Efficient OpenMP Runtime Support for General-Purpose and Embedded Multi-Core Platforms

A Dissertation

submitted to the designated
by the General Assembly of Special Composition
of the Department of Computer Science and Engineering
Examination Committee

by

Spiros N. Agathos

in partial fulfillment of the requirements for the degree of

DOCTOR OF PHILOSOPHY

University of Ioannina

April 2016

Advisory committee:

* **Vassilios V. Dimakopoulos**, Associate Professor, Department of Computer Science and Engineering, University of Ioannina
* **Panagiota Fatourou**, Assistant Professor, Department of Computer Science, University of Crete
* **Georgios Manis**, Assistant Professor, Department of Computer Science and Engineering, University of Ioannina

Examination committee:

* **Vassilios V. Dimakopoulos**, Associate Professor, Department of Computer Science and Engineering, University of Ioannina
* **Panagiota Fatourou**, Assistant Professor, Department of Computer Science, University of Crete
* **Ioannis Fudos**, Associate Professor, Department of Computer Science and Engineering, University of Ioannina
* **Panagiotis Hadjidoukas**, Senior Assistant, Computational Science and Engineering Laboratory, ETH Zurich
* **Kostas Magoutis**, Assistant Professor, Department of Computer Science and Engineering, University of Ioannina
* **Georgios Manis**, Assistant Professor, Department of Computer Science and Engineering, University of Ioannina
* **Dimitris Nikolopoulos**, Professor, School of Electronics, Electrical Engineering and Computer Science, Queen's University of Belfast

Dedication

I dedicate this dissertation to my family.

Acknowledgements

I would like to thank my advisor Prof. Vassilios V. Dimakopoulos for the great support and cooperation during my work as a PhD candidate. I would also like to thank the members of my advisory committee, Prof. Panagiota Fatourou and Prof. Georgios Manis for their guidance. Special thanks to Dr. Panagiotis Hadjidoukas and my colleagues Dr. Nikos Kallimanis, Alexandros Padadogiannakis and Aggelos Mourelis for their help and support in our joint works. Special acknowledgments to Adapteva for providing me with a Parallella-16 board through the Parallella university program. A major part of my work was based on the Parallella board. Many thanks to Prof. Nikitas Dimopoulos for providing access to large multiprocessor machines of the Western Canada Research grid where I ran some of the experiments of this dissertation. I would also like to thank the rest of the members of my examination committee Prof. Dimitris Nikolopoulos, Prof. Ioannis Fudos and Prof. Kostas Magoutis. Special thanks to all my friends for all their support, encouragement and patience during my studies. Finally, I would like to thank the Greek State Scholarships Foundation (IKY) for the financial support they provided me through a PhD scholarship.

Table of Contents

List of Figures

List of Tables

List of Algorithms

Abstract

Spiros N. Agathos, Ph.D., Department of Computer Science and Engineering, University of Ioannina, Greece, April 2016.
Efficient OpenMP Runtime Support for General-Purpose and Embedded Multi-Core Platforms.
Advisor: Vassilios V. Dimakopoulos, Associate Professor.

OpenMP is the standard programming model for shared memory multiprocessors and is currently expanding its target range beyond such platforms. The tasking model of OpenMP has been used successfully in a wide range of parallel applications. With tasking, OpenMP expanded its applicability beyond loop-level parallelization. Tasking allows efficient expression and management of irregular and dynamic parallelism. Recently, another significant addition to OpenMP was the introduction of device directives that target systems consisting of general-purpose hosts and accelerator devices that may execute portions of a unified application code. OpenMP thus encompasses heterogeneous computing, as well.

This dissertation deals with the problem of designing and implementing a productive and performance-oriented infrastructure to support the OpenMP parallel programming model. The first group of contributions refers to the efficient support of the OpenMP tasking model and its application to provide a novel solution to the problem of nested loop parallelism. We present the design and implementation of a tasking subsystem in the context of the ompi OpenMP compiler. Portions of this subsystem were re-engineered, and fast work-stealing structures were exploited, resulting a highly efficient implementation of OpenMP tasks for numa systems. Then we show how the tasking subsystem can be used to handle difficult problems such as nested loop parallelism. We provide a novel technique, whereby the nested parallel loops can be transparently executed by a single level of threads through the existing tasking subsystem.

The second group of contributions is related to the design and implementation of efficient OpenMP infrastructures for embedded and heterogeneous multicore systems. Here we present the way we enabled OpenMP exploitation of the sthorm accelerator. An innovative feature of our design is the deployment of the OpenMP model both at the host and the fabric sides, in a seamless way. Next we present the first implementation of the OpenMP 4.0 accelerator directives for the Parallella board, a very popular credit-card sized multicore system consisting of a dual-core arm host processor and a distinct 16-core Epiphany co-processor.

