runtime called SWITCHES, that implements a lightweight distributed triggering system for runtime dependence resolution and uses static scheduling and compile-time assignment policies to reduce runtime overheads. Unlike other systems, the granularity of loop-tasks can be increased to favor data-locality, even when having dependences across different loops. SWITCHES introduces explicit task resource allocation mechanisms for efficient allocation of resources and adopts the latest OpenMP API, as to maintain high-levels of programming productivity. It provides a source-to-source tool that automatically produces thread-based code. Application performance on an Intel Xeon-Phi shows good scalability and surpasses OpenMP by an average of 32%.

5.1 Motivation

SWITCHES [14] is a lightweight runtime system that supports the task-based Data-flow model as a way to scale performance on many-core architectures. The Data-flow paradigm has been proposed a long time ago [20, 21] but has only recently been widely adopted, as it is one of the most effective ways to exploit large-scale parallelism [9, 22–25, 75, 77, 83, 84]. It is mostly used in the form of task-based parallel systems, that allow for efficient handling of synchronization, memory access and communication latencies. This leads to better utilization of resources and increased performance in large-scale High-Performance Computing (HPC) systems with hundreds to thousands of cores [109].

SWITCHES implements a triggering system for dependence resolution, that distributes the runtime operations to all participating threads. It applies compile-time static scheduling and assignment policies in order to reduce runtime overheads. It improves locality, by providing simple mechanisms that allow flexible definition of task granularity. It provides constructs for declaring cross-loop dependences, consequently increasing the application coverage and improving the performance of applications with parallelism across different loops. It also provides for task resource allocation constructs that make efficient use of the hardware units and improve performance. SWITCHES maintains high levels of programming productivity by extending the latest standard of the widely used OpenMP Application Programming Interface (API) [46], while providing for a software tool that automatically produces parallel code.

Figure 5.1 presents how the state-of-the-art OpenMP runtime handles fine-grain task-parallelism on a many-core system. The graph (on the right) shows the speedup over the sequential execution of a synthetic application as the the number of tasks increases but maintains the total problem size constant. On the left of the figure is the Data-flow graph of the synthetic application that is composed of four parallel loops with a synchronization point between each one and each loop is divided into n tasks. In theory keeping the total work constant should keep the speedup unchanged. Nevertheless, results show that as the number of tasks increases the speedup is reduced. This can be caused by two factors: (1) the amount of work per task and (2) the synchronization overhead of the runtime system. As long as the amount of work per task is large enough, the synchronization time can be hidden, but as the number of tasks increases for a constant workload, the work per task is reduced while the synchronization primitives increase. This results in increased synchronization overhead and consequently a loss in performance.

The application in Figure 5.1 shows the impact of the runtime overhead on performance using different OpenMP scheduling policies. In *OMP-Static* and *OMP-Dynamic* the iterations of each loop are executed in parallel but synchronization points (barriers) must be added between the loops (Data-flow graph on the left). In the former tasks are scheduled statically while in the latter, the runtime handles everything dynamically. The static implementation has much less performance loss as most scheduling operations are handled during compilation. The dynamic scheduler on the other hand has a lot more work to do synchronizing all the tasks

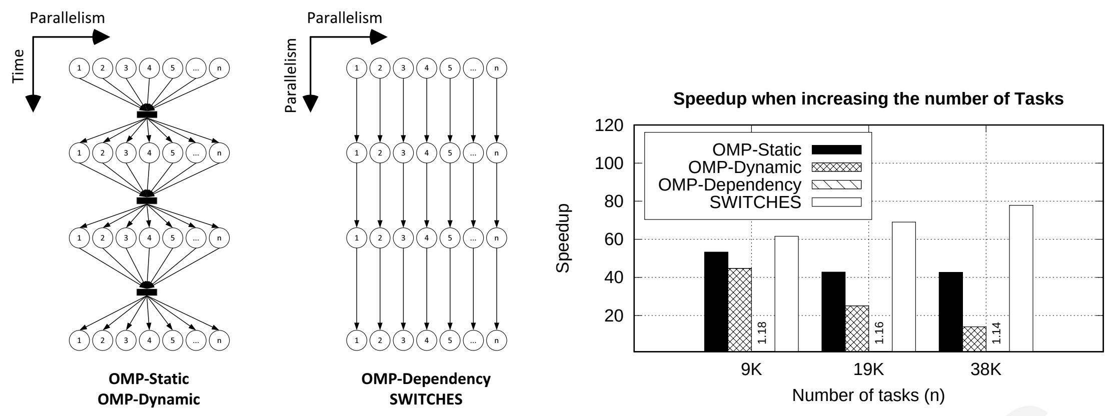


*Figure 5.1: Speedup on the Intel Xeon Phi with 240 threads, varying number of tasks and constant input compared to the sequential execution of a synthetic application. The Data-flow graph on the left shows two applications with four loops (the source code is presented later in Figure 5.5). The one on the left doesn't have dependences across the loops but instead uses synchronization barriers (*OMP-Static *and* OMP-Dynamic*). The one on the right has dependences across iterations of different loops (*OMP-Dependency *and SWITCHES). The graph on the right shows OpenMP loosing performance as the problem size is divided into more tasks (*n*).*

during execution. *OMP-Dependency* and SWITCHES remove the barriers and add dependences across the iterations of the loops (Data-flow graph on the right), but as the results show the overhead of runtime dependence resolution in OpenMP dominates the execution time. The implementation of lightweight synchronization primitives in SWITCHES, in combination with a static runtime system reduces the overheads and achieves the highest performance. Its low-overhead design also results in scaling performance even when increasing the number of tasks. Although this is just a simple application it presents a real problem of current task-based runtime systems with fine-grain parallelism, that can affect performance scalability in current and future many-core systems.

5.2 The SWITCHES System

SWITCHES is built to satisfy two major requirements: (1) the scalability of application performance, and (2) the reduction runtime overheads. To achieve scalability, it implements the Data-flow model as a fully de-centralized runtime by evenly distributing the scheduling operations to all software threads. To reduce runtime overheads, tasks are assigned to threads statically at compile-time. In addition, each task holds its own scheduling structures that will be loaded in the scheduler dur-

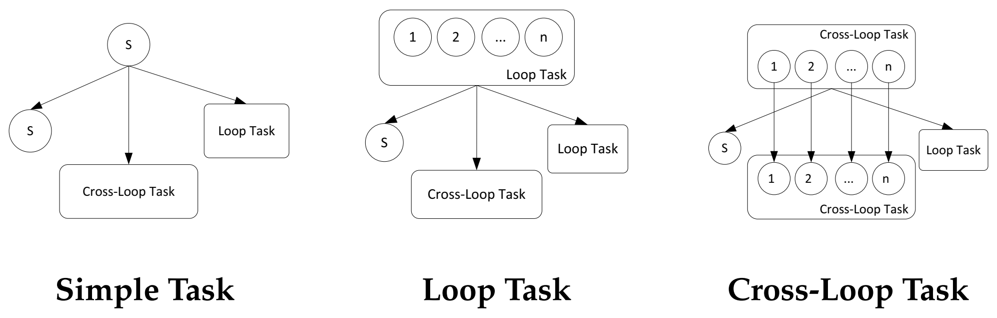


Figure 5.2: Type of tasks in the SWITCHES execution model.

ing execution. This design requires minimum support for dynamic scheduling of tasks, consequently reducing runtime overheads. SWITCHES is publicly available for download in [1].

5.2.1 The Execution Model

SWITCHES is a task-based model that allows the definition of dependences on every task in an application. The dependences form a producer/consumer relationship between tasks, where one task produces data and others consume it. The dependence itself shows the role of each task, *i.e.* if there is a dependence from *taskA* to *taskB*, then *taskA* is the producer and *taskB* is the consumer. The dependences also define when a task can be executed, thus additional synchronization primitives are not required. Dependences are resolved only after all producers have completed execution, and only then can a consumer task execute.

There are three types of tasks in a SWITCHES program (Figure 5.2): (1) `Simple-Tasks`, that represent a non-iterative structured block, (2) `Loop-Tasks`, that represent iterations of a *for* loop and (3) `Cross-Loop-Tasks`, that represent iterations of a *for* loop with dependences on iterations of other loops. The difference between 2 and 3 is the format of their dependences. `Loop-Tasks` are iterations of a loop that will execute in parallel but isolated from the rest of the tasks. That is, if another task (of type 1 or 2) depends on a `Loop-Task`, then it will wait for the entire loop to finish before starting execution. `Cross-Loop-Tasks` are iterations that have direct dependences on iterations of other loops. Assuming a granularity of one, the iterations of a loop have a one-to-one dependence with the iterations of another loop, therefore providing cross-loop parallelism.

The level of granularity in such a scenario can be defined by the programmer. The granularity is increased by packing consecutive iterations into a single task and

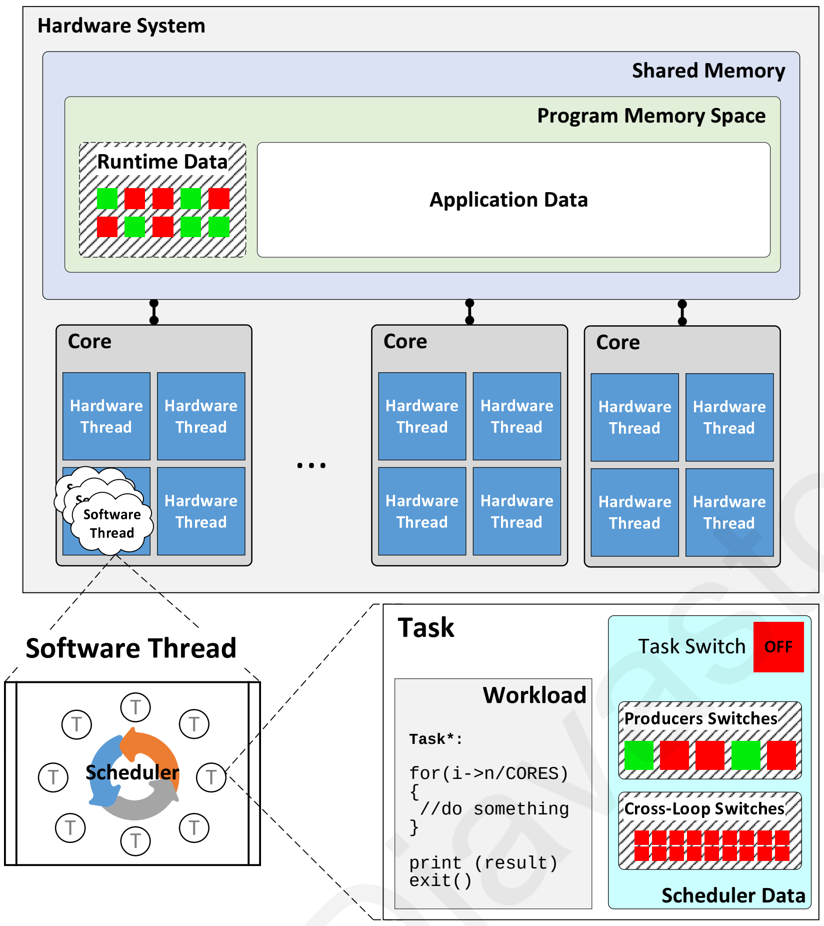


Figure 5.3: *The architectural design of SWITCHES.*

is particularly efficient as it favors spatial locality. A smaller granularity on the other hand, improves the parallelism by increasing the number of tasks (fine-grain parallelism). It is important to notice that, although cross-loop iteration dependences and task-loop granularity are supported by the latest release of the OpenMP standard (v4.5), the combination of the two clauses is not.

5.2.2 The Runtime

The scheduling structures of the runtime are called SWITCHES and are implemented using simple memory constructs, stored in shared-memory and cross-referenced by the tasks using them. Switches are boolean variables that denote whether a task has executed (ON) or not (OFF). Each task is assigned its own unique switch that can only be updated by the thread that executes the task (*single-writer*). A task is ready to execute only when all its producers' switches are set to ON. Each thread checks the producers' switches of a task to be executed (*multiple-readers*) and if all are set to ON, the task is executed.

Since SWITCHES is following a *single-writer / multiple-readers* model for all runtime data, a protection mechanism for simultaneous access (e.g. locking) on switches is not required. Because it is built on top of a shared-memory system, switches are manually updated (from producers caches) to main memory and self-invalidated in consumers caches. This process adds little overhead to the execution as the information accessed by each thread is statically assigned at compile-time.

The architectural design of SWITCHES in Figure 5.3 shows that the data of both the runtime and the application are stored in the shared-memory of the system. One or more software threads can be assigned to each core (or hardware thread), depending on the total number of threads defined by the user. A software thread consists of the SWITCHES *Scheduler* and the tasks it will execute. Each task is a combination of the application source code, its own switch and remote references to all switches of its producers.

The SWITCHES runtime system is implemented in the form of a software unit called *Scheduler*. The *Scheduler* is responsible for monitoring producers' switches, updating task switches and triggering ready tasks for execution.

5.2.3 The Scheduler

Most task-based parallel systems that exist today provide for a single, centralized scheduler that monitors shared-data during execution and resolves dependences based on reads and writes on those data [22,46]. Depending on the number of tasks, and the dependences defined, this can lead to an overload of the scheduling unit which can be expensive (as results show in Figure 5.1).

To solve this, in SWITCHES each thread is equipped with its own *Scheduler*. Every *Scheduler* is statically assigned a number of tasks to execute and only has knowledge of their incoming dependences, based on the cross-references that each task holds on its producers' switches (Figure 5.3). It does not require global information of the application, making it simple and scalable regardless of the number of threads, as the more threads used in the execution the less information each one will hold.

Algorithm 1 The SWITCHES Scheduler.

```
1:  procedure SCHEDULE
2:      while !$empty(ownTaskQueue)$ do
3:          $task \leftarrow getNextTask(ownTaskQueue)$
4:          $ready \leftarrow checkSwitches(task)$
5:          if $ready = TRUE$ then
6:              $execute(task)$
7:              $turnSwitchON(task)$
8:              $removeTaskFromQueue(task, ownTaskQueue)$
9:          end if
10:     end while
11:     $resetSwitches(ownTaskSwitches)$
12: end procedure
```

Algorithm 1 shows the pseudo-code of the *Scheduler*. The *Scheduler* is assigned a number of tasks at compile-time in its `ownTaskQueue` and executes a busy-wait loop (line 2) until all assigned tasks are finished. The *Scheduler* first gets a waiting task from its `ownTaskQueue` (line 3). It checks the switches of its producers and if all are set to ON, it triggers the task for execution (line 4). If at least one producer has not yet finished, the task is not ready for execution and the *Scheduler* gets the next task from its `ownTaskQueue`. To minimize long waits of tasks that are ready to execute, the `checkSwitches()` operation (in line 4) stops immediately when it finds the first producer switch that is not set to ON. Also, as soon as a task finishes execution, the *Scheduler* sets its switch to ON (line 7) and removes it from the `ownTaskQueue` (line 8), so it won't be checked again.

When a thread finishes all its assigned tasks (its `ownTaskQueue` is empty) the *Scheduler* breaks the busy-wait loop, resets all its tasks' switches (line 11), exits the current parallel section and returns to the main program. Switches are reset to avoid creating multiple instances of the same code and the same switches in case there is a repetitive execution of the same tasks later in the program (*e.g.* a function that is called multiple times). To avoid reseting switches that are still in use by other threads, resetting takes places only after all tasks of a parallel section have completed.

5.3 The SWITCHES API

The API of SWITCHES is an extension to the latest OpenMP v4.5 [90]. The applications are written in C/C++ with tasks and dependences declared using compiler directives. The API of OpenMP was chosen because it is widely used for parallel programming in shared-memory environments and, since v4.0 provides a complete set of directives for declaring tasks and dependences. Consequently, it satisfies the productivity goal of quickly porting applications for execution with SWITCHES. The SWITCHES API implements all tasks directives from OpenMP v4.5 and extends them by using existing clauses from non-task directives such as `num_threads`, `schedule` and `reduction`. Table 5.1 summarizes the basic directives used for writing a SWITCHES program and highlights in bold the extensions implemented. It is also important to notice that even though at this time only a portion of the OpenMP API is implemented, the SWITCHES runtime will not become heavier if it implements the entire API as it follow a static runtime approach. All necessary scheduling information are produced during the compilation of each application keeping the runtime system simple and light. For conventional OpenMP implementations (that use a dynamic runtime system) to be more flexible with different application most scheduling data is produced at runtime and require more information to be kept by the runtime system.

5.3.1 Compiler Directives

Any tasks declared in a SWITCHES program must be enclosed in a parallel section using a `#pragma omp parallel` directive. A parallel section works in the same way as in OpenMP and denotes that all enclosed tasks are to be executed in parallel according to their dependences. The use of `single` directive is not required as in OpenMP, since tasks will only be created once statically at compile time and started when the master thread reaches the specific parallel section. Any code written in a parallel section that is not enclosed in a `task` directive will be executed by all participating threads as in OpenMP. Different parallel sections are executed sequentially in the order found in the program, just like in OpenMP. Also, all data are considered shared across the entire program unless declared otherwise using the optional `private` or `firstprivate` clauses. The former creates a new empty

Table 5.1: The basic SWITCHES Programming API. In bold the proposed extensions are emphasized compared to OpenMP[a].

Pragma Directives	Supported Clauses (Optional)	Description

`#pragma omp parallel`	`num_threads(NUMBER)`
	`private(list)`
		Defines a parallel function to be executed by `NUMBER` threads and `private` is used to declare variables as private to each thread.

`#pragma omp parallel for`	`private(list)`
	`num_threads(NUMBER)`
	`schedule(type:[,CHUNK])`
	`reduction(OPERATION:list)`
		Defines a parallel loop with each iteration considered a task. All clauses have the same functionality as in OpenMP.

`#pragma omp task`
`#pragma omp master`
	`private(list)`
	`firstprivate(list)`
	`depend(type:list)`
		Defines a task and its dependences using the `depend()` clause. `private` and `firstprivate` have the same functionality as in OpenMP. The `master` clause in SWITCHES extends a normal `task` that will be executed by the master thread.