Finally, we propose a novel compilation technique which we term CARS; it features a Compiler-assisted Adaptive Runtime System which results in application-specific support by implementing each time only the required OpenMP functionality. The technique is of general applicability and can lead to dramatic reduction in executable sizes and/or execution times.

Εκτεταμενη Περιληψη

Σπυρίδων Ν. Αγάθος, Δ.Δ., Τμήμα Μηχανικών Η/Υ και Πληροφορικής, Πανεπιστήμιο Ιωαννίνων, Απρίλιος 2016.
Αποδοτική Υποστήριξη OpenMP για Γενικού Σκοπού και Ενσωματωμένες Πολυπύρηνες Αρχιτεκτονικές.
Επιβλέπων: Βασίλειος Β. Δημακόπουλος, Αναπληρωτής Καθηγητής.

Η συνεχής ανάγκη για περισσότερη επεξεργαστική ισχύ, σε συνδυασμό με φυσικούς περιορισμούς στην κατασκευή μικροηλεκτρονικών διατάξεων πολύ μεγάλης κλίμακας ολοκλήρωσης, οδήγησαν στη εμφάνιση πολυπύρηνων επεξεργαστών. Οι πολυπύρηνοι επεξεργαστές αποτελούν πια μονόδρομο στην κατασκευή υπολογιστικών συστημάτων. Οι μεγάλες εταιρίες κατασκευής επεξεργαστών κατασκευάζουν επεξεργαστές με δύο, τέσσερις ή οκτώ πυρήνες επεξεργασίας. Οι σύγχρονοι υπερυπολογιστές αποτελούνται πλέον από χιλιάδες πολυπύρηνους κόμβους οι οποίοι διασυνδέονται μέσω ταχύτατων δικτύων. Τα σύγχρονα υπολογιστικά συστήματα χαρακτηρίζονται επίσης από την ανομοιογένεια, την χρήση δηλαδή επεξεργαστών διαφορετικών τύπων και δυνατοτήτων, όπως παραδείγματος χάριν πολυπύρηνους επεξεργαστές και μονάδες γραφικής επεξεργασίας γενικού σκοπού (GPGPU).

Στην καθημερινότητά μας έχουν εισέλθει επίσης τα ενσωματωμένα συστήματα (embedded systems), δηλαδή υπολογιστές με σχετικά περιορισμένους πόρους, σχεδιασμένοι να εκτελούν ορισμένες εξειδικευμένες λειτουργίες. Παραδείγματα τέτοιων συσκευών είναι οι ψηφιακοί βοηθοί, mp3/mp4 players, κινητά τηλέφωνα, κονσόλες βιντεοπαιχνιδιών, ψηφιακές φωτογραφικές μηχανές, καθώς και οικιακές συσκευές. Μέχρι πρόσφατα τα ενσωματωμένα συστήματα διέθεταν έναν πυρήνα επεξεργασίας που είχε συνήθως την μορφή μικροελεγκτή ή επεξεργαστή ψηφιακού σήματος (DSP). Οι ολοένα αυξανόμενες απαιτήσεις σε ταχύτητα και λειτουργικότητα είχαν ως αποτέλεσμα την καθιέρωση ενσωματωμένων συστημάτων τα οποία πλέον σχεδιάζονται βάσει πολυπύρηνων επεξεργαστών γενικού σκοπού.

Το βασικό μειονέκτημα των παράλληλων συστημάτων ήταν ανέκαθεν η δυσκολία που υπάρχει στον προγραμματισμό τους. Η μετάβαση από τα σειριακά προγραμματιστικά μοντέλα στα παράλληλα είναι μια αρκετά επίπονη διαδικασία και εισάγει πολλά προβλήματα τόσο στην σχεδίαση όσο και στην ανάπτυξη/αποσφαλμάτωση των εφαρμογών.

Το OpenMP αποτελεί το πλέον διαδεδομένο και αποδεκτό πρότυπο για ανάπτυξη πολυνηματικών εφαρμογών σε συστήματα κοινόχρηστης μνήμης. Είναι βασισμένο στις γλώσσες C/C++ και Fortran και σήμερα βρίσκεται στην έκδοση 4.5. Στην έκδοση 3.0, που παρουσιάστηκε το 2008, προστέθηκαν αρκετές νέες δυνατότητες έκφρασης παραλληλισμού. Η κυριότερη αλλαγή ήταν η προσθήκη των αυτόνομων εργασιών (tasks), χάρη στις οποίες μπορεί εύκολα να εκφραστεί αναδρομικός και ακανόνιστος (irregular) παραλληλισμός. Το 2013 παρουσιάστηκε η έκδοση 4.0, όπου προστέθηκε η δυνατότητα αξιοποίησης ανομοιογενών επεξεργαστικών μονάδων όπως για παράδειγμα gpgpus και διάφορων τύπων επιταχυντών (accelerators) όπως για παράδειγμα ο Intel Xeon Phi.

Η διατριβή αυτή ασχολείται με την σχεδίαση και υλοποίηση μιας υποδομής για τον προγραμματισμό παράλληλων συστημάτων η οποία στοχεύει σε υψηλές επιδόσεις. Το πρώτο μέρος της διατριβής αναφέρεται στην αποδοτική υποστήριξη του μοντέλου εργασιών του OpenMP και στην αξιοποίησή του στην περίπτωση του εμφωλευμένου παραλληλισμού (nested parallelism).

Αρχικά παρουσιάζεται ο σχεδιασμός και η υλοποίηση μιας βιβλιοθήκης για την υποστήριξη των εργασιών στον ερευνητικό μεταφραστή ompi. Στην συνέχεια παρουσιάζονται οι αλλαγές που έγιναν στη βιβλιοθήκη αυτή με στόχο να βελτιστοποιηθούν οι επιδόσεις της κατά την εκτέλεση εφαρμογών σε συστήματα που έχουν χαρακτηριστικά numa (Non Uniform Memory Access). Στο πλαίσιο αυτό επανασχεδιάστηκαν κρίσιμα μέρη της και αναπτύχθηκε ένας βελτιστοποιημένος μηχανισμός κλεψίματος εργασιών (work-stealing mechanism). Στην συνέχεια αναλύεται ο τρόπος με τον οποίο ένα σύστημα εκτέλεσης εργασιών μπορεί διαχειριστεί το πρόβλημα του εμφωλευμένου παραλληλισμού. Προτείνουμε μια καινοτόμα τεχνική, όπου βρόχοι εμφωλευμένου παραλληλισμού (nested parallel loops) μπορούν να εκτελεστούν από νήματα μιας ομάδας OpenMP, χωρίς την ανάγκη δημιουργίας νέων εμφωλευμένων νημάτων. Η τεχνική αυτή μπορεί να υλοποιηθεί διαφανώς στο σύστημα εκτέλεσης εργασιών ενός μεταφραστή OpenMP.

Το δεύτερο μέρος της διατριβής σχετίζεται με τον σχεδιασμό και την υλοποίηση

αποδοτικών υποδομών για ενσωματωμένα και πολυπύρηνα ετερογενή συστήματα. Αρχικά σχεδιάσαμε και αναπτύξαμε μια υποδομή η οποία θα υποστηρίζει την εκτέλεση κώδικα OpenMP σε ετερογενή συστήματα τα οποία διαθέτουν τον πολυπύρηνο ενσωματωμένο συν-επεξεργαστή sthorm. Η καινοτομία της συγκεκριμένης εργασίας έγκειται στην υποστήριξη της εκτέλεσης οδηγιών OpenMP τόσο στον κεντρικό επεξεργαστή του συστήματος (host) όσο και στον sthorm. Στην συνέχεια παρουσιάζεται η πρώτη υλοποίηση των οδηγιών OpenMP 4.0 για επιταχυντές στο ετερογενές σύστημα Parallella. Πρόκειται για ένα πολυπύρηνο σύστημα μεγέθους πιστωτικής κάρτας, το οποίο διαθέτει έναν διπύρηνο επεξεργαστή arm ως κεντρικό (host) και έναν 16-πύρηνο συν-επεξεργαστή Epiphany.

Τέλος, παρουσιάζεται μια καινοτόμα τεχνική που μπορεί να εφαρμοστεί σε μεταφραστές OpenMP. Η τεχνική αυτή ονομάζεται CARS (Compiler-assisted Adaptive Runtime System) και αποσκοπεί στην δημιουργία βιβλιοθηκών υποστήριξης OpenMP κατάλληλα προσαρμοσμένων στις απαιτήσεις της εκάστοτε εφαρμογής. Σύμφωνα με την προτεινόμενη τεχνική, κατά την ανάλυση του κώδικα υπολογίζονται ορισμένες μετρικές που σκιαγραφούν την συμπεριφορά της εφαρμογής. Έπειτα οι μετρικές αυτές αξιοποιούνται από υπομονάδα του μεταφραστή ώστε να επιλεγεί ή να δημιουργηθεί δυναμικά μια προσαρμοσμένη/βελτιστοποιημένη έκδοση των βιβλιοθηκών υποστήριξης για την συγκεκριμένη εφαρμογή. Η τεχνική αυτή είναι γενικού σκοπού, όμως μπορεί να αποδειχθεί ιδιαίτερα χρήσιμη στην περίπτωση όπου κώδικας OpenMP εκτελείται σε κάποιος είδος συν-επεξεργαστή, οδηγώντας σε δραματική μείωση του μεγέθους του παραγόμενου εκτελέσιμου καθώς και σε αύξηση επιδόσεων.