`#pragma omp taskloop`
`#pragma omp for`
	`private(list)`
	`firstprivate(list)`
	`grainsize(CHUNK)`
	depend`(type:list)`
	num_threads`(NUMBER)`
	schedule`(type:[,CHUNK])`
	reduction`(OPERATION:list)`
		Defines a loop task with `grainsize` defining the number of consecutive iterations assigned to each task. The depend clause is used to apply dependences to the iterations, while the `num_threads` clause explicitly allocates resources for a loop task. The `schedule` clause defines the scheduling policy of the loop (`static` or `cross`). A `taskloop` directive can also support a reduction function using the `reduction` clause.

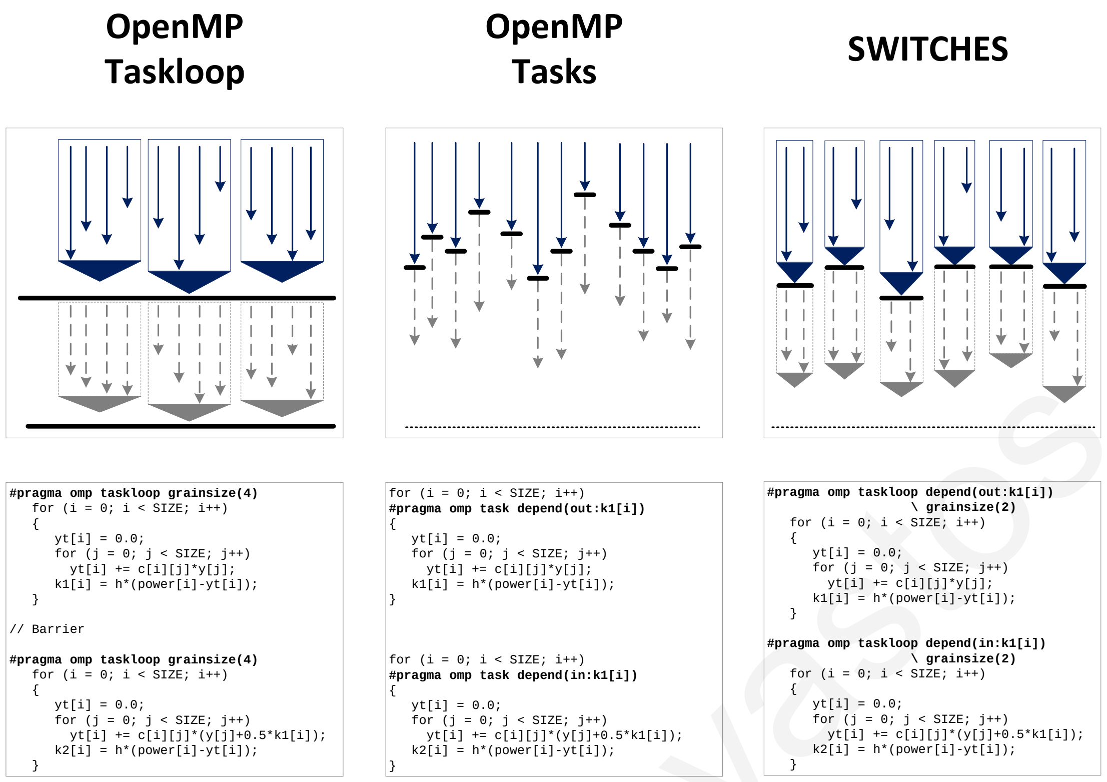
[a]More directives are implemented but in this table only the most relevant ones to the applications tested are presented. Details on all supported directives and instructions for installing SWITCHES can be found in Appendix A and on-line in [1]

variable private for every task, while the latter will also initialize it with the value of the corresponding global variable at the time the task is called.

The #pragma omp task directive defines a structured block as a task of type Simple-Task. The depend(type:list) clause defines the data processed by the task. The list argument holds the name of the data (variables, arrays, etc.). The type argument indicates whether the data will be read (in), written (out), or both (inout).

The #pragma omp taskloop directive declares iterations of a loop as tasks of type Loop-Task or Cross-Loop-Task. Similarly to OpenMP, the grainsize clause defines the number of consecutive iterations to be packed for each task. In contrast to OpenMP, in SWITCHES the user can explicitly define the number of threads to use for the execution of a taskloop with the num_threads clause. This can be beneficial for applications with limited parallelism where loops with little work do not occupy all available resources. By using this clause it is possible to utilize cores, that would otherwise be idle or free cores that don't do any work at all. SWITCHES also supports the definition of dependences on a taskloop directive. The depend clause is used as described earlier for the task directive. The SWITCHES taskloop directive also implements the schedule clause (from the #pragma omp for directive). This clause is used to define a policy for scheduling loop tasks to threads. Two policies are supported at the moment, (1) static, which is similar to OpenMP and (2) cross, that declares the iterations of the loop as tasks with dependences. With static, iterations of the loop are declared as tasks of type Loop-Tasks, while with cross iterations are declared as Cross-Loop-Tasks. Note that, in such case all associated loops must have the cross policy.

If the static scheduling policy is used along with the depend statement, a single dependence will be declared on the entire loop. If the cross policy is used, dependences will be applied on individual iterations of the associated loops and create a scenario with *Cross-Loop Iteration Dependences*. The CHUNK parameter is used to define the number of consecutive iterations to be packed in a single task (similarly to the grainsize clause). The final extension of SWITCHES compared to OpenMP, is the reduction clause where a loop of tasks can be declared as having a reduction operation after all iterations are completed. SWITCHES supports all OpenMP standard reduction operations with multiple reduction variables declared in the list option.



```
#pragma omp taskloop grainsize(4)
   for (i = 0; i < SIZE; i++)
   {
      yt[i] = 0.0;
      for (j = 0; j < SIZE; j++)
        yt[i] += c[i][j]*y[j];
      k1[i] = h*(power[i]-yt[i]);
   }

// Barrier

#pragma omp taskloop grainsize(4)
   for (i = 0; i < SIZE; i++)
   {
      yt[i] = 0.0;
      for (j = 0; j < SIZE; j++)
        yt[i] += c[i][j]*(y[j]+0.5*k1[i]);
      k2[i] = h*(power[i]-yt[i]);
   }
```

```
for (i = 0; i < SIZE; i++)
#pragma omp task depend(out:k1[i])
{
   yt[i] = 0.0;
   for (j = 0; j < SIZE; j++)
     yt[i] += c[i][j]*y[j];
   k1[i] = h*(power[i]-yt[i]);
}



for (i = 0; i < SIZE; i++)
#pragma omp task depend(in:k1[i])
{
   yt[i] = 0.0;
   for (j = 0; j < SIZE; j++)
     yt[i] += c[i][j]*(y[j]+0.5*k1[i]);
   k2[i] = h*(power[i]-yt[i]);
}
```

```
#pragma omp taskloop depend(out:k1[i])
                   \ grainsize(2)
   for (i = 0; i < SIZE; i++)
   {
      yt[i] = 0.0;
      for (j = 0; j < SIZE; j++)
        yt[i] += c[i][j]*y[j];
      k1[i] = h*(power[i]-yt[i]);
   }

#pragma omp taskloop depend(in:k1[i])
                   \ grainsize(2)
   for (i = 0; i < SIZE; i++)
   {
      yt[i] = 0.0;
      for (j = 0; j < SIZE; j++)
        yt[i] += c[i][j]*(y[j]+0.5*k1[i]);
      k2[i] = h*(power[i]-yt[i]);
   }
```

Figure 5.4: Cross-Loop Iterations Dependences.

5.3.2 Cross-Loop Iteration Dependences

As explained earlier, OpenMP does not allow the definition of dependences on a `taskloop` directive. It supports dependences on loop iterations only by using the `task` directive within the body of a `for` loop. This limits the level of granularity of tasks with dependences to one iteration per task, affecting the locality of the data and limiting the performance in certain applications. To increase the level of granularity of a taskloop, OpenMP offers the `grainsize` clause but since dependences cannot be defined on a `taskloop` directive, a synchronization barrier will be inserted at the end of the loop. This adds additional synchronization overheads and ignores cross-loop parallelism that may exist between two different loops (see Figure 5.4).

With the extensions on the `taskloop` directive described earlier, SWITCHES allows the packing of consecutive iterations into a single task and can then define dependences on these tasks (`Cross-Loop-Tasks`). Thus, SWITCHES provides locality for tasks of the same loop and at the same time offers asynchronous execution of tasks from different loops (cross-loop parallelism) by removing all barrier synchronization. Note that this is applied to different loops, in contrast to OpenMPs' *doacross* technique that uses the `ordered` directive to serialize iterations in nested loops [90].

5.3.3 Task Resource Allocation

The distribution of resources within a parallel region in OpenMP is a responsibility of the runtime. OpenMP only allows explicit definition of resources in a `parallel` construct, that denote how many threads will participate in the specified parallel region. But, the increased parallelism and dependences in task-based models, provide the user with information that can be vital to the performance of an application. Dependences can be a natural limitation in the scalability of an algorithm, and using the entire pool of execution units leads to wasting resources. A possible solution in a dynamic runtime system is the use of a work-stealing approach that reassigns tasks to different threads during execution. Such a technique tries to balance the workload in the available resources but it also increases the work of the scheduler and depending on the application it may increase runtime overheads. Such a solution cannot be used in a static runtime system as it would diminish all the benefits of a static implementation. To address this problem, SWITCHES employs the explicit allocation of resources per task. SWITCHES extends the `taskloop` directive to use the `num_threads` clause to allow for explicit definition of threads to be used for the execution of `Loop-Tasks` and `Cross-Loop-Tasks`. Therefore, SWITCHES allows for finer-grain allocation of resources that overcomes the scalability boundary in algorithms with data-dependences and efficiently uses the available resources.

5.3.4 Example

Figure 5.5 shows an example that makes use of the extensions proposed. It depicts a kernel with two loops that have cross-loop iteration dependences and split the execution resources. Figure 5.5 also presents a graphical diagram of the same example. *Loop A* is producing data in array `k1`, while *Loop B* consumes data from `k1`, thus defining a dependence. Analyzing the algorithm, the iterations of the two loops have a one-to-one dependence on array `k1` only, thus only `k1` needs to be declared in the `depend` statements. All other data used by the two loops could have been declared as well but the system would have ignored them, as there are no true dependences on any other data. To define the cross-loop dependence, the scheduling policy of the two loops is declared as `cross` and in the `depend` clause the indexes of the continuous dependent iterations are added. Note that the `CHUNK` in the `schedule` clause must be the same number as the end-index of the dependences, so that consecutive

```
#pragma omp parallel num_threads(10)
{
   // Loop A
   #pragma omp taskloop schedule(cross , 2) depend(out: k1[0:2]) num_threads(5)
   {
      for (i = 0; i < SIZE; i++)
      {
         yt[i] = 0.0;
         for (j = 0; j < SIZE; j++)
            yt[i] += c[i][j]*y[j];
         k1[i] = h*(power[i]-yt[i]);
      }
   }

   // Loop B
   #pragma omp taskloop schedule(cross , 2) depend(in: k1[0:2]) depend(out: k2[0:2])
                                             \ num_threads(5)
   {
      for (j = 0; j < SIZE; j++)
      {
         yt[j] = 0.0;
         for (k = 0; k < SIZE; k++)
            yt[j] += c[j][k]*(y[k]+0.5*k1[j]);
         k2[j] = h*(power[j]-yt[j]);
      }
   }
   ...
}
```

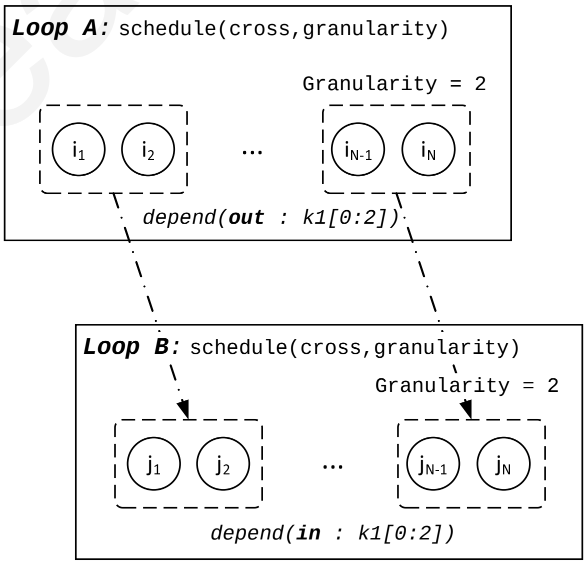
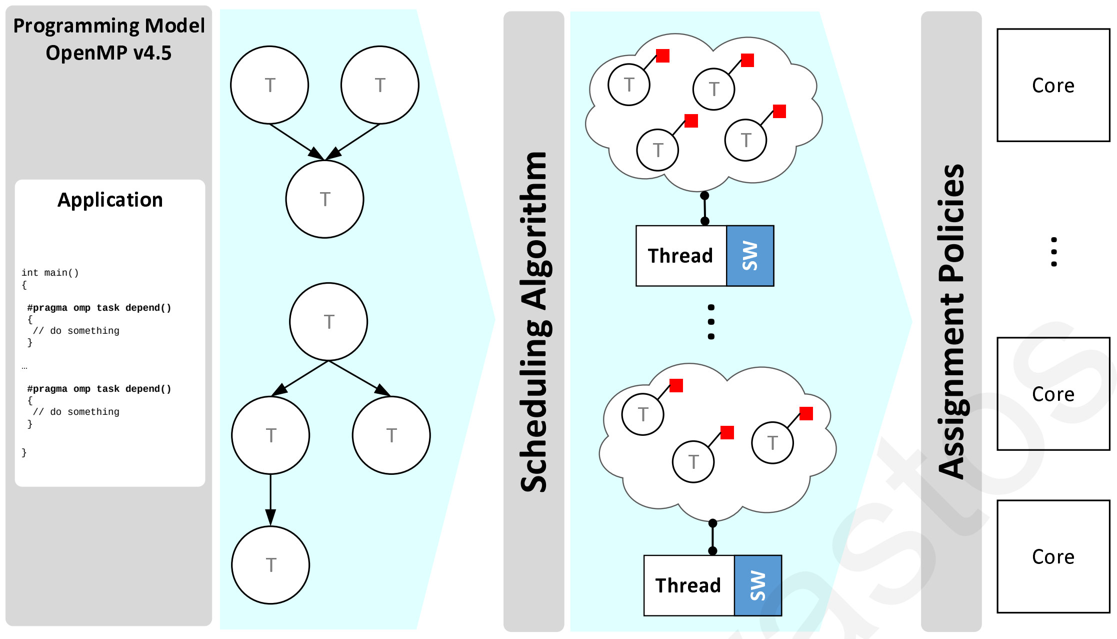


Figure 5.5: Cross-loop Iteration Dependences example and diagram. This example is part of a larger synthetic application that is based on a differential equation kernel (RK4, presented in Section 5.5). The Data-flow graph of the entire application is shown in Figure 5.1.



```
int main()
{

 #pragma omp task depend()
 {
  // do something
 }

…

 #pragma omp task depend()
 {
  // do something
 }


}
```

Figure 5.6: The layered design of SWITCHES and Translators operations.

dependent iterations are packed to the same task. Summarizing, the example shows that each loop consists of tasks with chunk size of two iteration per task, and every task of *Loop A* has a direct dependence on the corresponding task of *Loop B*. Finally, it is explicitly specified that only 5 threads will be used to execute the tasks of each loop. Thus, the two loops split the available resources of the parallel region (10 threads) to execute their tasks in parallel.

5.4 The Translator

The translation of a directive-based application to a SWITCHES parallel program is automatically done by a source-to-source tool called the *Translator*. The *Translator* is a software tool built using Lex and Yacc that parses the C/C++ directive-based code and produces threaded parallel code. Pragma directives can be inserted anywhere in the code and also in multiple files. The *Translator* also produces error and warning messages when directives or clauses are not used correctly. Such messages include directive syntax errors, unknown parameters in clauses, mismatching resource allocation values among others.

The layered design of SWITCHES and the operations executed by the *Translator* are shown in Figure 5.6. The *Translator* takes four major inputs from the programmer:

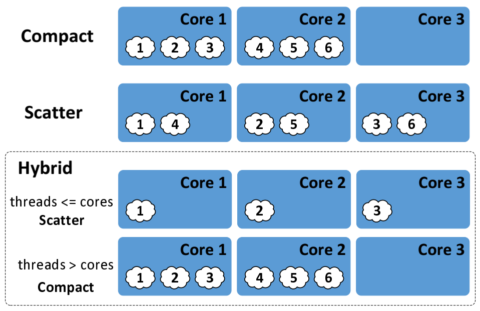
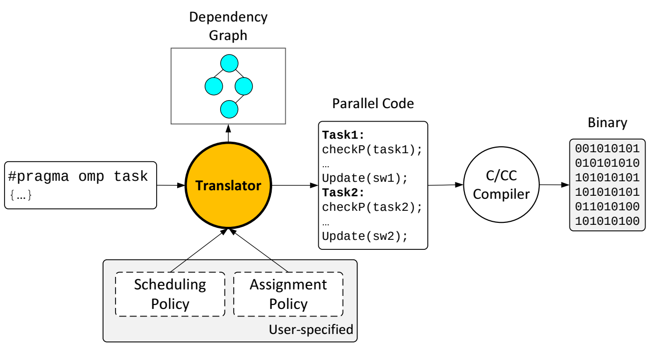


Figure 5.7: Assignment policies.



```
#pragma omp task
{…}
```

```
Task1:
checkP(task1);
…
Update(sw1);
Task2:
checkP(task2);
…
Update(sw2);
```

```
001010101
010101010
101010101
101010101
011010100
101010100
```

Figure 5.8: Application translation procedure.

(1) *the source code* files, embedded with pragma-directives, (2) *the scheduling policy*, that defines how tasks are divided to participating software threads, (3) *the assignment policy*, that denotes how software threads are assigned to hardware resources, and (4) *the number of threads*, that execute the application. The number of software threads can be as many the Operating System (OS) allows. After parsing the directives in the source code, it automatically extracts the tasks and their dependences and produces the synchronization graph of the application. It then executes a *transitive reduction* operation on the graph to remove redundant dependences that might have been implicitly or explicitly declared [110]. This optimization reduces the size of the graph and the runtime data structures that are produced, thus reducing the workload of the scheduler and minimizing runtime overheads.

To further reduce runtime overheads compared to other dynamic parallel systems, many of the scheduling operations in SWITCHES are moved to compile-time and executed by the *Translator*. The *Translator* will use the final graph to impose the scheduling and assignment policies by statically mapping tasks to threads and threads to cores respectively. At the moment SWITCHES schedules tasks based on the availability of the threads using a *round-robin* scheme. During the program translation the user can choose from 3 predefined assignment policies: `Compact` and `Scatter` that can also be found in the *icc* compiler as affinity policies (`KMP_AFFINITY`) or the *gcc* compiler with the names `close` and `spread`. The third and new assignment policy is called `Hybrid` as it implements a combination of the previous two. Each policy is presented in the example of Figure 5.7 where a system with 3 cores and 3 hardware threads per core is assumed. The `Compact` policy assigns software threads close to each other occupying the hardware thread units of a core before moving to the next one. This policy does not utilize all cores when the software threads are less than the hardware threads of a system. To evenly utilize all available cores, the `Scatter` policy assigns the software threads to the cores in a round-robin way. The `Hybrid` policy is a combination of the two previous, where `Scatter` is used when the software threads are fewer than the number of the available cores to increase processor utilization, while `Compact` is used when the number of threads is more than the available cores to favor locality of shared data in the caches. The `Hybrid` policy is implemented for simplicity of the execution of the *Translator*. Depending on the number of threads defined by the user at the translation stage, the system automatically determines the appropriate policy to use. All policies use the software *thread-id* to assign the threads to the cores.

After invoking the *Scheduler* it creates the output parallel source code as shown in Figure 5.8. It can also generate a fully-detailed synchronization graph for possible debugging of the application. The produced source files consist of the tasks code, the creation of the software threads (pthreads) and the SWITCHES runtime system. The output source code can be compiled with any commodity C/C++ compiler.

5.5 Experimental Setup

SWITCHES is evaluated on a set of seven data- and task-parallel applications. Applications were chosen based on references from other evaluations of task-based runtime systems and many-core processors ([18, 72, 109, 111, 112]). Details for all the applications and their input sizes are shown in Table 5.2. The Data Set Sizes represent the total number of computation iterations each application executes on its data.

As representatives of data-parallel applications the following ones are used: (1) Q12, a C-code version of Query 12 from the TPC-H Benchmark suite [73] that emulates the Scan and Join operations on the data from two tables representing a memory-bound application, (2) MMULT, implements a matrix multiplication algorithm [18], (3) RK4, solves a differential equation [18] and (4) SU3, is a component of Wilson Dirac equation that involves the multiplication with the gauge-links (vector multiplication of complex C99 numbers) [111].

Task-parallel applications include: (1) Poisson2D, a 5-point 2D stencil computational kernel of the Poisson equation from the KASTORS Benchmark suite [109], (2) SparseLU, from the BOTS Benchmark suite [112] that computes an LU matrix fac-

Table 5.2: *Experimental workloads description and data set sizes.*

Benchmark		Description	Data Set Sizes			Compared Implementations
Type	Name		(Computation Iterations)			
			DS1	DS2	DS3	
Data-parallel	Q12	Nested-Loop Join from TPC-H [73]	$60K \cdot 1.5K$	$60K \cdot 15K$	$600K \cdot 150K$	OMP-Dynamic-For
	MMULT	Matrix multiply [18]	$256 \cdot 256$	$512 \cdot 512$	$1024 \cdot 1024$	OMP-Static-For
	RK4	Differential equation [18]	4800	9600	19200	OMP-Task
	SU3	Wilson Dirac equation [111]	$1920K$	$3840K$	$7680K$	OMP-Taskloop
Task-parallel	Poisson2D	5-point 2D stencil computation [109]	4096	8192	16384	OMP-Task
	SparseLU	LU factorization of sparse matrices [112]	$120 \cdot 32$	$240 \cdot 32$	$480 \cdot 32$	OMP-Task-Dep
	OCEAN	Red-Black solution (Gauss-Seidel [113])	$4096 \cdot 4096$	$8192 \cdot 8192$	$16384 \cdot 16384$	OMP-Taskloop

torization using sparse matrices creating an imbalance workload and (3) OCEAN, representing a Red-Black solution of the Gauss-Seidel method [113].

The main evaluation platform is an Intel Xeon Phi 7120P (Knights Corner, KNC) with 61 cores and 4 threads per core (totaling 244 hardware threads). Note that only 60 cores are used as to avoid any interference by the OS that always uses the last core of the system. This board has a total of 16GB of main memory and runs at 1.238GHz. To cross-compile applications for the Xeon Phi the Intel icc v.17.0.2 compiler is used (and the corresponding libiomp5 library) with the *-mmic* flag indicating the Many Integrated Architecture (MIC) target. SWITCHES was also tested on a smaller system, a 12-core machine with 2 6-core (12 hardware threads) AMD Opteron 2427 running at 2.2GHz with an available main memory of 31GB. This system is running an Ubuntu SMP x86-64 OS with the gcc v.5.4 compiler (with libgomp1 v.6.2). For both compilers the *-O3* optimization flag was used. The results are presented as *Speedup*, calculated by dividing the execution time of the best sequential implementation of each application with the time of the parallel execution. The execution times are collected using the `gettimeofday` system call that provides a resolution of microsecond. The time is measured from the start of the first parallel function until the last, including all runtime costs (such as thread creation and scheduler initialization). Execution time is determined by the arithmetic average of five consecutive execution runs after removing the outliers (for all experiments, the standard deviation is within 5%).

5.6 Experimental Evaluation

The main objectives of the evaluation are: (1) to show the performance scalability achieved by SWITCHES, and (2) to compare it with the state-of-the-art OpenMP. The OpenMP applications used in the evaluation, are the original source codes taken from the suites described in Table 5.2. In BOTS suite, SparseLU is implemented with both the `parallel_for` and the `omp_single` directives for creating tasks. In these tests the `omp_single` version was used because it makes the implementation purely task-based and identical to the scenarios tested in KASTORS [109]. In order to provide a fair comparison of the two runtime systems, OpenMP source codes were used as is for the SWITCHES evaluation without extra optimizations. Only the syntax of the directives for the scenarios with cross-loop dependences were modified since OpenMP currently doesn't support it.

As far as KNC-specific optimizations, only SU(3) uses vector intrinsics for both implementations as it was the only application that already supported it. Because this works studies performance scalability of the runtimes on a large-scale many-core and not the capabilities of the underlying hardware, we chose not to alter the original source codes with hardware-specific optimizations. Scalability results for each application are for the largest data set. For granularity the value that produces the highest speedup for each application was used. The input size is defined in the title and the granularity for each runtime tested is shown in parenthesis in the key-legend of each chart. Section 5.6.4 shows how each system performs for each data set by presenting results of a weak scaling test. SWITCHES results are taken using the *hybrid* assignment policy to increase resource utilization when the number of threads is less than the number of available cores (60).

5.6.1 Data-Parallel Application

Data-parallel applications (Q12, MMULT, RK4 and SU3), are compared against the OpenMP `parallel-for` using both `static` and `dynamic` scheduling policies (*Static-For* and *Dynamic-For* respectively). For the scenarios using the `dynamic` policy, the runtime system dynamically decides all scheduling options, therefore granularity is not statically changed. The applications are also implemented using OpenMP tasks (*Task*). Note that the version of the OpenMP library used at the time of this work still

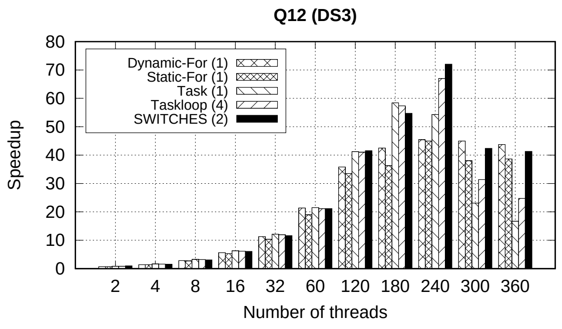


Figure 5.9: Q12 speedup on the KNC.

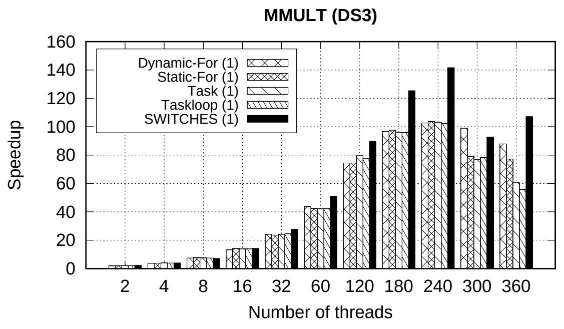


Figure 5.10: MMULT speedup on the KNC.

doesn't support the OpenMP v4.5 (that includes the `taskloop` directive), therefore to support higher granularities in OpenMP the taskloop scenarios (*Taskloop*) were manually implemented using the `taskgroup` directive, as suggested by the Red Hat Developer Program in [114]. The granularity level used to achieve the highest performance for the specific system is shown in paranthesis next to the name of each system in the charts.

Results in Figure 5.9 show Q12 as an application that benefits more from a task-based runtime system. A task-based implementation provides isolation to the execution of each task, therefore allows better scheduling of the Q12 workload as the tasks execute independently. Both SWITCHES and OpenMP (*Taskloop*) achieve the highest speedup for 240 threads (72× and 67× respectively) but only in the scenarios where the granularity of tasks is increased. The rest of the OpenMP implementations achieve a maximum speedup at only 180 threads. The work distribution at 240 threads is too fine-grain to hide the runtime overheads of these implementations, while the light-weight runtime of SWITCHES achieves the highest performance at 240 threads. Oversubscribing the KNC to 300 and 360 threads results in degraded performance as it can cause higher resource contention and pipeline latencies [115].

MMULT in Figure 5.10 shows that different OpenMP implementations achieve the same performance. Also, increasing the number of threads from 180 to 240 results in little benefit for OpenMP. The size of the work assigned to each thread decreases as the number of threads increases, creating fine-grain parallelism, with the overhead of the OpenMP runtime dominating the execution. The low overhead imposed by SWITCHES allows scalability regardless of the number of threads and benefits compared to OpenMP as the number of threads increases. SWITCHES achieves a speedup of 141×, while the best OpenMP results is 103× for *Static-For*.

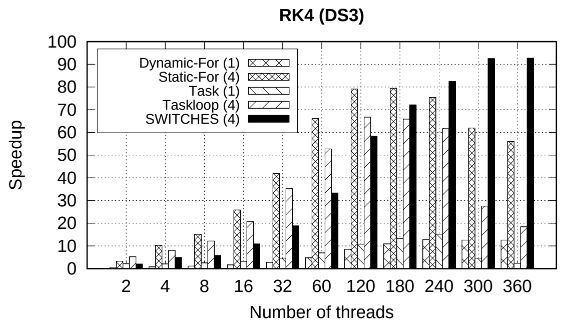


Figure 5.11: RK4 speedup on the KNC.

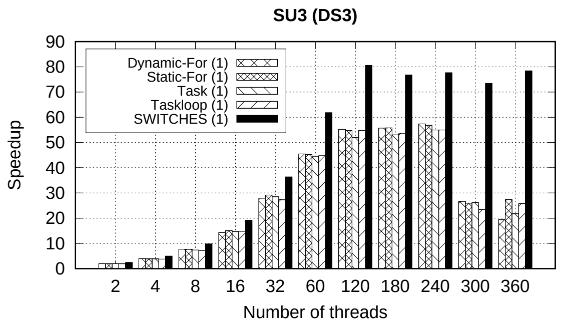


Figure 5.12: SU3 speedup on the KNC.

Again, oversubscribing the cores reduces the speedup due to overheads imposed due to resource contention.

RK4 (results in Figure 5.11) is a load-balanced application where each iteration of the containing loops produces the same amount of work. Its algorithm suggests that consecutive iterations use data from consecutive memory locations. Therefore, the best results are produced with the implementations that increase the granularity to favor locality (*Static-For* (79×), *Taskloop* (66×) and SWITCHES (92×)). The *Task* and *Dynamic-For* implementations are limited to a maximum of 15× and 10× speedup respectively, because the runtime does not take into account the data-locality of adjacent tasks. Because each task in RK4 uses different data for its computations, oversubscribing of the cores could hide memory latencies and increase the performance and that is why SWITCHES performance improves after 240 threads.

SU3 is another application where the impact of the runtime system is shows on its execution (Figure 5.12). Although the algorithm implemented limits the scalability of all systems to 120 threads, the low overhead of SWITCHES to the execution allows it to achieve a speedup of 81× compared to the best OpenMP result of 57× (*Dynamic-For*). SU3 shows a large drop of performance when the number of threads is increased to more than 240, in contrast to SWITCHES that maintains steady performance.

5.6.2 Task-Parallel Applications

SparseLU, OCEAN and Poisson2D are compared against OpenMP task-based implementations with dependences (*Task-Dep*), and without dependences (*Task* and *Taskloop*). The latter require explicit declaration of synchronization between parallel loops as dependences are not declared.

SparseLU is an application that provides cross-loop parallelism between three

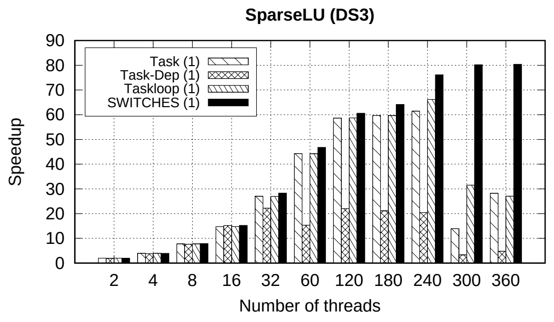


Figure 5.13: LU speedup on the KNC.

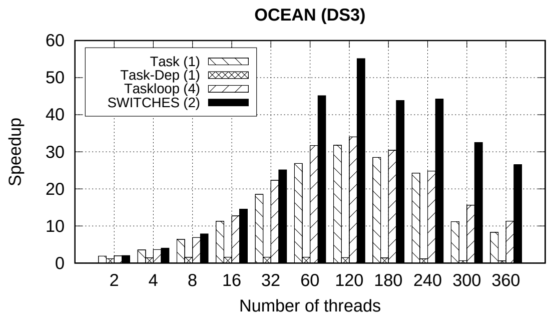


Figure 5.14: Ocean speedup on the KNC.

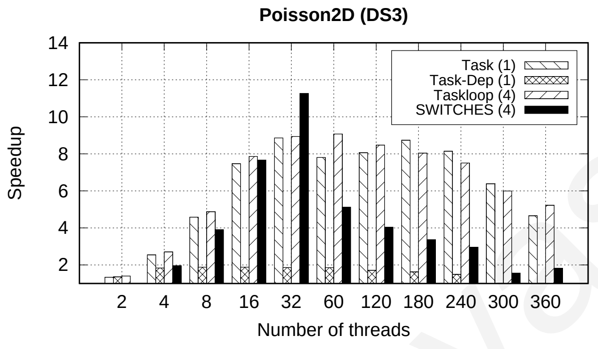


Figure 5.15: Poisson2D speedup on the KNC.

loops that can be expressed using tasks with dependences. But, results in Figure 5.13 show that if dependences are applied on OpenMP tasks (*Task-Dep*), the parallelism offered cannot produce enough performance to overcome the overhead of runtime dependence resolution. When dependences are declared in OpenMP, shared data are packed in a single dependence graph and marked for monitoring during execution. This monitoring is managed by a centralized runtime system with its workload increasing as the amount of data to monitor is increased. If the dependences are removed, this overhead is removed but parallelism across loops is not exposed and potential performance is lost. Because SWITCHES takes care of the dependences and the scheduling during compilation, it is possible to expose parallelism across loops without incurring additional runtime overheads and increase the performance over all OpenMP implementations, achieving a speedup of 80× compared to the 66× of *Taskloop*. Oversubscribing cores in SparseLU helps SWITCHES to improve performance beyond 240 threads, while for OpenMP the performance degrades as it happens in RK4.

OCEAN (Figure 5.14) has a very balanced workload and using dependences exposes even more parallelism. But the large number of dependences in the algorithm produce too much overhead for the OpenMP runtime to handle efficiently. The static

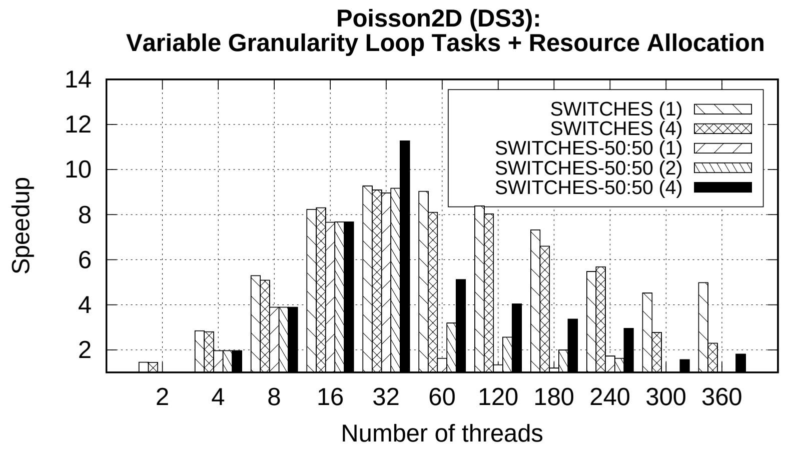


Figure 5.16: Performance when using the Cross-Loop dependence support in combination with increasing granularity and allocation of resources (50:50 means the threads are equally divided between the two loops).

and distributed nature of SWITCHES dependence resolution avoids the extra overhead and produces as speedup of 55× for 120 threads. Increasing the granularity of tasks benefits the execution even more and the speedup of *Taskloop* increases to 34× compared to *Tasks'* 31×. The algorithm of Poisson2D (Figure 5.15) shows a scalability limitation and the maximum speedup is achieved at 32 threads (11× for SWITCHES and 9× for *Taskloop*). SWITCHES also seems to be loosing performance as the number of threads increase beyond that point, while OpenMP maintains steady results. The reason for this is that the workload of Poisson2D is not well-balanced by the algorithm, therefore a static implementation such as SWITCHES is bound to loose performance. On the other hand, the dynamic scheduler of OpenMP can handle the load imbalance at runtime and maintain the higher performance. To address the issue of load-balance in SWITCHES explicit allocation of resources is used for the two parallel task-loops of Poisson2D as shown in Section 5.6.3.