Chapter 1

Introduction

1.1 Contemporary Parallel and Embedded Systems

Throughout the history of computers the demand for increasing their performance was always on the foreground and directed the design and architecture of the new technology products. Until the late 90's the continuous shrinking of semiconductor size allowed the increase of integration density and clock frequencies, which usually resulted in greater performance. The physical limits of microelectronics made slowly their appearance, causing energy consumption and other problems. The answer to all these challenges was the introduction of multicore CPUs. The idea of integrating two cores into a single chip existed since the mid 80's when Rockwell International manufactured versions of the 6502 with two processors on one chip namely the R65C00, R65C21, and R65C29. Nowadays, multicore CPUs have conquered all the computing system areas. Major manufacturers started developing CPUs with two, four, eight or more processing cores.

Packaging an increasing number of cores however, implies communication issues: Specialized Networks-on-Chip (NoCs) are also integrated in some many-core systems. Other systems are equipped with complicated memory/cache hierarchies. Recent systems for personal computing include multicore CPUs with 8 or more cores and an advanced interconnection network. Furthermore, some systems may include two or more multicore chips in a non-uniform memory access (NUMA) fashion. Finally, the contemporary supercomputers are comprised of hundreds (or thousands) nodes containing multicore CPUs, which communicate through very fast optical networks.

Aside from personal computers, smaller portable devices have recently earned their place in every day life interactions. The internet of things (IoT) is taking its first steps; this network of physical objects creates opportunities for more direct integration between the physical world and computer-based systems, and results in improved efficiency, accuracy and economic benefit. Some experts estimate that by 2020 the number of connected devices will be around 50 billion [1]. These devices are in essence miniature systems developed to execute special tasks. For example we have personal assistants, smart devices (phones, televisions, cameras), video consoles or media players. They tend to be "smart" by embedding a complete computing subsystem. Modern home appliances and automated systems are characteristic examples: Microwave ovens, washing machines, refrigerators, advanced air conditioning etc, all include embedded computing systems.

The first generations of the embedded systems were based on custom application-driven hardware, designed for a fixed set of computational tasks. Later, the design methodology shifted towards programmable hardware combined with low-level system software to implement the task in question. Micro-controllers and/or digital signal processor (DSP) are still used as the "brains" in many embedded systems. Nevertheless, with the rapid advances in hardware design and manufacture, general-purpose, CPUs conquered the embedded system world. These are full CPUs albeit with possibly limited resources, such as cache memories.

As in computer systems, the functional complexity and energy restrictions led to the development of multicore embedded systems. These systems have usually heterogeneous computing units. For example they may include a dual core CPU chip packaged with a set of DSPs or with another chip that acts as an accelerator. An example of a heterogeneous multiprocessor system on chip is the STHORM [2] architecture. The platform consists of a set of low-power general-purpose processing

elements, working in full MIMD mode, interconnected by an asynchronous NoC. In addition, the architecture allows the inclusion of special-purpose processing cores to fit particular application domains.

The majority of recent systems, either general-purpose or embedded, follow a heterogeneous paradigm. Personal computers are equipped with multicore CPUs and one or more GPUs; the latter were originally designed to accelerate graphical processing and are placed in video game consoles and even in our cell phones. Modern GPUs include hundreds of logical processing units, large amount of memory and high bandwidth memory channels resulting in massive computing capabilities. During the 00's, with the advent of programmable shaders and floating point support on GPUs, the idea of using them as a general-purpose accelerator became both popular and practical, giving rise to GPGPUs.

Another popular computing system that acts as a co-processor is the Intel Xeon Phi [3], supporting up to 61 general purpose cores and up to 16GB of high bandwidth memory. The super computer area has turned to heterogeneous nodes both for computational and energy savings reasons. The upcoming generation of Xeon Phi processor family, namely Knights Landing (KNL) [4] will sport 72 cores (144 threads) and 16 gigabytes of very fast memory (MCDRAM) on a chip. This chip can be used either as a many-core host processor or as a PCIe attached co-processor. The notion of the accelerator computing is also present in embedded systems; the popular Parallella [5] board sports a dual-core ARM-based host processor and a 16-core Epiphany accelerator.

1.2 Challenges in Multicore Systems

Nowadays multicore systems are at the disposal of everyone. From high end supercomputer systems to wearable devices, all benefit from parallel hardware. High performance hardware is not expensive either: The credit card sized Parallella board costs around 99$ and its Epiphany accelerator has a peak performance of approximately 25 GFLOPS (single-precision) with a maximum power dissipation of less than 2 Watt. It is with no doubt that the hardware is making big progress steps, but this alone is not enough to achieve increased performance. Currently the biggest challenge is the development of software that takes advantage of a parallel system.