5.6.3 Explicit Task Resource Allocation

As explained in Section 5.3.3 one way to solve the low-utilization problem of an application when using a static scheduling runtime system is to explicitly allocate resources for tasks. Figure 5.16 highlights the benefit this technique in combination with the cross-loop dependences and variable granularity introduced by SWITCHES (results in Figure 5.15 also make use of all these techniques). When the granularity is increased to 4 iterations per task and at the same time split the resources (threads)

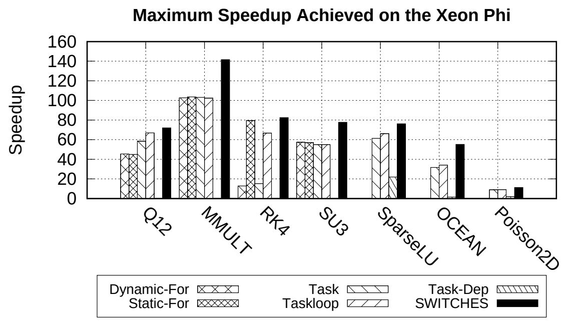


Figure 5.17: Best speedup achieved by each system on the Intel KNC.

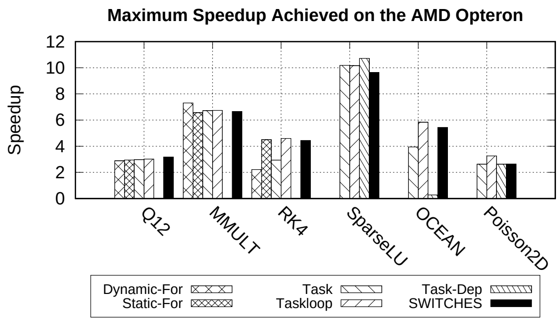


Figure 5.18: Best speedup achieved by each system on the AMD Opteron.

among the two loops of the application, the maximum speedup of Poisson2D is increased by almost 24%. SWITCHES unveils the cross-loop parallelism and the splitting of the resources allows for better allocation of the hardware resources as the two loops execute simultaneously. This technique also helps to use the resources more efficiently during the execution, especially in applications with scaling limitations such as Poisson2D that don't scale beyond 32 threads.

5.6.4 Discussion

Figure 5.17 summarizes all results on the KNC by showing the highest achieved speedup for each application for all OpenMP policies and SWITCHES. Table 5.3 shows the execution times of each of these scenarios. To examine the behavior of SWITCHES on a smaller system the same workloads were executed on an AMD 12-core Opteron processor and present the highest achieved speedups in Figure 5.18. Overall, the results show that SWITCHES is on par with OpenMP for most application on the AMD system. When the applications are scaled to a larger system, SWITCHES surpasses OpenMP for all applications, achieving an average of 32% performance increase compared to the the best OpenMP results. Results for SU3 on the AMD system are not presented as the implementation used had Intel-specific intrinsics for the Xeon Phi processor. Even though it is not shown in these charts, for SparseLU, which is an application from the BOTS benchmark suite [112], SWITCHES was also compared against OmpSs. OmpSs showed performance that is on par with OpenMP on both hardware systems and thus the same conclusions drawn for OpenMP are also valid for OmpSs.

Figures 5.19 and 5.20 present results from a weak scaling analysis of SWITCHES

Table 5.3: The execution time of each scenario that achieves the highest speedup result. The input sizes for all applications are those of previous scenarios except for SparseLU on the AMD which a size of $(96 \cdot 64)$ *was used.*

System	Application	Serial	SWITCHES	System	Application	Serial	SWITCHES
KNC	Q12	*373.35s*	*5.18s*	AMD	Q12	*398.15s*	*125.21s*
	MMULT	*78.17s*	*0.55s*		MMULT	*10.74s*	*1.61s*
	RK4	*32.27s*	*0.34s*		RK4	*5.78s*	*1.30s*
	SU3	*3.59s*	*0.048s*		-	-	-
	SparseLU	*6798.37s*	*84.53s*		SparseLU	*40.12s*	*4.16s*
	OCEAN	*14.57s*	*0.26s*		OCEAN	*3.88s*	*0.71s*
	Poisson2D	*12.14s*	*1.07s*		Poisson2D	*4.36s*	*1.65s*

compared to all other OpenMP implementation described earlier. In a weak scaling scenario the runtime systems is tested as to how it behaves when varying the data input with a fixed number of resources. In these tests the maximum hardware available resources (240 threads) were used for all application except Poisson2D where the number of threads that achieves the highest performance (that is 32 threads) was used. Three different data sizes were used to monitor the behavior. In general for most applications OpenMP closes the performance gap with SWITCHES as the data size increases. This happens because with larger data sizes the work per task increases compared to the work of the runtime system. Consequently, OpenMP manages to hide its runtime overheads within the application computation. Looking at the results from the opposite perspective, reducing the data size should expose the runtime systems operations and make them more prone to overheads. This happens to OpenMP but not to SWITCHES due to its low-overhead runtime implementation. Another important outcome of this study is that for Task-parallel applications with dependences between the executing tasks OpenMP never surpasses the performance of SWITCHES (Figure 5.20) as the dependence resolution mechanism of SWITCHES performs better than that of OpenMP (*Task-Dep*) and at the same time the extra parallelism produced by using task dependences keeps the performance of SWITCHES at high levels compared to the OpenMP scenarios that don't make use of them (*Task* and *Taskloop*).

Taking all results into account, the a static implementation of a Task Data-flow model produces less overheads and has little impact on the parallel execution of the tested application. The de-centralized architecture of SWITCHES allows for

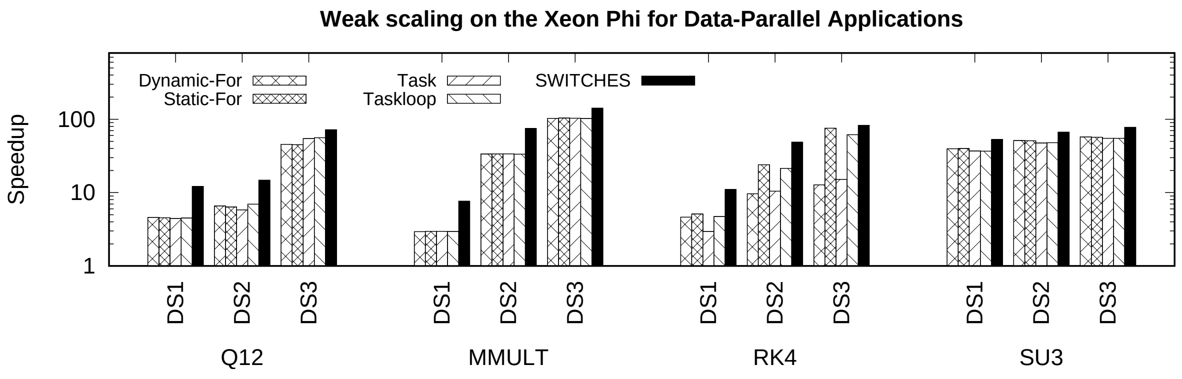


Figure 5.19: Speedup achieved with the best configuration for each system when scaling the input size of the Data-parallel *applications with fixed resources (240 threads).*

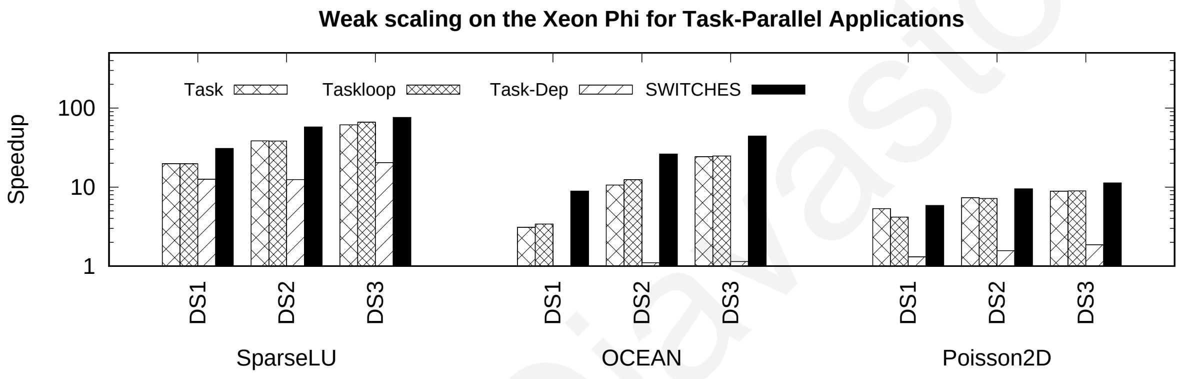


Figure 5.20: Speedup achieved with the best configuration for each system when scaling the input size of the Task-parallel *applications with fixed resources (240 threads for all, except Poisson2D that achieves best speedup results at 32 threads).*

performance scalability regardless of the number of threads, while its lightweight implementation of handling dependences minimizes the runtime overhead of resolving dependences and minimizes the negative effect on the application execution. This is obvious in many of the results as application performance when using OpenMP in many cases stops scaling at 180 threads as the work per task becomes too little to hide the scheduling overheads. On the other hand, for some application performance increases when using SWITCHES even when oversubscribing the system with more software threads than the available hardware resources as it reduces runtime overheads.

Although a static implementation of a runtime system may not be able to efficiently handle applications that dynamically change their load, alternative approaches are shown that can be beneficial. One such approach is the explicit task resource allocation construct that in combination with the cross-loop dependences proposed helps in hiding load-imbalances or low resource utilization.

5.7 Conclusions and Contributions

This chapter introduces SWITCHES [14], a lightweight scalable runtime system that uses the Data-flow paradigm to schedule parallel tasks in many-core systems. In large-scale systems where the number of cores keeps increasing, it is important to improve performance that comes from parallelism. To achieve this, SWITCHES implements a fully distributed scheduler that uses static scheduling policies and produces smaller runtime overheads. Therefore, it allows for finer-grain tasks to be implemented as a means to increase the parallelism exposed without loosing performance.

SWITCHES also incorporates a source-to-source tool that produces parallel code from a sequential application embedded with OpenMP v4.5 API directives. It automatically produces the Data-flow graph with the tasks and their dependences and statically schedules the tasks to the available execution units. Its execution model supports variable-granularity loop tasks that, combined with cross-loop iterations dependences and explicit resource allocation techniques, increases the exploitable parallelism, takes advantage of the data-locality in loop tasks and efficiently allocates hardware resources.

The main contributions of this work are:

- A lightweight, scalable runtime system for task-based Data-flow execution on HPC many-cores, called SWITCHES [14];
- Extensions to the OpenMP v4.5 API to support explicit resource allocation and cross-loop dependences with variable granularity on the `taskloop` directive;
- A source-to-source tool (Translator) that uses the source code with directives to produce parallel pthread code embedded with the runtime, that can be compiled with any commodity compiler;
- Comparison of SWITCHES performance with state-of-the-art OpenMP on a real HPC many-core using task- and data-parallel applications.

SWITCHES is evaluated on a 61-core Intel Xeon Phi, using both task- and data-parallel applications from different benchmark suites. Without affecting programming productivity, SWITCHES achieves significant application performance benefits (an average of 32%), compared to the state-of-the-art OpenMP implementation.

Chapter 6

Autonomic Mapping for Efficient Utilization of Resources

The main objective of the work described in this chapter is to efficiently map the tasks to the underlying hardware topology in an automated way, using application characteristics such as the dependences between tasks. Currently, to achieve this, each application must be studied exhaustively as to define the usage of the data by the different tasks, that would provide the knowledge for mapping tasks that share the same data close to each other. In addition, different hardware topologies require different mappings for the same application to produce the best performance. In this work the synchronization graph of a task-based parallel application that is produced during compilation is used in order to try and automatically tune the scheduling policy on top of any underlying hardware using machine learning techniques. This tool is integrated into an actual task-based parallel programming platform - SWITCHES (see Chapter 5) - and is evaluated using real applications from the SWITCHES benchmark suite. Results are compared with the execution time of predefined schedules within SWITCHES and observe that the tool can converge close to an optimal solution with no effort from the user and using fewer resources.

6.1 Motivation

Designing a parallel program has its degree of difficulty. Several programming models have been proposed to alleviate this problem. The current trend that seems to be more appropriate to exploit large degree of parallelism in an application is to specify the program as a set of tasks which may also be related with each other by

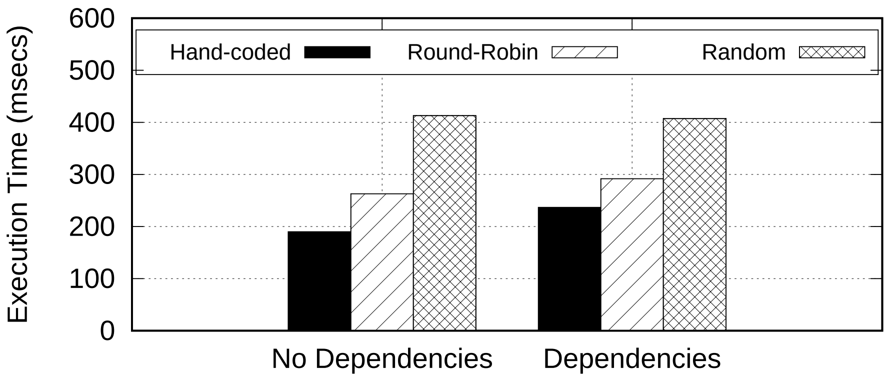


Figure 6.1: Execution time of Round-Robin *and* Random *policies compared to hand-coding an optimum schedule for synthetic kernels implemented in SWITCHES. This test was performed on an Intel Xeon Phi using 240 threads.*

data-flow dependences. Many runtime and programming systems today support the task-based model of execution with the most widely known system to be the latest release of OpenMP v4.5 [90]. Analyzing the original code and generating these tasks and dependences is already a difficult task. Nevertheless, as systems increase their scale, it is not only the number of processing elements that increase but also the heterogeneity of the system as a whole. Thus, the allocation of tasks to resources becomes a huge challenge. The challenge is not only to address runtime dynamic behavior of the tasks but also to determine a schedule of the tasks that results in the best execution time, as for example, a task that produces data that is consumed by another task should be co-located in the same resource or placed nearby as to avoid or reduce the data transfer overhead. Given the complexity of the underlying infrastructure and of the synchronization graph representing an application, this mapping is a difficult problem to solve.

Figure 6.1 shows the performance of *Round-Robin* and *Random* schedules for two synthetic task-based kernels (one without dependences and one with dependences between a number of tasks), compared to hand-coding an optimum schedule. Hand-coding an efficient schedule of any application requires a highly experienced programmer with significant knowledge of the application characteristics (tasks, dependences and data usage) and the underlying hardware. Results show that in such a scenario, execution time can be reduced by as much as 2*x*. Nevertheless, the complexity and the variety of both applications and hardware systems increases with time and make the *Hand-coded* scenario an unfeasible solution.

Therefore, for future complex applications and large-scale parallel systems, through this work the use of a machine-learning approach is proposed as to determine a task-to-resource mapping, that will take into account both the application and the hardware characteristics. This work uses a Genetic Algorithm (GA) where the population is composed of different task-to-resource assignment schedules and the generations evolution is guided by the evaluation metrics (e.g. execution time or energy consumption) as to converge to the best schedule, depending on the metric defined by the user. The GA is integrated with an actual parallel programming and runtime system, thus the iteration over the different steps of the optimization such as creation of new schedules, execution and evaluation, are executed in an autonomic way, on real applications. The final schedule is determined automatically by the process, but this is not without cost as it requires the execution of the application several times. In this work the initial population is chosen carefully so that the GA converges faster towards the better schedule. This optimization step is done either ahead of the execution of the real application, in an initialization/setup phase of the system and application tuning, or as part of a scenario where the same application is to be executed multiple times on the same system and each time it executes its metrics results are stored and used by the GA to improve the schedule for the future runs.

6.2 Optimizing Task Scheduling

The task scheduling problem is a well studied field and appears numerous times in the literature. Background search is focused mostly in GAs or even the more general heuristic-based solutions for task scheduling problems and shows a large number of algorithms proposed to solve different scheduling problems [116–124]. All these algorithms are implemented and tested on multi-processor systems. Some are targeting heterogeneous systems with computation units with variable capabilities and other were proposed for homogeneous parallel systems. What all these have in common though is that they are all tested using randomly generated task graphs and are implemented and evaluated as simulations. In contrast, this work implements a GA within a parallel programming and runtime system that allows for using it with real applications and producing schedules for real hardware systems.

6.2.1 The NSGA-II Genetic Algorithm

A Genetic Algorithm is a heuristic procedure that tries to find the optimal solution to a presented problem. The main principle of a GA is that crossing two individuals can result an offspring that is better than both the parents. Also, a slight mutation of the produced offspring can generate better individuals. The crossover mating takes two individuals of a population as inputs and generates two new offsprings. This way some of the parent characteristics are maintained in individual offsprings in new generations. The mutation randomly transforms an offspring that was also randomly chosen from the set of all new offsprings produced by the crossover process. Finally, the best solutions are selected using a fitness function and transferred as inputs to the next generation and the next crossover mating. A fitness function is defined upon the problem that the GA is trying to solve and the best individual corresponds to the one having the best fitness value. For example, in most scenarios tested in this work the fitness function detects the smaller execution time therefore, the best individual corresponds to the one with the smallest execution time.

A GA is a loop that starts with an initial population and evolves through generations using a selection followed by a sequence of crossovers and a sequence of low-probability mutations. The loop can terminate either by a limit on the total number of iterations or the stability of the results defined by the fitness evaluation function.

To optimize the scheduling of the tested applications in this work, an already existing and well known multi-objective genetic algorithm was used, the Non-dominated Sorting Genetic Algorithm II (NSGA-II) [125]. The NSGA-II was proposed in order to address the main disadvantages of the previous NSGA algorithm proposed in [126]. The original NSGA algorithm suffered from high computational complexity ($O(MN^3)$), lack of elitism and the need for specifying the shared parameter. The NSGA-II alleviates the above disadvantages and presents a solution with a fast non-dominated sorting approach with $O(MN^2)$ computational complexity, where M is the number of independent sortings to be executed and N is the size of the population.

The NSGA-II is using a selection operator that creates a mating pool by combining the parent and offspring populations and selecting the best (with respect to the fitness function) N solutions. Being a multi-objective algorithm the NSGA-II can also use

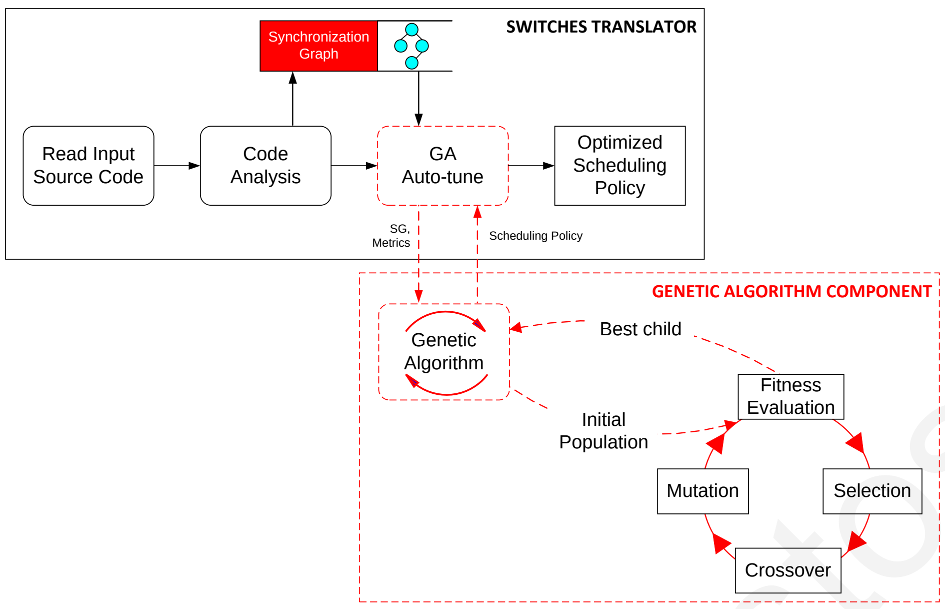


Figure 6.2: The design of the auto-tuning tool and how it is integrated in the SWITCHES Translator.

more than one parameter to its fitness function. It can combine the values of 2 or more variables as to select the best individuals within a population. This can be particularly useful in scheduling problems where more than one parameters might be important in the execution of a task-based application (e.g. power-performance efficiency).

Details of the NSGA-II implementation [125, 127] are omitted from this work as it is used unmodified.

6.3 Auto-tuning Static Scheduling

To create the auto-tuning scheduling tool, the NSGA-II algorithm was integrated inside the SWITCHES *Translator* (see Section 5.4). Figure 6.2 shows the design of this integration and how the GA works together with the *Translator* to produce an optimized schedule for any input application. The directive-based source code of the application is given to the *Translator* as it would normally happen for a SWITCHES program. The *Translator* then analyzes the code and produces its SG. The SG is then passed to the Genetic Algorithm Component (GAC), that implements the NSGA-II. The GAC also needs some parameters that are required for the GA execution. These parameters are: (i) the number of generations to execute the algorithm, (ii) the size of the population for each generation, (iii) the objectives that the fitness function will use to evaluate each schedule and (iv) the mutation and crossover probabilities.

The number of generations and the size of each population can be explicitly defined by the user. The values are based on the cost tolerance accepted by the user for a specific application. The larger the size of a population or the more generations requested, the more time it will take for the auto-tuning tool to finish and produce a schedule that is optimal. The fitness objectives are used by the GAC to evaluate each produced schedule and rank it in the current population. The objectives currently supported by the tool are performance (execution time), power consumption and processor temperature. The tool ranks the produced schedules in a population based on the objective chosen by the user. As mentioned in Section 6.2.1, the NSGA-II is a multi-objective algorithm that allows using multiple objectives to decide the classification of the fitness evaluation. Therefore, the user can chose more than one objective to be considered for the evaluation fitness of the population. Finally, the mutation and crossover probabilities are usually decided empirically as they are affected by the problem that is to be solved.

When the GAC receives the SG and the GA parameters, it produces an initial population with random schedules. As an optimization the default SWITCHES [14] schedules are included in the initial population. The GAC then starts executing the application with each schedule in the population and stores the evaluation results when the execution is finished (*fitness evaluation*). At the end of each generation, each schedule is ranked based on the objective/s requested. If, for example, the objective is performance, the schedules are ranked in ascending order, starting with the schedule that produces the smallest execution time. At the *Selection* stage, the GAC implements a tournament selection process [128] that uses this ranking to decide which schedules in the population should be used for the *Crossover*, that are then used to create the population of the next generation. When the new generation is created, a *Mutation* function is applied to the population that alters one or more values of a schedule. The mutation process is useful to maintain the genetic diversity from one generation to the next. Based on the mutation probability, it is likely that one or more schedules don't change from one generation to the next.

When the GAC execution reaches the limit of generations defined by the user, the best ranked policy is identified and passed back to the *Translator* to produce the parallel source code. This schedule is also stored in a text file, that can be loaded by the *Translator* at a later time.

Table 6.1: *Experimental workloads description and data set sizes.*

Benchmark	Description	Complexity
MMULT	Matrix multiply [18]	$1024 \cdot 1024$
RK4	Differential equation [18]	19200
Poisson2D	5-point 2D stencil computation [109]	$16384 \cdot 128$
No-Dependences	Random generated tasks	$1024 \cdot 100$
Dependences	Random generated tasks with dependences	$1024 \cdot 100$

6.4 Experimental Results

6.4.1 Experimental Setup

This auto-tuning tool is evaluated on a set of 3 data- and task-parallel real applications from the SWITCHES evaluation suite 5.5. Two computation kernels with randomly generated task graphs are also tested (*No-Dependences* and *Dependences*). The former allows fully parallel execution of the tasks without imposing any kind of dependences between them, while the latter represents a synthetic kernel with randomly generated dependences between tasks. Details for all the application and their input sizes are shown in Table 6.1. The complexity column represents the total number of computation iterations each application executes on its data.

The evaluation platform of this work is an Intel Xeon Phi 7120P with 61 cores and 4 threads per core (totaling 244 hardware threads). Note that only 60 cores were used as to avoid any interference with the OS that always uses the last core of the system. This board has a total of 16GB of main memory and runs at 1.238GHz. To cross-compile applications for the Xeon Phi the Intel icc v.17.0.2 compiler (and the corresponding libiomp5 library) is used with the *-mmic* flag indicating the Many Integrated Architecture (MIC) target and the *-O3* optimization flag. The results are presented as execution time that is collected using the `gettimeofday` system call that provides a resolution of microsecond. The time is measured from the start of the first parallel function until the last, including all runtime costs (such as thread creation and scheduler initialization).

The parameters used for the GA algorithm are 10 generations for the real applications tested and 50 for the synthetic kernels. Each generation has a population of 64 schedules. The mutation and crossover probabilities are 0.0001 and 0.6 respectively.

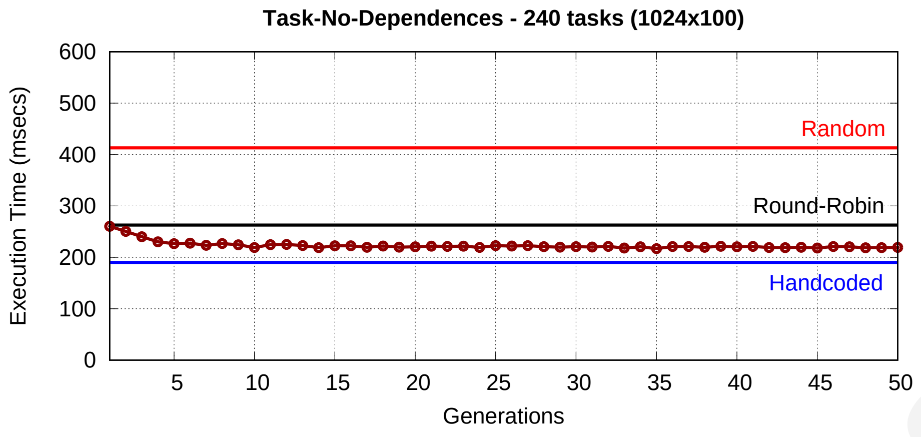


Figure 6.3: The execution times for the No-Dependences *synthetic kernel. The dotted line shows the results obtained by the auto-tuning tool for 50 generations. The produced schedule converges within 15% of the optimal scenario, while it uses 30% less computational resources.*

These values were chosen based on literature study and small scale testing scenarios with various other values that show that beyond these values, no significant difference is observed. It is important to notice that different problems require different values, therefore for new applications it is important to test various options.

6.4.2 Synthetic Applications

The synthetic kernels were evaluated using two SWITCHES predefined policies, *Round-Robin* that assigns equal number of tasks to all threads in a round-robin way and *Random* that assigns the tasks to the threads in random way. The synthetic kernels were also analyzed to find the data usage of the tasks in both of them and the task dependences in the second kernel. This information was used to create a *Handcoded* scheduling policy that was deemed as the optimal scenario. Results of these executions are shown in Figures 6.3 and 6.4. Running the synthetic kernels through the auto-tuning tool shows that it can converge close to the optimal scenario (within 15% for the *No-Dependences* and 10% for the *Dependences* scenarios) and in both cases achieves better performance than the original predefined SWITCHES policies. Results showed that the kernel with the task dependences requires more generations of the algorithm and this happens due to the dependences between the tasks and complexity of its SG.

In addition, an important outcome of these scenarios is that the schedule produced by the auto-tuning tool that is near the optimal *Handcoded* schedule is achieved

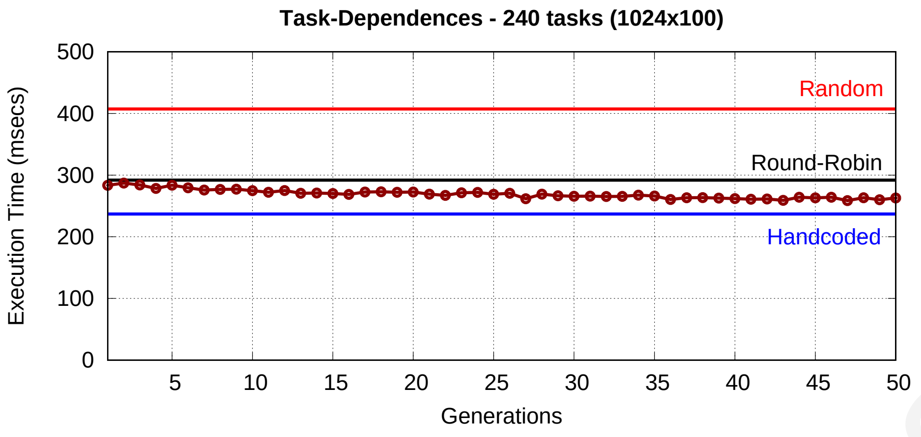


Figure 6.4: The execution times for the Dependences *synthetic kernel. The dotted line shows the results obtained by the auto-tuning tool for 50 generations. The produced schedule converges within 10% of the optimal scenario, while it uses 30% less computational resources.*

by using 30% less computing resources. Scenarios *Random*, *Round-Robin* and *Hand-coded* are using all 240 hardware threads of the system, while the schedule of the auto-tuning tool is using 168 hardware threads for the *No-Dependences* and 166 hardware threads for the *Dependences*.

6.4.3 Data-Parallel Applications

Two data-parallel application from the SWITCHES benchmark suite were used for this scenario, MMULT and RK4. The results are shown in Figures 6.5 and 6.6 respectively. The data set of both applications is equally divided to all tasks with most of the tasks using their own data. In the cases that tasks share data, they are shared in a consecutive way. That is, consecutive tasks share data from consecutive memory locations. Therefore, the best policy is to assign tasks in a consecutive way. This is exactly what the *Round-Robin* policy does and this is why there is no extra benefit achieved from the auto-tuning tool. Finally, in contrast to the synthetic kernels presented earlier, for these data-parallel applications, the auto-tuning tool chooses to use all hardware resources available to achieve the performance shown.

6.4.4 Task-Parallel Applications

This scenario uses the Poisson2D, a task-parallel application from the SWITCHES benchmark suite. This application achieves its highest speedup with the default SWITCHES schedule at 32 hardware threads (on an Intel Xeon Phi system). It

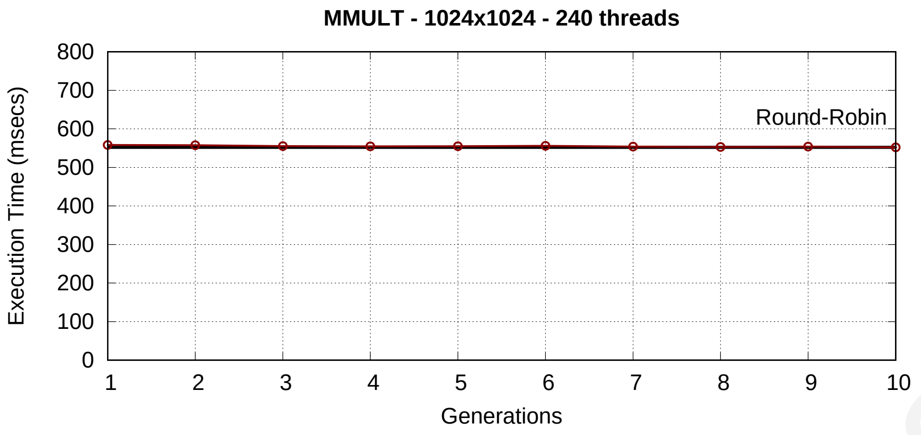


Figure 6.5: The performance achieved by the auto-tuning tool for MMULT is the same as what is already achieved by the Round-Robin *policy of SWITCHES.*

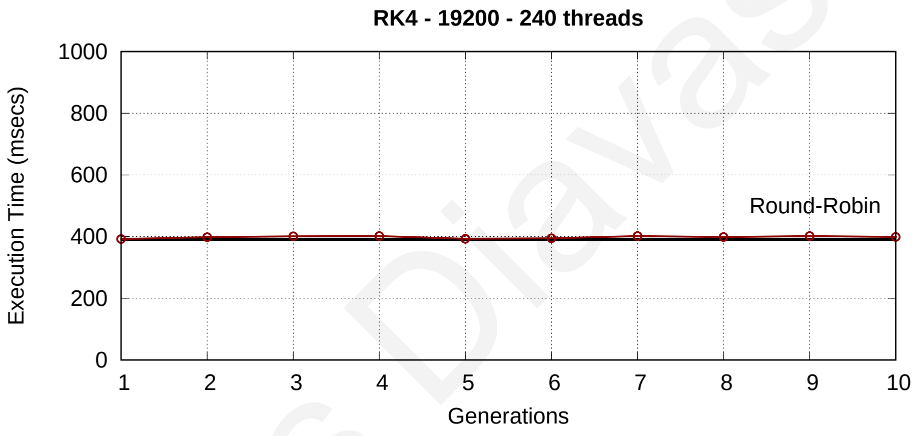


Figure 6.6: The performance achieved by the auto-tuning tool for RK4 is the same as what is already achieved by the Round-Robin *policy of SWITCHES.*

also shows significant performance loss for any number of threads greater than 32. Figure 6.7 shows results when the auto-tuning tool runs for only 32 hardware threads and observe a small performance improvement compared to the default SWITCHES schedule. Studying the policy produced by the auto-tuning tool, it chooses to use hardware threads that belong to the same core, while the *Round-Robin* policy in this case is using separate cores for each of the 32 hardware threads used. The auto-tuning tool chooses to place depended tasks that share the same data on the same cores and thus minimizes the data transfer overhead.

As explained earlier, using all the hardware resources for Poisson2D with SWITCHES results in decreasing performance due to algorithmic limitations of the application and inefficient assignment of the tasks by the SWITCHES default policy. Figure 6.8 shows the results of the auto-tuning tool for a Poisson2D execution using 240 threads.

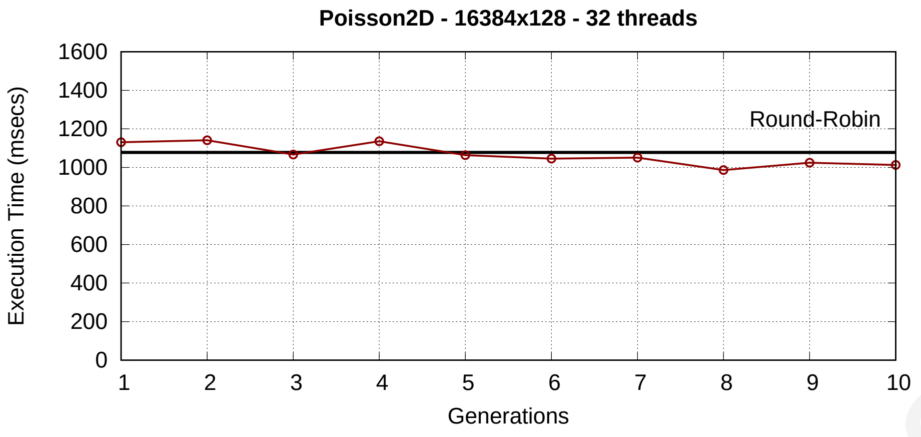


Figure 6.7: The auto-tuning tools slightly reduces the execution time by using hardware threads from the same cores, in contrast to the Round-Robin *policy that uses different cores.*

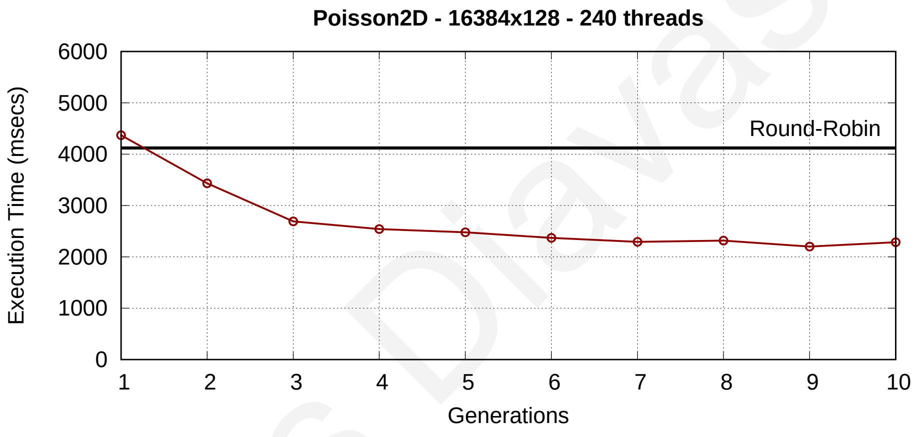


Figure 6.8: The execution time of Poisson2D when using all available resources is reduce by 2× *when using the auto-tuning tool. The tool also achieves this by using only 70% of the total hardware resources.*

The auto-tuning tool finds a schedule that significantly increases the performance and achieves a 2× improvement compared to the default *Round-Robin* policy. Similar to the case of the synthetic kernels presented earlier, this results is achieved by using 30% less hardware resources.