The rise of parallel or multicore systems appeared alongside with the difficulty to write programs that benefit from the extra computing cores. The transition from serial to parallel programming models is a laborious procedure and induces a lot of issues. The sensible way to write a program is serially: A programmer places commands one after another to complete a task. This fundamental principle is not applicable when it comes to programs for multicore machines. The design phase of parallel program involves the notion of cooperating execution entities, for example processes, or some kind of available threads. Thus, the designer takes over the thread synchronization, the mutual exclusion of shared resources and data, the avoidance of possible deadlocks, the workload balance of the threads etc. These additional restrictions result in time consuming and complicated designs for the applications.

After the design, the development phase follows, which implies a different set of issues. The performance of a parallel application depends on whether the corresponding code takes advantage of the underlying hardware characteristics. For example programming a NUMA machine requires deep knowledge of the multi-level cache hierarchies, the interconnection networks, the point-to-point communication between CPUs etc. Programming heterogeneous systems that include a GPGPU or another type of co-processor involves special skills and knowledge of their unique system architectures. Low level facilities that are closer to the hardware offer higher performance gains and hardware utilization. This performance comes with lower programming productivity and code portability. Finally the debugging procedure changes; each run may be different from all the previous ones because it highly depends on the relative load of the cooperating cores and the system scheduler. This alone makes error discovery and correction a difficult task.

1.3 Programming in Parallel

Programming models represent an abstraction of the capabilities of the hardware to the programmer. A programming model is a bridge between the actual machine organization and the software that is going to be executed by it. A programming model is usually not tied to one specific machine, but rather to all machines with architectures that adhere to the abstraction. A successful parallel programming model needs to combine some diametrically opposite properties. On one hand, it has to be

simple and as close to serial languages as possible, in order to keep productivity and code portability in accepted levels. On the other hand, it needs to allow expressing parallelism in a meaningful way to fully exploit the underlying hardware.

The literature includes a significant number of parallel programming models for several types of parallel systems. The standard approach to exploit the multiplicity of the available compute units is to divide the computation of a program among multiple *threads*. A thread is a stream of instructions that can be scheduled to run independently, and is also known as a 'lightweight' process. A typical parallel program spawns a number of concurrent threads (usually equal to the number of CPUs or cores) which cooperate in order to complete the computation. Another important building block, common in many programming models is the *task* abstraction. In general terms, a task is a portion of code that can be scheduled for asynchronous execution. A task is accompanied by its data environment on which it will operate when executed. There may be all kinds of dependencies (for example data, timing, synchronization) between tasks. Concurrent threads either explicitly or implicitly undertake the execution of tasks.

The range of the applications that can benefit from parallel hardware is with no doubt enormous, and each type of application carries a different set of characteristics. Thus, providing a general programming model to express parallel units of work in a friendly and productive way is a hard problem. OpenMP [6] is a very intuitive parallel programming model which can help in dealing with the above issue. OpenMP is the standard programming model for shared memory multiprocessors and is currently expanding its applicability beyond HPC. It is a directive-based model whereby simple annotations added to a sequential C/C++ or Fortran program are enough to produce decent parallelism without significant effort, even by non-experienced programmers.

1.4 Contributions

This dissertation deals with the difficult problem of designing and implementing a productive and performance-oriented parallel programming infrastructure. The first group of contributions refers to the efficient support of the OpenMP tasking model and its application to provide a novel solution to the problem of nested parallelism. In particular:

Ia. We design a general tasking subsystem in the context of the ompi [7] compiler. We present the necessary code transformations and optimizations performed by the compiler and the architecture of the runtime system that supports them. Among the novel aspects of our system is the utilization of *fast* execution path, which allows speedy task execution based on various conditions. This contribution has been presented in [8].

Ib. Portions of the tasking runtime were re-engineered to match the modern scalable systems: The deep cache hierarchies and private memory channels of recent multicore cpus make such systems behave with pronounced non-uniform memory access (numa) characteristics. To exploit these architectures our runtime system is reorganized in such a way as to maximize local operations and minimize remote accesses which may have detrimental performance effects. This organization employs a novel work-stealing system which is based on an efficient blocking algorithm that emphasizes operation combining and thread cooperation in order to reduce synchronization overheads. Our experiments show impressive performance benefits as compared to other OpenMP implementations, and have been demonstrated in [9].

Ic. We show how the tasking subsystem can be used to handle difficult problems such as nested parallelism. We provide a novel technique, whereby the nested parallel loops can be transparently executed by a single level of threads through the existing tasking subsystem. A similar, albeit limited effort has been made recently in OpenMP v4.5 with the taskloop directive. Apart from its limited applicability, this directive requires programmer actions in contrast to our proposal. Our technique was presented in [10] and has many applications; for example it can be used to provide nested loop parallelism even in platforms with limited resources e.g. embedded systems.

It is our thesis that OpenMP can form the basis for an attractive and productive programming model for multicore embedded systems. However providing OpenMP facilities on such devices is far from straightforward. Embedded systems include diverse groups of relatively weak cores, usually connected with a few powerful cores. In addition, it is highly likely that these systems come with a partitioned address space and private memories, leaving portions of memory management as a programmer responsibility. Finally, embedded systems usually have limited resources, as for example small-sized fast local memories. The second group of contributions is related to the design and implementations of efficient OpenMP infrastructure for embedded and heterogeneous multicore systems.