6.4.5 Seed Optimization

As explained in Section 6.3, in its initial population the auto-tuning tool includes the default schedules that SWITCHES can produce. Using the default schedules as a starting point for the auto-tuning tool reduces the size of the search space of the GAC and improves the convergence speed to an optimal solution. This is especially important in many-core systems where the search space of the auto-tuning tool

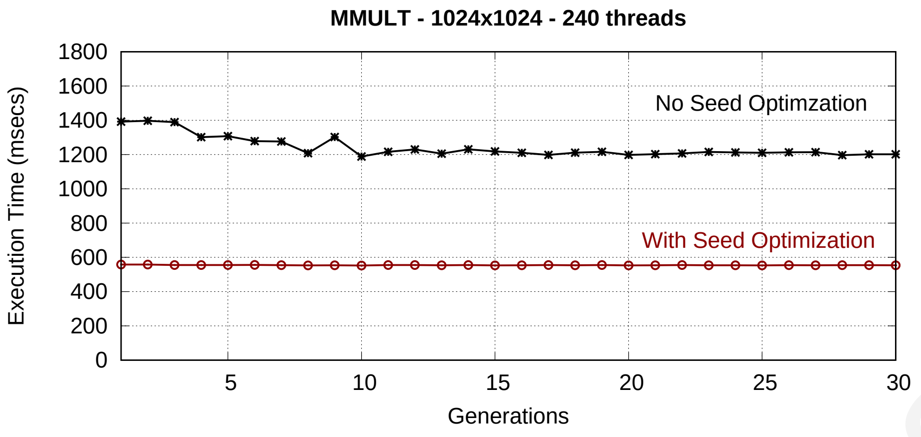


Figure 6.9: The auto-tuning tools performs better when SWITCHES predefined assignment policies are included in its initial population.

(*i.e.* the number of resources) will be very large, as in such a case the number of combinations of resources that create a possible schedule will also be large. Figure 6.9 shows results of this optimization tested with MMULT.

6.4.6 Discussion

The results show significant performance benefit especially in the cases of task-based applications. Irregularities in the execution, the dependences and the data accesses require more effort from the programmer in order to produce the most performance-efficient scheduling policy. A rigorous and detailed analysis of each application is required by the user in order gather all these information and produce the fastest scheduling policy. The aim of this work is to automate this procedure and relief the programmer from this task. Currently, the auto-tuning tool requires for the application to be executed multiple times with different scheduling policies as input in order to decide on the best policy for a specific application. But, in HPC (that is the target of this work) the same applications are executed many times to study different parameters of their output. The execution though, remains the same. Therefore, every time an application is executed, we could use a different scheduling policy and store the performance outputs for every execution. These outputs can then be loaded to the auto-tuning tool for training in order to provide a faster schedule.

We also studied the case of training the auto-tuning tool with a small input size and then use the output schedule to execute the application for a normal input size.

This study showed that applications with data irregularities will not benefit as data accesses are not taken into account. To speedup this process and provide useful information to the user community, a study of application characterization would help create categories of applications with similar characteristics that react similarly to the same schedules as well.

6.5 Conclusions and Contributions

The auto-tuning tool is designed and implemented to optimize schedules for execution time, power consumption and temperature. Because the NSGA-II algorithm used is a multi-objective GA, the tool also supports the optimization of schedules for any combination of the above metrics. The GA (NSGA-II) is integrated within the SWITCHES programming tool (the *Translator*) and because SWITCHES runtime uses static schedules it allows feeding the *Translator* with any policy during compilation. This allowed the implementation of a GA that produces schedules that are fed to the *Translator* and evaluated at the same time. This integration, allows the sharing of the application synchronization graph with the auto-tuning tool to use it to extract information about the tasks, their dependences and their data usage. The SWITCHES runtime system uses a static scheduler and avoids any interference in the execution of other runtime overheads as to isolate the auto-tuning process for better understanding of the results.

The main contributions of this work are:

- The integration of a well known GA (NSGA-II [126]) in a real parallel programming and runtime task-based system (SWITCHES [14]) that offers an auto-tuning scheduling tool for parallel applications;
- Achieve maximum performance with fewer resources for applications with complex dependences and irregular data accesses without any significant effort from the programmer.

Results on the 61-core Intel Xeon Phi show that maximum performance can be achieved by using significantly fewer resources (approx. 30%) than what is available and what the default scheduling policies use. This improves the efficiency of the execution as higher performance with fewer resources can result in a reduction of the power consumption.

Chapter 7

Conclusions and Future Work

Improving the performance of parallel applications is a joint task that involves the hardware and the software. From the hardware design perspective, scalable architectures with large number of cores increase the throughput of an application, thus more small and simple cores must be added in future processors. From the software design perspective, lightweight, low-overhead and scalable runtime systems increase parallelism exploitation. Combine the two and application performance can be increased and scaled on large-scale many-core systems. At the same time, power-performance efficiency is achieved by efficiently utilizing the available resources while minimizing the communication overhead with locality-aware scheduling. Finally, programming productivity is increased with powerful tools that provide a simplified way for parallel application development. The work presented in this thesis uses the task-based Data-flow model as a way to extract application parallelism and proposes a complete software framework for parallel application development and execution on HPC many-core processors. Each of the objectives addressed in this thesis is evaluated using a set of application that show the benefit of each studied characteristic.

7.1 Achieved Objectives and Contributions

To exploit more application parallelism (Objective 1 - Section 1.4.2) TM is integrated in the Data-flow model and a way to exploit speculative parallelism. The DDM+TM system [10, 11] is the first implementation of a Data-flow-based runtime that introduces shared mutable data and uses TM to monitor transactional data and abort conflicting accesses on shared variables. Results show good opportunities for performance when using the integration of the two models, while at the same time more parallelism is exposed to Data-flow, that would otherwise be a set of serialized computations. Although performance improvement is achieved in a small-scale multi-core, the runtime overheads analysis shows that a software implementation of a TM runtime system can't scale sufficiently. Therefore, integrating TM into the Data-flow model shows significant potential but is still not efficient enough to be used in large-scale many-core systems.

To increase programming productivity (Objective 2 - Section 1.4.3), all DDM runtime systems to date have been incorporated under a common programming framework [12, 13]. To provide for a user-friendly programming environment, the most commonly used API, the OpenMP v4.5, was extended to support explicit task resource allocation mechanisms and variable loop task granularity to increase data-locality even for loop tasks with inter-dependences. A source-to-source tool is implemented that automatically produces thread-based code that can be compiled by any off-the-shelf C/C++ compiler, applying all existing optimizations [14].

To overcome the limitations of scalable architectures (Objective 3 - Section 1.4.4), the TFluxSCC system [16] is proposed in this work as the first implementation of the DDM model on a many-core processor. A set of different applications was used to test the scalability of their performance and good speedup is observed for most applications. The introduced implementation proves that a software runtime of a Data-flow-based model can produce significant performance benefits on future generations of many-core processors. While a considerable speedup was achieved for certain applications, porting the TFlux platform on the Intel SCC, revealed various issues that can potentially harm the scalability of the runtime on more complicated architectures. Issues such as size of the runtime system and its data structures, communication between scheduling units loaded on each core and the number of messages exchanged can potentially become a bottleneck as we scale to more cores.

These findings led to the design of a runtime system that is lightweight, low-overhead and scalable to any number of cores, with little support from the hardware (Objective 4 - Section 1.4.5). In this thesis a software framework called SWITCHES [14] is developed for exploiting large amounts of parallelism on many-core processors supporting global address space without needed hardware support for cache-coherence. The evaluation of the proposed system was conducted using

HPC-based application on a real HPC-based many-core (the 61-core Intel Xeon Phi). Results show good scalability and surpass the state-of-the-art by an average of 32% for all applications. To reduce the underutilization of hardware resources (Objective 5 - Section 1.4.6) this work employs machine-learning techniques. The proposed framework is extended to use the synchronization graph of the application and automatically tune the scheduling policy using machine learning techniques. Results show that maximum performance is achieved by using significantly less resources ($\approx$ 30%) than what common scheduling policies use. This leads to better hardware utilization and reduced energy consumption.

7.2 Open Research Directions

The results of this thesis show that if algorithms exhibit enough parallelism and architecture designs large number of hardware resources, it is possible to offer tools to the users that will provide increasing performance in a user-friendly way. At the same time, new research directions arise from the outcomes of this work.

7.2.1 Direction 1: Speculative Parallelism on Many-cores

The study on integration of speculative parallelism inside the Data-flow model (Chapter 3) has shown the success of integrating two different models as a way to increase application parallelism exploitation. Results also show that current TM software implementations produce significant runtime overheads that increase with the complexity of the application and the number of resources used. But newly introduced many-core processors offer large number of resources, that allow for the exploitation of more parallelism, therefore potential performance benefits are lost. Solutions such as hardware TM implementations (e.g. Intel's Transactional Synchronization Extensions (TSX) [129]) have appeared but not widely adopted due to the complexity of the hardware. New low-overhead TM implementations that provide speculative execution must be developed in order to extract more parallelism and in combination with a Data-flow runtime system to achieve maximum exploitation of parallelism in an application. The large number of cores that will be provided in future many-core processors will allow to sacrifice a small number of them for handling costly TM runtime operations such as transactional data monitoring.

7.2.2 Direction 2: Inter-node Scalability using Data-flow

This thesis addresses the scalability of application performance on many-cores with large number of cores integrated in a single processor. Since the target of this work is the HPC community it is essential to study how the proposed system can be integrated in large clusters with multiple many-cores connected together. Message-passing systems are already very efficient is such systems but the definition and implementation of senders and receivers as well as the messages to be exchanged fall into the ability of the programmer. One direction to follow is the integration of the Data-flow implementation we propose in this thesis within a message-passing system. The Data-flow runtime system can be used for executing tasks on a single node, while the message-passing system can be used for expanding the execution to the entire cluster. The data dependences already provided by Data-flow can be used as a way to identify the messages to be exchanged with other nodes taking part in the execution. This allows the exploitation of the benefits of the Data-flow model and takes advantage of the very efficient implementations of message-passing systems that exist today.

7.2.3 Direction 3: Dynamic Rescheduling of Static tasks

In this thesis we studied ways of providing better scheduling and assignment policies in order to efficiently utilize the underlying hardware. We employed machine-learning techniques to train the system and automatically produce optimal policies based on characteristics from both the hardware and the applications. Such techniques though are application specific and require time. This time may not be always available, depending on the problem the application is trying to solve. A combination of a dynamic load-balancer and a machine-learning algorithm can provide a resource manager that can dynamically adapt to any workload irregularities during the execution of the application.

7.2.4 Direction 4: Heterogeneous Many-cores

In this thesis we concentrated on increasing parallelism exploitation and scaling application performance on many-core processors. With only requiring global address space for application data, the architecture design was assumed to be homogeneous across the entire processor. It is possible that in the future many-core processors will include cores of different characteristics as to address the heterogeneity of the workloads. Large-scale systems with multiple processors also use heterogeneous designs with the use of accelerators with different architecture that the host machine (e.g. GPUs). The SWITCHES runtime system is designed in such a way that it distributed the scheduling information to all participating resources. Each task holds its own scheduling data structures, therefore it can execute anywhere it is appointed to regardless of the design of the hardware. The programming model though needs be adapted to new architectures and the framework to be extended t support heterogeneous systems.

7.2.5 Direction 5: Extension of the programming tool-chain

The programming tools developed in this thesis are used to recognize only parallel constructs (OpenMP v4.5). The input source code is parsed and whenever parallel constructs are recognized, the tools (*i.e.* the Translator) will read-in the information needed and transform the construct into parallel pthread-based source code. This procedure is done at a preprocessing stage before compiling the code to produce the binary. There are tools though that could be used to support the SWITCHES runtime system along with its API such as the LLVM Compiler Infrastructure [130] in a more complete infrastructure, that will eventually speedup the process of producing SWITCHES-based parallel software. At the same time, such tools can provide more information to the programming platform of SWITCHES that can be used when developing the Data-flow graph of an application and when implementing its scheduling policy.

7.2.6 Direction 6: Fault-tolerance

One other direction that this work could be extended to is fault-tolerance and how the runtime system would react in case of failures in the hardware. It is possible that many-core processors will have failures and cores could be disabled. In a case this happens during the execution of an application, a mechanism in the runtime system that would be able to react accordingly is necessary. In case of a failed core, the runtime system must be able to reschedule tasks that were assigned to that core to a different available core, or even equally distribute them to all other cores. SWITCHES is built to be fully distributed, thus from the runtime perspective a rescheduling of a software thread to a different core is possible. From the task perspective, each SWITCHES task is built in such a way that it holds all the information needed to be executed. Also, some techniques from the TM runtime support can be used to suppress changes made to the data of the task in case there is a core failure. Therefore, if a task is rescheduled to a different core it will have all the information needed to execute correctly. To support fault-tolerance within SWITCHES a dynamic rescheduling mechanism is needed (see Direction 3 in Section 7.2.3) that would also take into account the cases of hardware failures.

Bibliography

[1] A. Diavastos, "SWITCHES Platform," https://github.com/diavastos/SWITCHES, 2017, [Online].

[2] M. M. Resch, "The end of Moore's law: Moving on in HPC beyond current technology," in *Keynote Presentation in PDP2016 Conference*, Heraklion, Crete, February 2016.

[3] Q. Huang, Z. Huang, P. Werstein, and M. Purvis, "GPU as a General Purpose Computing Resource," in *9th International Conference on Parallel and Distributed Computing, Applications and Technologies*, Dec 2008, pp. 151–158.

[4] Intel MIC, "Intel Many Integrated Core Architecture," http://www.intel.com/content/www/us/en/architecture-and-technology/many-integrated-core, 2016, [Online].

[5] M. Snir, "Programming models for high-performance computing," in *2013 13th IEEE/ACM International Symposium on Cluster, Cloud, and Grid Computing*, May 2013, pp. 1–1.

[6] X. Zhou, H. Chen, S. Luo, Y. Gao, S. Yan, W. Liu, B. Lewis, and B. Saha, "A case for software managed coherence in manycore processors," in *the 2nd USENIX Workshop on Hot Topics in Parallelism (Poster Paper)*, 2010.

[7] B. Choi, R. Komuravelli, H. Sung, R. Smolinski, N. Honarmand, S. V. Adve, V. S. Adve, N. P. Carter, and C. T. Chou, "DeNovo: Rethinking the Memory Hierarchy for Disciplined Parallelism," in *2011 International Conference on Parallel Architectures and Compilation Techniques*, Oct 2011, pp. 155–166.

[8] J. H. Kelm, D. R. Johnson, W. Tuohy, S. S. Lumetta, and S. J. Patel, "Cohesion: A Hybrid Memory Model for Accelerators," in *Proceedings of the 37th Annual International Symposium on Computer Architecture*, ser. ISCA '10. New York, NY, USA: ACM, 2010, pp. 429–440.

[9] C. Kyriacou, P. Evripidou, and P. Trancoso, "Data-Driven Multithreading Using Conventional Microprocessors," *IEEE Transactions on Parallel and Distributed Systems*, vol. 17, no. 10, pp. 1176–1188, 2006.

[10] A. Diavastos, P. Trancoso, M. Luj´n, and I. Watson, "Integrating Transactions into the Data-Driven Multi-threading Model Using the TFlux Platform," in *2011 First Workshop on Data-Flow Execution Models for Extreme Scale Computing*, Oct 2011, pp. 19–27.

[11] A. Diavastos, P. Trancoso, M. Luján, and I. Watson, "Integrating Transactions into the Data-Driven Multi-threading Model Using the TFlux Platform," *International Journal of Parallel Programming*, vol. 44, no. 2, pp. 257–277, Apr 2016.

[12] A. Diavastos and P. Trancoso, "Unified Data-Flow Platform for General Purpose Many-core Systems," *Technical Report UCY-CS-TR-17-2*, pp. 1–22, September 2017.

[13] P. E. Andreas Diavastos, George Matheou and P. Trancoso, "Data-Driven Multithreading Programming Tool-chain," *Technical Report UCY-CS-TR-17-3*, pp. 1–18, September 2017.

[14] A. Diavastos and P. Trancoso, "SWITCHES: A Lightweight Runtime for Dataflow Execution of Tasks on Many-Cores," *ACM Trans. Archit. Code Optim*, vol. 14, no. 3, Aug. 2017.

[15] T. G. Mattson, M. Riepen, T. Lehnig, P. Brett, W. Haas, P. Kennedy, J. Howard, S. Vangal, N. Borkar, G. Ruhl *et al.*, "The 48-core SCC processor: the programmer's view," in *Proceedings of the ACM/IEEE SC '10*, 2010, pp. 1–11.

[16] A. Diavastos, G. Stylianou, and P. Trancoso, "TFluxSCC: Exploiting Performance on Future Many-Core Systems through Data-Flow," in *2015 23rd Euromicro International Conference on Parallel, Distributed, and Network-Based Processing*, March 2015, pp. 190–198.

[17] A. Diavastos and P. Trancoso, "Auto-tuning Static Schedules for Task Dataflow Applications," in *Proceedings of the 1st ACM Workshop on AutotuniNg and aDaptivity AppRoaches for Energy efficient HPC Systems, (ANDARE)*, Portland, Oregon, USA, September 2017.

[18] K. Stavrou, M. Nikolaides, D. Pavlou, S. Arandi, P. Evripidou, and P. Trancoso, "TFlux: A Portable Platform for Data-Driven Multithreading on Commodity Multicore Systems," in *37th International Conference on Parallel Processing*, 2008, pp. 25–34.

[19] G. Matheou and P. Evripidou, "Architectural Support for Data-Driven Execution," *ACM Trans. Archit. Code Optim.*, vol. 11, no. 4, pp. 52:1–52:25, Jan. 2015.

[20] J. B. Dennis, "First version of a data flow procedure language," in *Programming Symposium*. Springer, 1974, pp. 362–376.

[21] J. B. Dennis and D. P. Misunas, "A Preliminary Architecture for a Basic Dataflow Processor," in *Proceedings of the 2nd Annual Symposium on Computer Architecture*, ser. ISCA '75. New York, NY, USA: ACM, 1975, pp. 126–132.

[22] A. Duran, E. Ayguadé, R. M. Badia, J. Labarta, L. Martinell, X. Martorell, and J. Planas, "OmpSs: A Proposal For Programming Heterogeneous Multi-Core Architectures," *Parallel Processing Letters*, vol. 21, no. 02, pp. 173–193, 2011.

[23] A. Kukanov and M. J. Voss, "The Foundations for Scalable Multi-core Software in Intel Threading Building Blocks." *Intel Technology Journal*, vol. 11, no. 4, pp. 309 – 322, 2007.

[24] C. Lauderdale, M. Glines, J. Zhao, A. Spiotta, and R. Khan, "SWARM: A unified framework for parallel-for, task dataflow, and distributed graph traversal," *ET International Inc., Newark, USA*, 2013.

[25] R. Giorgi, R. M. Badia, F. Bodin, A. Cohen, P. Evripidou, P. Faraboschi, B. Fechner, G. R. Gao, A. Garbade, R. Gayatri, S. Girbal, D. Goodman, B. Khan, S. Koliaï, J. Landwehr, N. M. Lê, F. Li, M. Lujàn, A. Mendelson, L. Morin, N. Navarro, T. Patejko, A. Pop, P. Trancoso, T. Ungerer, I. Watson, S. Weis,

S. Zuckerman, and M. Valero, “TERAFLUX: Harnessing dataflow in next generation teradevices,” *Microprocessors and Microsystems*, vol. 38, no. 8, Part B, pp. 976 – 990, 2014.

[26] E. A. Lee and D. G. Messerschmitt, “Synchronous data flow,” *Proceedings of the IEEE*, vol. 75, no. 9, pp. 1235–1245, Sept 1987.

[27] J. R. Gurd, C. C. Kirkham, and I. Watson, “The Manchester Prototype Dataflow Computer,” *Commun. ACM*, vol. 28, no. 1, pp. 34–52, Jan. 1985.

[28] G. M. Papadopoulos and D. E. Culler, “Monsoon: An Explicit Token-store Architecture,” in *Proceedings of the 17th Annual International Symposium on Computer Architecture*, ser. ISCA ’90. New York, NY, USA: ACM, 1990, pp. 82–91.

[29] D. Cann, “Retire Fortran? A Debate Rekindled,” in *Proceedings of the 1991 ACM/IEEE Conference on Supercomputing*, ser. Supercomputing ’91. New York, NY, USA: ACM, 1991, pp. 264–272.

[30] I. Watson, V. Woods, P. Watson, and et. al., “Flagship: A Parallel Architecture for Declarative Programming,” in *Proceedings of the 15th Annual International Symposium on Computer Architecture*, ser. ISCA ’88. Los Alamitos, CA, USA: IEEE Computer Society Press, 1988, pp. 124–130.

[31] W. M. Johnston, J. R. P. Hanna, and R. J. Millar, “Advances in Dataflow Programming Languages,” *ACM Comput. Surv.*, vol. 36, no. 1, pp. 1–34, Mar. 2004.

[32] A. Parashar, M. Pellauer, M. Adler, B. Ahsan, N. Crago, D. Lustig, V. Pavlov, A. Zhai, M. Gambhir, A. Jaleel, R. Allmon, R. Rayess, S. Maresh, and J. Emer, “Triggered instructions: A control paradigm for spatially-programmed architectures,” in *Proceedings of the 40th Annual International Symposium on Computer Architecture*, ser. ISCA ’13. New York, NY, USA: ACM, 2013, pp. 142–153.

[33] M. D. Allen, S. Sridharan, and G. S. Sohi, “Serialization sets: A dynamic dependence-based parallel execution model,” in *Proceedings of the 14th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, ser. PPoPP ’09. New York, NY, USA: ACM, 2009, pp. 85–96.

[34] S. S. Bhattacharyya, G. Brebner, J. W. Janneck, J. Eker, C. von Platen, M. Mattavelli, and M. Raulet, “OpenDF: A Dataflow Toolset for Reconfigurable Hardware and Multicore Systems,” *SIGARCH Comput. Archit. News*, vol. 36, no. 5, pp. 29–35, Jun. 2009.

[35] H. W. Tseng and D. M. Tullsen, “CDTT: Compiler-generated data-triggered threads,” in *2014 IEEE 20th International Symposium on High Performance Computer Architecture (HPCA)*, Feb 2014, pp. 650–661.

[36] T. Nowatzki, V. Gangadhar, and K. Sankaralingam, “Exploring the potential of heterogeneous von neumann/dataflow execution models,” in *Proceedings of the 42Nd Annual International Symposium on Computer Architecture*, ser. ISCA ’15. New York, NY, USA: ACM, 2015, pp. 298–310.

[37] G. Gupta and G. S. Sohi, "Dataflow execution of sequential imperative programs on multicore architectures," in *Proceedings of the 44th Annual IEEE/ACM International Symposium on Microarchitecture*, ser. MICRO-44. New York, NY, USA: ACM, 2011, pp. 59–70.

[38] S. Swanson, K. Michelson, A. Schwerin, and M. Oskin, "Wavescalar," in *Proceedings of the 36th Annual IEEE/ACM International Symposium on Microarchitecture*, ser. MICRO 36. Washington, DC, USA: IEEE Computer Society, 2003, pp. 291–.

[39] S. Tomov, R. Nath, H. Ltaief, and J. Dongarra, "Dense linear algebra solvers for multicore with GPU accelerators," in *Parallel Distributed Processing, Workshops and Phd Forum (IPDPSW), 2010 IEEE International Symposium on*, April 2010, pp. 1–8.

[40] J. D. Hogg, J. K. Reid, and J. A. Scott, "Design of a Multicore Sparse Cholesky Factorization Using DAGs," *SIAM Journal on Scientific Computing*, vol. 32, no. 6, pp. 3627–3649, 2010.

[41] J. Masci, A. Giusti, D. Ciresan, G. Fricout, and J. Schmidhuber, "A fast learning algorithm for image segmentation with max-pooling convolutional networks," in *2013 IEEE International Conference on Image Processing*, Sept 2013, pp. 2713–2717.

[42] A. Giusti, D. C. Ciresan, J. Masci, L. M. Gambardella, and J. Schmidhuber, "Fast Image Scanning with Deep Max-Pooling Convolutional Neural Networks," *CoRR*, vol. abs/1302.1700, 2013.

[43] P. Sermanet, D. Eigen, X. Zhang, M. Mathieu, R. Fergus, and Y. LeCun, "OverFeat: Integrated Recognition, Localization and Detection using Convolutional Networks," *CoRR*, vol. abs/1312.6229, 2013.

[44] A. Zlateski, K. Lee, and H. S. Seung, "ZNN - A Fast and Scalable Algorithm for Training 3D Convolutional Networks on Multi-Core and Many-Core Shared Memory Machines," *CoRR*, vol. abs/1510.06706, 2015.

[45] J. Dean and S. Ghemawat, "MapReduce: simplified data processing on large clusters," in *Proceedings of the 6th conference on Symposium on Opearting Systems Design & Implementation*, 2004.

[46] L. Dagum and R. Menon, "OpenMP: An industry standard API for shared-memory programming," *IEEE Computational Science and Engineering*, vol. 5, no. 1, pp. 46–55, Jan 1998.

[47] J. Planas, R. M. Badia, E. Ayguadé, and J. Labarta, "Hierarchical task-based programming with StarSs," *International Journal of High Performance Computing Applications*, vol. 23, no. 3, pp. 284–299, 2009.

[48] Z. Budimlić, M. Burke, V. Cavé, K. Knobe, G. Lowney, R. Newton, J. Palsberg, D. Peixotto, V. Sarkar, F. Schlimbach *et al.*, "Concurrent collections," *Scientific Programming*, vol. 18, no. 3-4, pp. 203–217, 2010.

[49] O. Pell and O. Mencer, "Surviving the end of frequency scaling with reconfigurable dataflow computing," *SIGARCH Comput. Archit. News*, vol. 39, no. 4, pp. 60–65, Dec. 2011.

[50] K. Stavrou, P. Evripidou, and P. Trancoso, "DDM-CMP: data-driven multi-threading on a chip multiprocessor," in *Embedded Computer Systems: Architectures, Modeling, and Simulation*. Springer, 2005, pp. 364–373.

[51] C. Kyriacou and P. Evripidou, *Network interface for a data driven network of workstations (D^2NOW)*. Berlin, Heidelberg: Springer Berlin Heidelberg, 1999, pp. 257–268.

[52] C. Kyriacou, P. Evripidou, and P. Trancoso, *CacheFlow: A Short-Term Optimal Cache Management Policy for Data Driven Multithreading*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2004, pp. 561–570.

[53] S. Arandi and P. Evripidou, "Programming multi-core architectures using Data-Flow techniques," in *International Conference on Embedded Computer Systems (SAMOS)*, July 2010, pp. 152–161.

[54] G. Michael, S. Arandi, and P. Evripidou, "Data-flow concurrency on distributed multi-core systems," in *Proceedings of the International Conference on Parallel and Distributed Processing Techniques and Applications (PDPTA)*, Athens, 2013, pp. 515–523.

[55] P. Trancoso, K. Stavrou, and P. Evripidou, "DDMCPP: The data-driven multithreading c pre-processor," in *The 11th Workshop on Interaction between Compilers and Computer Architectures*, 2007, p. 32.

[56] T. Harris, J. R. Larus, and R. Rajwar, *Transactional Memory, 2nd edition*, ser. Synthesis Lectures on Computer Architecture. Morgan & Claypool Publishers, 2010.

[57] S. L. Peyton Jones, *Beautiful Concurrency*. O'Reilly, 2007.

[58] M. M. K. Martin, M. D. Hill, and D. J. Sorin, "Why On-chip Cache Coherence is Here to Stay," *Commun. ACM*, vol. 55, no. 7, pp. 78–89, Jul. 2012.

[59] J. Howard, S. Dighe, and et. al., "A 48-Core IA-32 message-passing processor with DVFS in 45nm CMOS," in *IEEE International Solid-State Circuits Conference - (ISSCC)*, Feb 2010, pp. 108–109.

[60] E. Totoni, B. Behzad, S. Ghike, and J. Torrellas, "Comparing the power and performance of Intel's SCC to state-of-the-art CPUs and GPUs," in *Proceedings of the ISPASS '12*. Washington, DC, USA: IEEE Computer Society, 2012, pp. 78–87.

[61] I. A. C. Ureña, M. Riepen, and M. Konow, "RCKMPI–lightweight MPI implementation for Intel's Single-chip Cloud Computer (SCC)," in *Recent Advances in the Message Passing Interface*. Springer, 2011, pp. 208–217.

[62] C. Clauss, S. Lankes, P. Reble, and T. Bemmerl, "Evaluation and improvements of programming models for the Intel SCC many-core processor," in *HPCS International Conference*. IEEE, 2011, pp. 525–532.

[63] P. Gschwandtner, T. Fahringer, and R. Prodan, "Performance analysis and benchmarking of the Intel SCC," in *Proceedings of the IEEE CLUSTER '11*. Washington, DC, USA: IEEE Computer Society, 2011, pp. 139–149.

[64] D. Baliley, E. Barszcz, J. Barton, D. Browning, R. Carter, L. Dagum, R. Fatoohi, P. Frederickson, T. Lasinski, R. Schreiber *et al.*, "The NAS parallel benchmarks-summary and preliminary results," in *Proceedings of the 1991 ACM/IEEE Conference on Supercomputing*, vol. 91. New York, NY, USA: ACM, 1991, pp. 158–165.

[65] I. X. Phi, "The Intel Xeon Phi coprocessor," http://www.intel.com/content/www/us/en/processors/xeon/xeon-phi-detail.html, 2016, [Online].

[66] *The native POSIX thread library for Linux*, Red Hat Inc., 2003.

[67] W. Gropp, E. Lusk, N. Doss, and A. Skjellum, "A high-performance, portable implementation of the MPI message passing interface standard," *Parallel Computing*, vol. 22, no. 6, pp. 789 – 828, 1996.

[68] S. Saini, H. Jin, D. Jespersen, H. Feng, J. Djomehri, W. Arasin, R. Hood, P. Mehrotra, and R. Biswas, "An early performance evaluation of many integrated core architecture based SGI rackable computing system," in *2013 SC - International Conference for High Performance Computing, Networking, Storage and Analysis (SC)*, Nov 2013, pp. 1–12.

[69] A. Diavastos, P. Petrides, G. Falcao, and P. Trancoso, "LDPC Decoding on the Intel SCC," in *2012 20th Euromicro International Conference on Parallel, Distributed and Network-based Processing*, Feb 2012, pp. 57–65.

[70] R. G. Gallager, "Low-Density Parity-Check Codes," *IRE Transactions on Information Theory*, vol. 8, no. 1, pp. 21–28, January 1962.

[71] D. J. C. Mackay and R. M. Neal, "Near Shannon Limit Performance of Low Density Parity Check Codes," *IEEE Electronics Letters*, vol. 32, no. 18, pp. 1645–1646, August 1996.

[72] P. Petrides, A. Diavastos, C. Christofi, and P. Trancoso, "Scalability and Efficiency of Database Queries on Future Many-Core Systems," in *2013 21st Euromicro International Conference on Parallel, Distributed, and Network-Based Processing*, Feb 2013, pp. 24–28.

[73] T. P. Council, "TPC Benchmark H (Decision Support)," Standard Specification Revision 2.6.1, 2006.

[74] R. M. Karp and R. E. Miller, "Properties of a Model for Parallel Computations: Determinacy, Termination, Queueing," *SIAM Journal on Applied Mathematics*, vol. 14, no. 6, pp. 1390–1411, 1966.

[75] H. Vandierendonck, G. Tzenakis, and D. S. Nikolopoulos, "A Unified Scheduler for Recursive and Task Dataflow Parallelism," in *2011 International Conference on Parallel Architectures and Compilation Techniques (PACT 2011)*, Oct 2011, pp. 1–11.

[76] 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*, ser. PLDI '98. New York, NY, USA: ACM, 1998, pp. 212–223.

[77] K. B. Wheeler, R. C. Murphy, and D. Thain, “Qthreads: An API for programming with millions of lightweight threads,” in *2008 IEEE International Symposium on Parallel and Distributed Processing*, April 2008, pp. 1–8.

[78] M. C. Gilliland, B. J. Smith, and W. Calvert, “HEP - A semaphore-synchronized multiprocessor with central control (heterogeneous element processor),” in *1976 Summer Computer Simulation Conference*, July 1976, pp. 57–62.

[79] “Cray XMT platforrm,” http://www.cray.com/products/xmt/index.html, 2007, [Online].

[80] J. B. Brockman, S. Thoziyoor, S. K. Kuntz, and P. M. Kogge, “A Low Cost, Multithreaded Processing-in-memory System,” in *Proceedings of the 3rd Workshop on Memory Performance Issues: In Conjunction with the 31st International Symposium on Computer Architecture*, ser. WMPI ’04. New York, NY, USA: ACM, 2004, pp. 16–22.

[81] M. Hall, P. Kogge, J. Koller, P. Diniz, J. Chame, J. Draper, J. LaCoss, J. Granacki, J. Brockman, A. Srivastava, W. Athas, V. Freeh, J. Shin, and J. Park, “Mapping Irregular Applications to DIVA, a PIM-based Data-intensive Architecture,” in *Proceedings of the 1999 ACM/IEEE Conference on Supercomputing*, ser. SC ’99. New York, NY, USA: ACM, 1999.

[82] P. M. Kogge, S. C. Bass, J. B. Brockman, D. Z. Chen, and E. Sha, “Pursuing a Petaflop: point designs for 100 TF computers using PIM technologies,” in *Frontiers of Massively Parallel Computing, 1996. Proceedings Frontiers ’96., Sixth Symposium on the*, Oct 1996, pp. 88–97.

[83] M. Bauer, S. Treichler, E. Slaughter, and A. Aiken, “Legion: Expressing Locality and Independence with Logical Regions,” in *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, ser. SC ’12. IEEE Computer Society Press, 2012, pp. 1–11.

[84] T. G. Mattson, R. Cledat, V. Cavé, V. Sarkar, Z. Budimlić, S. Chatterjee, J. Fryman, I. Ganev, R. Knauerhase, M. Lee, B. Meister, B. Nickerson, N. Pepperling, B. Seshasayee, S. Tasirlar, J. Teller, and N. Vrvilo, “The Open Community Runtime: A runtime system for extreme scale computing,” in *2016 IEEE High Performance Extreme Computing Conference (HPEC)*, Sept 2016, pp. 1–7.

[85] J. Dokulil and S. Benkner, “Retargeting of the Open Community Runtime to Intel Xeon Phi,” *Procedia Computer Science*, vol. 51, pp. 1453 – 1462, 2015.

[86] S. Zuckerman, J. Suetterlein, R. Knauerhase, and G. R. Gao, “Using a ”Codelet” Program Execution Model for Exascale Machines: Position Paper,” in *Proceedings of the 1st International Workshop on Adaptive Self-Tuning Computing Systems for the Exaflop Era*, ser. EXADAPT ’11. New York, NY, USA: ACM, 2011, pp. 64–69.

[87] J. E. Stone, D. Gohara, and G. Shi, “OpenCL: A Parallel Programming Standard for Heterogeneous Computing Systems,” *Computing in Science Engineering*, vol. 12, no. 3, pp. 66–73, May 2010.

[88] J. Kim, S. Seo, J. Lee, J. Nah, G. Jo, and J. Lee, “SnuCL: an OpenCL framework for heterogeneous CPU/GPU clusters,” in *Proceedings of the 26th ICS ’12*. New York, NY, USA: ACM, 2012, pp. 341–352.

[89] J. Lee, J. Kim, J. Kim, S. Seo, and J. Lee, "An OpenCL framework for homogeneous manycores with no hardware cache coherence," in *PACT'11*. IEEE, 2011, pp. 56–67.

[90] OpenMP Architecture Review Board, "OpenMP 4.5 API C/C++ Syntax Reference Guide," http://www.openmp.org/wp-content/uploads/openmp-4.5.pdf, 2015, [Online].