IIa. We present the first implementation to provide OpenMP support in the sthorm platform [2]. The sthorm consists of a set of low-power general-purpose processing elements (pes), working in full mimd mode, interconnected by an asynchronous Network-on-Chip (NoC). In addition, the architecture allows the inclusion of special-purpose processing cores to fit particular application domains. The system is completed with a dual-core arm and 1GiB of memory. In our view, this will be a reference architecture for future multicore accelerators as it combines the ability to perform general-purpose computations alongside with specialized hardware, while also offering a scalable interconnection structure that will allow large core counts. To exploit the processing power of both the host cpu and the sthorm accelerator, we propose the simultaneous execution of OpenMP constructs on *both* processors, for the first time. Two distinct runtime libraries are used for the host and the sthorm and code may be offloaded from the former onto the latter through simple directives. Our system was presented in [11]. OpenMP v4.0 came to follow a similar model [12].

IIb. As a second case study we present the design and implementation of the OpenMP device model [12] for the Parallella-16 board. It is the first OpenMP implementation for this particular system and also one of few OpenMP 4.0 implementations in general. Parallella is an open source project and its processing power comes from a dual-core arm cpu and a 16- (or 64-core) embedded accelerator, named Epiphany. The accelerator delivers up to 32 gflops (102 gflops, for the 64-core version) and is based on a 2D-mesh NoC of tiny, high performance, floating-point capable risc cores with a shared global address space. We discuss the necessary compiler transformations and the general runtime organization, emphasizing the important problem of data environment handling. We also present the design and implementation of the runtime system that provides the necessary support for the host and the device parts. This work is presented in [13, 14].

IIc. Finally, we propose a novel runtime organization designed to work with an OpenMP infrastructure; instead of having a single monolithic runtime for a given device, we propose an adaptive, compiler-driven runtime architecture which implements only the OpenMP features required by a particular application. To the best of our knowledge, we are the first to propose a compiler assisted adaptive runtime. More specifically, the compiler is required to analyse the kernels that are to be offloaded to the device, and to provide metrics which are later used to select a particular runtime configuration tailored to the needs of the application. This way the user's

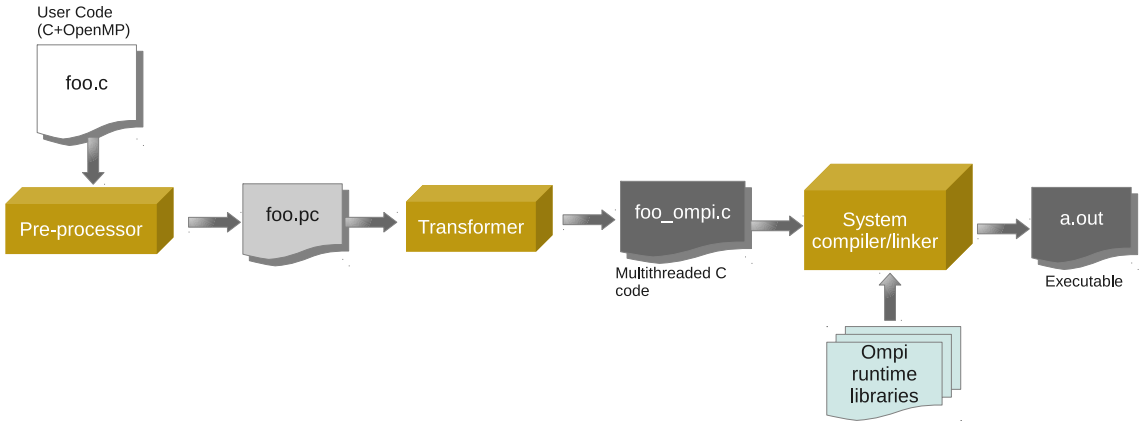


Figure 1.1: OMPi compilation chain

code implies the choice of an appropriate optimized runtime which may result to dramatically reduced executable sizes and/or lower execution times. This work can be found in [15].

1.5 The OMPi Compiler

All our proposals have been implemented fully as parts of an actual OpenMP infrastructure, in particular the OMPi compiler. OMPi is an experimental, open source, lightweight OpenMP infrastructure for the C language that conforms to V3.1 [16] of the specifications. Its compact and modular design accompanied with the detailed documentation, expandability and competitive performance, make it suitable for experimenting with new features and functionalities. OMPi consists of a source-to-source compiler and a runtime library. The compilation process is shown in Fig. 1.1. The compiler takes as input C code with OpenMP directives, and after the preprocessing and the transformation steps, it outputs multithreaded C code augmented with calls to its runtime libraries. This file is ready to be compiled by any standard C compiler in order to produce the final executable.

The runtime system of OMPi has a modular architecture, as shown in Fig. 1.2. It is composed of two largely independent layers. The upper layer (ORT) carries all required OpenMP functionality; this includes everything needed for the orchestration of an executing application, such as the bookkeeping of all thread teams data and internal control variables, scheduling of loop iterations, etc. There exist modules for the management of the teams of threads, for synchronization constructs, for scheduling

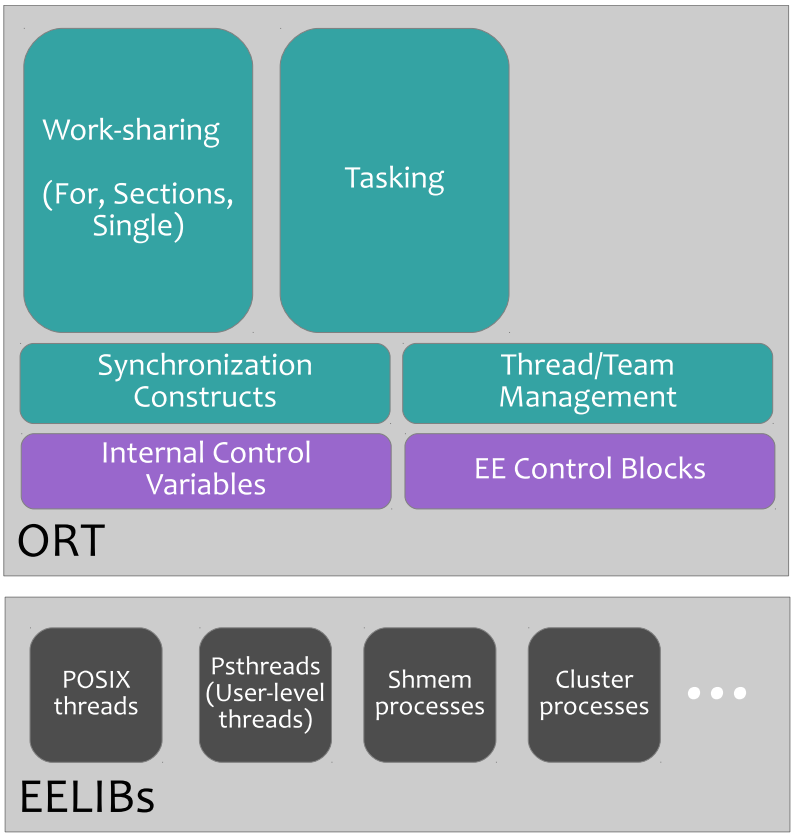


Figure 1.2: Runtime architecture of OMPi

worksharing constructs as well as the tasking infrastructure. The ORT layer operates by controlling a number of abstract *execution entities* (EEs) provided by the lower layer (EELIB). This way the OMPi architecture allows experimentation with different threading libraries. In particular, the lower layer is responsible for the EEs, along with some synchronization primitives. A significant number of EELIBs is available. The default one is built on top of POSIX threads, while there also exists a library which is based on high-performance user-level threads [17]. There are also libraries that utilize heavyweight processes using either SysV shared memory or MPI over clusters of multicore nodes.

1.6 Dissertation Structure

This dissertation is organized as follows:

* In Chapter 2 we discuss related work.
* Next, in Chapter 3 we present our contributions related to the OpenMP tasking

model. We discuss the code transformations, the runtime architecture as well as the optimized code transformations employed by the compiler. We then present a novel runtime organization targeted towards NUMA systems.

* In Chapter 4 we show how to employ a tasking subsystem to execute nested parallel loops.

* The runtime support for multicore embedded systems is described in Chapter 5. We give an overview of our two target accelerators, namely STHORM and Epiphany. Then we present the design, the implementation and the experimental results regarding the OPENMP support for these platforms.

* In Chapter 6 we discuss a novel modular design for the OMPI compiler that meets the new requirements of the OPENMP heterogeneous programming model. We deal with the device-agnostic parts of the infrastructure and propose a novel approach for handling the device data environments.

* A novel flexible compiler-assisted runtime that adapts to the requirements of a given application is presented in Chapter 7. Here we describe the kernel analysis made by the compiler in order to extract a set of metrics. Then we describe the way a mapper/selector handles these metrics, to provide an optimized library tailored to the needs of the specific kernel.

* Finally in Chapter 8 we present the conclusions of this dissertation and some thoughts for interesting future work.

Chapter 2

Related Work

In this chapter we present a general overview of the parallel programming landscape. This landscape is vast, so our purpose is not to give a complete survey. We focus on languages that extend established sequential programming languages such as C, C++, Java or Fortran. We present past and current state of the art programming models for general purpose and embedded multicore platforms. Additional bibliographic information related to the work of this thesis is given in each chapter.