[91] K. Costas, P. Evripidou, and P. Trancoso, "Data-Driven Multithreading Using Conventional Microprocessors," *IEEE Transactions on Parallel and Distributed Systems*, pp. 1176–1188, October 2006.

[92] R. Giorgi, Z. Popovic, and N. Puzovic, "DTA-C: A Decoupled multi-Threaded Architecture for CMP Systems," *19th International Symposium on Computer Architecture and High Performance Computing, 2007. SBAC-PAD 2007*, pp. 263 – 270, October 2007.

[93] P. S. Barth, R. S. Nikhil, and Arvind, "M-Structures: Extending a Parallel, Non-strict, Functional Language with State," in *Proceedings of the 5th ACM Conference on Functional Programming Languages and Computer Architecture*. London, UK: Springer-Verlag, 1991, pp. 538–568.

[94] R. Milner, M. Tofte, and R. Harper, *The Definition of Standard ML*. Cambridge, MA, USA: MIT Press, 1990.

[95] D. Syme, A. Granicz, and A. Cisternino, *Expert F# (Expert's Voice in .Net)*. Apress, 2012.

[96] S. P. Jones, A. Gordon, and S. Finne, "Concurrent Haskell," in *Annual Symposium on Principles of Programming Languages*. ACM, 1996, pp. 295–308.

[97] F. Pascal, F. Christof, and R. Torvald, "Dynamic Performance Tuning of Word-Based Software Transactional Memory," *Proceedings of the 13th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP)*, 2008.

[98] D. Dice, O. Shalev, and N. Shavit, "Transactional Locking II," *Proceedings of the 20th Intl. Symposium on Distributed Computing (DISC)*, 2006.

[99] J. Sreeram, R. Cledat, T. Kumar, and S. Pande, "RSTM : A Relaxed Consistency Software Transactional Memory for Multicores," *Parallel Architectures and Compilation Techniques, International Conference on*, vol. 0, p. 428, 2007.

[100] M. Ansari, C. Kotselidis, K. Jarvis, M. Luján, C. C. Kirkham, and I. Watson, "Advanced Concurrency Control for Transactional Memory Using Transaction Commit Rate," in *Euro-Par*, 2008, pp. 719–728.

[101] M. Ansari, M. Luján, C. Kotselidis, K. Jarvis, C. C. Kirkham, and I. Watson, "Steal-on-Abort: Improving Transactional Memory Performance through Dynamic Transaction Reordering," in *HiPEAC*, 2009, pp. 4–18.

[102] C. C. Minh, J. Chung, C. Kozyrakis, and K. Olukotun, "STAMP: Stanford Transactional Applications for Multi-processing," *In IISWC '08: Proceedings of The IEEE International Symposium on Workload Characterization,*, September 2008.

[103] I. Watson, C. Kirkham, and M. Luján, “A Study of a Transactional Parallel Routing Algorithm,” *PACT ’07 Proceedings of the 16th International Conference on Parallel Architecture and Compilation Techniques*, pp. 388–398, 2007.

[104] J. Howard, S. Dighe, Y. Hoskote, S. Vangal, D. Finan, G. Ruhl, D. Jenkins, H. Wilson, N. Borkar, G. Schrom *et al.*, “A 48-core IA-32 message-passing processor with DVFS in 45nm CMOS,” in *IEEE International ISSCC 2010*. IEEE, 2010, pp. 108–109.

[105] M. Loghi, M. Poncino, and L. Benini, “Cache coherence tradeoffs in shared-memory MPSoCs,” *TECS’06*, vol. 5, no. 2, pp. 383–407, 2006.

[106] D. Hackenberg, D. Molka, and W. E. Nagel, “Comparing cache architectures and coherency protocols on x86-64 multicore SMP systems,” in *Proceedings of the MICRO 42*. New York, NY, USA: ACM, 2009, pp. 413–422.

[107] M. C. Seiler and F. A. Seiler, “Numerical recipes in C: the art of scientific computing,” *Risk Analysis*, vol. 9, no. 3, pp. 415–416, 1989.

[108] M. R. Guthaus, J. S. Ringenberg, D. Ernst, T. M. Austin, T. Mudge, and R. B. Brown, “MiBench: A Free, Commercially Representative Embedded Benchmark Suite,” in *Proceedings of WWC ’01 IEEE International Workshop*. Washington, DC, USA: IEEE Computer Society, 2001, pp. 3–14.

[109] P. Virouleau, P. Brunet, F. Broquedis, N. Furmento, S. Thibault, O. Aumage, and T. Gautier, “Evaluation of OpenMP Dependent Tasks with the KASTORS Benchmark Suite,” in *Proceedings of the 10th International Workshop on OpenMP (IWOMP 2014).*, 2014, pp. 16–29.

[110] A. V. Aho, M. R. Garey, and J. D. Ullman, “The Transitive Reduction of a Directed Graph,” *SIAM Journal on Computing*, vol. 1, no. 2, pp. 131–137, 1972.

[111] A. Diavastos, G. Stylianou, and G. Koutsou, “Exploiting Very-Wide Vectors on Intel Xeon Phi with Lattice-QCD Kernels,” in *24th Euromicro International Conference on Parallel, Distributed, and Network-Based Processing (PDP)*, Feb 2016, pp. 296–300.

[112] A. Duran, X. Teruel, R. Ferrer, X. Martorell, and E. Ayguade, “Barcelona OpenMP Tasks Suite: A Set of Benchmarks Targeting the Exploitation of Task Parallelism in OpenMP,” in *Proceedings of the International Conference on Parallel Processing*, ser. ICPP ’09. Washington, DC, USA: IEEE Computer Society, 2009, pp. 124–131.

[113] S. Yoon and A. Jameson, “Lower-upper symmetric-Gauss-Seidel method for the Euler and Navier-Stokes equations,” *AIAA journal*, vol. 26, no. 9, pp. 1025–1026, 1988.

[114] Red Hat Developer Program, “What is new in OpenMP 4.5,” https://developers.redhat.com/blog/2016/03/22/what-is-new-in-openmp-4-5-3, 2016, [Online].

[115] B. Li, H. C. Chang, S. Song, C. Y. Su, T. Meyer, J. Mooring, and K. W. Cameron, “The Power-Performance Tradeoffs of the Intel Xeon Phi on HPC Applications,” in *2014 IEEE International Parallel Distributed Processing Symposium Workshops*, May 2014, pp. 1448–1456.

[116] R. A. A. Na'mneh and K. A. Darabkh, "A new genetic-based algorithm for scheduling static tasks in homogeneous parallel systems," in *2013 International Conference on Robotics, Biomimetics, Intelligent Computational Systems*, Nov 2013, pp. 46–50.

[117] A. Y. Zomaya, C. Ward, and B. Macey, "Genetic scheduling for parallel processor systems: comparative studies and performance issues," *IEEE Transactions on Parallel and Distributed Systems*, vol. 10, no. 8, pp. 795–812, Aug 1999.

[118] M. M. Najafabadi, M. Zali, S. Taheri, and F. Taghiyareh, "Static Task Scheduling Using Genetic Algorithm and Reinforcement Learning," in *2007 IEEE Symposium on Computational Intelligence in Scheduling*, April 2007, pp. 226–230.

[119] H. M. Ghader, K. Fakhr, M. Javadi, and G. Bakhshzadeh, "Static task graph scheduling using learner Genetic Algorithm," in *2010 International Conference of Soft Computing and Pattern Recognition*, Dec 2010, pp. 357–362.

[120] S. Gupta, V. Kumar, and G. Agarwal, "Task Scheduling in Multiprocessor System Using Genetic Algorithm," in *2010 Second International Conference on Machine Learning and Computing*, Feb 2010, pp. 267–271.

[121] Y. Wen, H. Xu, and J. Yang, "A heuristic-based hybrid genetic-variable neighborhood search algorithm for task scheduling in heterogeneous multiprocessor system," *Information Sciences*, vol. 181, no. 3, pp. 567 – 581, 2011.

[122] T. Lewis and H. El-Rewini, "Parallax: a tool for parallel program scheduling," *IEEE Parallel Distributed Technology: Systems Applications*, vol. 1, no. 2, pp. 62–72, May 1993.

[123] S. Pei, J. Wang, W. Cui, L. Jiang, T. Geng, J. L. Gaudiot, and S. Zuckerman, "Codelet Scheduling by Genetic Algorithm," in *2016 IEEE Trustcom/BigDataSE/ISPA*, Aug 2016, pp. 1492–1499.

[124] K. Kaur, A. Chhabra, and G. Singh, "Heuristics based genetic algorithm for scheduling static tasks in homogeneous parallel system," *International Journal of Computer Science and Security (IJCSS)*, vol. 4, no. 2, pp. 183–198, 2010.

[125] K. Deb, A. Pratap, S. Agarwal, and T. Meyarivan, "A fast and elitist multiobjective genetic algorithm: NSGA-II," *IEEE Transactions on Evolutionary Computation*, vol. 6, no. 2, pp. 182–197, Apr 2002.

[126] N. Srinivas and K. Deb, "Muiltiobjective Optimization Using Nondominated Sorting in Genetic Algorithms," *Evolutionary Computation*, vol. 2, no. 3, pp. 221–248, 1994.

[127] Kanpur Genetic Algorithms Laboratory, "Multi-objective NSGA-II code in C," http://www.iitk.ac.in/kangal/codes.shtml, 2011, [Online].

[128] B. L. Miller and D. E. Goldberg, "Genetic algorithms, tournament selection, and the effects of noise," *Complex systems*, vol. 9, no. 3, pp. 193–212, 1995.

[129] Intel, "Intel Transactional Synchronization Extensions (Intel TSX) Overview," https://software.intel.com/en-us/node/524022, 2016, [Online].

[130] LLVM Team, "The LLVM Compiler Infrastructure," https://llvm.org/, 2017, [Online].

Appendix A

SWITCHES Compiler Directives

A.1 SWITCHES API

The SWITCHES API described here is a subset of the OpenMP standard v4.5. The implementation of the Translator tool and instructions for installation and usage of the tool-chain can be found in [1].

A.1.1 Master

```
#pragma omp master
{ ... }
```

Definition	Specifies that the containing structured block will be executed only by the master thread of the execution (`__MAIN_KERNEL`).

Clause	**Default Value**	**Description**
NONE	NONE	NONE

A.1.2 Global variables

```
__sw_global__
```

Definition	Used only in front of a global variable declaration in the program in order to declare the global variable as extern to the new produced file.

A.1.3 Parallel

```
#pragma omp parallel [clause, [[,]clause]...]
{ ... }
```

Definition	Forms a team of threads and start parallel execution. Tasks within the `parallel` construct will be executed in parallel.

Clause	**Default Value**	**Description**
`num_threads(INT)`	ALL	Defines the number of threads to use in the execution of task declared in the specified parallel construct.
`default(STR)`	shared	Defines how to treat all variables inside the parallel construct. **`STR: shared, none`**
`private(list)`	-	For each variable in the `private` list, a separate copy will be created for thread contributing to the execution.
`firstprivate(list)`	-	Same as previous but each separate copy that will be created will also be initialized to the global value at the time of calling.

A.1.4 For

```
#pragma omp for [clause [[,]clause]...]
{
  for (I = 0; I < SIZE; I++)
    ...
}
```

Definition	Specifies that the iterations of the for loop found after the directive will be executed in parallel by threads in the team.

Clause	Default Value	Description
`num_threads(INT)`	ALL	Defines the number of threads to use in the execution of task declared in the specified parallel construct.
`private(list)`	-	For each variable in the `private` list, a separate copy will be created for thread contributing to the execution.
`firstprivate(list)`	-	Same as previous but each separate copy that will be created will also be initialized to the global value at the time of calling.
`schedule(type [, chunk_size])`	static, 32	Declare the type of scheduling policy to use for executing the iterations of the loop. **`type: static, cross`** **`chunk_size: VARIABLE | INT`**
`reduction (operation :list)`	-	Identify the loop as a reduction loop. The `list` declares all reduction variables in the body of the loop **`operation: +, *, -, &, ^, |, &&, ||`**
`depend(type:list)`	-	Declare the dependences of this loop with other tasks. **`type: in, out, inout`**

A.1.5 Sections

```
#pragma omp sections
{
  [#pragma omp section]
    ...
  [#pragma omp section]
    ...
}
```

Definition	A non-iterative work-sharing construct that contains a set of tasks (with no dependences) that are to be distributed among and executed by the threads in the team.

Clause	**Default Value**	**Description**
NONE	NONE	NONE

A.1.6 Section

```
#pragma omp section
{ ... }
```

Definition	Specify a section task (with no dependences).

Clause	**Default Value**	**Description**
NONE	NONE	NONE

A.1.7 Task

```
#pragma omp task [clause [[,]clause]...]
{ ... }
```

Definition	Defines an explicit task with possible dependences.

Clause	**Default Value**	**Description**
`default(STR)`	shared	Defines how to treat all variables inside the parallel construct. **`STR: shared, none`**
`private(list)`	-	For each variable in the `private` list, a separate copy will be created for thread contributing to the execution.
`firstprivate(list)`	-	Same as previous but each separate copy that will be created will also be initialized to the global value at the time of calling.
`depend(type:list)`	-	Declare the dependences of this loop with other tasks. **`type: in, out, inout`**

A.1.8 Taskloop

```
#pragma omp taskloop [clause [[,]clause]...]
{
  for (I = 0; I < SIZE; I++)
    ...
}
```

Definition	Specifies that the iterations of the for loop found after the directive will be executed in parallel by threads in the team.

Clause	Default Value	Description
`num_threads(INT)`	ALL	Defines the number of threads to use in the execution of task declared in the specified parallel construct.
`private(list)`	-	For each variable in the `private` list, a separate copy will be created for thread contributing to the execution.
`firstprivate(list)`	-	Same as previous but each separate copy that will be created will also be initialized to the global value at the time of calling.
`schedule(type [, chunk_size])`	static, 32	Declare the type of scheduling policy to use for executing the iterations of the loop. **`type: static, cross`** **`chunk_size: VARIABLE | INT`**
`reduction (operation :list)`	-	Identify the loop as a reduction loop. The `list` declares all reduction variables in the body of the loop **`operation: +, *, -, &, ^, |, &&, ||`**
`depend(type:list)`	-	Declare the dependences of this loop with other tasks. **`type: in, out, inout`**

A.1.9 Parallel For

```
#pragma omp parallel for [clause [[,]clause]...]
{
  for (I = 0; I < SIZE; I++)
    ...
}
```

Definition	This construct defines a new parallel function that will only contain one for loop task. Specifies that the iterations of the for loop found after the directive will be executed in parallel by threads in the team.

Clause	**Default Value**	**Description**
`num_threads(INT)`	ALL	Defines the number of threads to use in the execution of task declared in the specified parallel construct.
`default(STR)`	shared	Defines how to treat all variables inside the parallel construct. **`STR: shared, none`**
`private(list)`	-	For each variable in the `private` list, a separate copy will be created for thread contributing to the execution.
`schedule(type [, chunk_size])`	static, 32	Declare the type of scheduling policy to use for executing the iterations of the loop. **`type: static, cross`** **`chunk_size: VARIABLE | INT`**
`reduction (operation :list)`	-	Identify the loop as a reduction loop. The `list` declares all reduction variables in the body of the loop **`operation: +, *, -, &, ^, |, &&, ||`**

A.1.10 Parallel Sections

```
#pragma omp parallel sections [clause [[,]clause]...]
{
  [#pragma omp section]
    ...
  [#pragma omp section]
    ...
}
```

Definition	This construct defines a new parallel function that will only contain section task with no dependences. A non-iterative work-sharing construct that contains a set of tasks (with no dependences) that are to be distributed among and executed by the threads in the team.

Clause	Default Value	Description
`num_threads(INT)`	ALL	Defines the number of threads to use in the execution of task declared in the specified parallel construct.
`default(STR)`	shared	Defines how to treat all variables inside the parallel construct. **`STR: shared, none`**

A.2 SWITCHES Examples

In this section we present some examples of how a SWITCHES program is implemented using `#pragma` directives. More applications and examples, along with the Translator tool can be found in the SWITCHES framework on-line at [1].

A.2.1 Example 1: Tasks Dependences

Figure A.1 below, shows a SWITCHES example that implements simple tasks with dependences. In this example task are declared using the `#pragma omp task` directive and name the policy to execute the iterations as `cross`. More SWITCHES applications and examples can be found in [1].

```
...
#pragma omp parallel
{
   // Task 1
   #pragma omp task depend(out: a, result)
   {
      result = a + b;
      a++;
   }

   // Task 2
   #pragma omp task depend(in: a, result) depend(out: a, b)
   {
      b += result;
      a++;
   }

   // Loop Task 3
   #pragma omp for private(j) schedule(static, 4) depend(in: a, b, result)
   {
      for (j = 0; j < SIZE; j++)
         k1[j] = a*(power[j]-result) + b;
   }
}
...
```

Figure A.1: This is an example of how task and dependences are declared in a SWITCHES program.

A.2.2 Example 2: Cross-Loop Dependences

Figure A.2 below, shows an example of a SWITCHES application. It implements RK4, that solves a differential equation [18]. This type of application presents cross-loop iteration dependences that can be declared using the `#pragma omp taskloop` directive and name the policy to execute the iterations as `cross`. This example also shows the introduced clause for resource allocation on task-based directives.

```
...
#pragma omp parallel num_threads(10)
{
  // Loop A
  #pragma omp taskloop schedule(cross, 4) depend(out: k1[0:4]) num_threads(5)
  {
    for (i = 0; i < SIZE; i++)
    {
      yt[i] = 0.0;
      for (j = 0; j < SIZE; j++)
        yt[i] += c[i][j]*y[j];
      k1[i] = h*(power[i]-yt[i]);
    }
  }

  // Loop B
  #pragma omp taskloop schedule(cross, 4) depend(in: k1[0:4]) depend(out: k2[0:4])
  {
    for (j = 0; j < SIZE; j++)
    {
      yt[j] = 0.0;
      for (k = 0; k < SIZE; k++)
        yt[j] += c[j][k]*(y[k]+0.5*k1[j]);
      k2[j] = h*(power[j]-yt[j]);
    }
  }
  ...
}
```

Figure A.2: Source code from an actual SWITCHES application (RK4, presented in Section 5.5). This example is part of a larger synthetic application that is based on a differential equation kernel.