2.1 Parallel Programming Languages and Models

Programming models represent an abstraction of the capabilities of the hardware to the programmer [18]. A programming model is a bridge between the actual machine organization and the software that is going to be executed by it. As an example, the abstraction of the memory subsystem leads to two fundamentally different types of parallel programming models, those based on shared memory and those based on message passing. A programming model is usually not tied to one specific machine, but rather to all machines with architectures that adhere to the abstraction. In addition

one particular system may support more than one programming model, albeit often not with the same efficiency. A successful parallel programming model must carefully balance opacity and visibility of the underlying architecture; intricate details and idiosyncrasies should be hidden while features which enable the full computational power of the underlying device need to be exposed. The programming model also determines how program parts executed in parallel communicate and which types of synchronization among them are available.

Since the appearance of the parallel systems, a significant number of models for multithreaded code development has been proposed. Most of these models share some techniques in order to cope with the challenge of programming heterogeneous multicore systems. Popular and widely used languages are taken as a base and are enhanced with extensions to provide parallel expressiveness. Working with familiar languages makes it easier and more productive for programmers to make the transition to parallel systems execution environment. Furthermore, the huge amount of legacy algorithms written in those languages can be parallelized in a more efficient way.

Development of a parallel application requires the segmentation of the workload into coarse- or fine-grain parts that can be executed concurrently. It is one of the programmer responsibilities to identify the portions of the code which can run in parallel and this procedure may become difficult depending on the application. On the other hand the programming environment has to provide the programmer all the necessary tools to express the parallelism of an application in the best possible way. The use of lower level facilities that are close to hardware offers higher level of control and also performance gains. Nevertheless, nowadays higher abstraction level languages begin to win the race due to their ease of use, higher productivity and increased portability. Higher abstraction level languages include constructs to parallelize `for`-loops, or to annotate parts of the code that are eligible to run concurrently as separate tasks. When referring to a heterogeneous system (a system that contains processing units of two or more different types), these tasks can be either executed on the main (multicore) processor or on an accelerator. Thus special constructs provide the programmer the ability to offload parts of code and data to a co-processor for execution. After the identification of the parallel regions takes place, then a different set of constructs is used to denote dependencies between the different parts of the code that can run in parallel and to dictate their synchronization. This way during

the compile- or at run-tine a dependency graph is created which implies the execution order of all parallel eligible parts of the code. The scheduling mechanisms may utilize the dependency graph in order to distribute workload to the available execution entities. In the rest of this section we focus on some well known parallel programming models and languages.

2.1.1 Cilk

Cilk [19] is a parallel programming language introduced in 1994, targeting shared memory systems. Cilk was later commercialized as Cilk++ by Cilk Arts company. That company was later acquired by Intel, which increased its compatibility with existing C and C++ code, calling the result Cilk Plus. It is based on the well known C language where four keywords where added to denote parts of code (tasks) that can be executed in parallel along with a synchronization mechanism. Cilk became very popular due to its simple interface and high performance runtime support. Some of its innovations where the "work-first" principle and the work-stealing algorithm. In short, the "work-first" principle states that the scheduling overheads of the tasks, related to thread synchronization and racing for task execution, have to be as minimum as possible.

During the execution of a program, a bunch of threads is formed. These threads create tasks when they encounter the `spawn` keyword, and execute them in parallel in certain synchronization points. Each thread is equipped with a queue that stores tasks ready for execution. The scheduler follows the depth-first policy, that is when a thread creates a new task, it suspends the execution of its current one, saves it on the top of its ready-queue, and begins the execution of the new task. The suspended task can be resumed later. In the case where a thread is idle, i.e. it has no tasks in its queue, it becomes a *thief*, steals suspended tasks from the bottom of other queues and executes them. The selection of the victim, i.e. the queue where the tasks are to be stolen, occurs in random order.

2.1.2 Kaapi

Kaapi [20] is a parallel programming environment targeting applications executing on clusters containing multicore CPUs. It is designed to support applications that follow the data-flow model. In the data-flow model the application is designed based on the different states of the corresponding data and the synchronization is achieved

through data dependencies. Kaapi creates a special process for each multicore CPU within the cluster. This process coordinates the kernel-level threads executing in the corresponding CPU. Each task is represented by a user-level thread. Kaapi follows the M:N thread model, where M user level threads are executed by N kernel-level threads. Each kernel-level thread executes all the tasks assigned to it, until completion or until the task is blocked. If it ends up idle then it initially applies for stealing a task from a nearby processor and if that do not succeed then it forwards the request to a more distant one.

The data flow programming model is based on the notions of the data-flow architecture that was pioneered in the 80's through influential architectures which included the MIT Arvind machine [21] and the Dally's J Machine [22]. Data-flow architecture [23, 24] directly contrasts the traditional von Neumann architecture. In data-flow architectures there is no program counter, and execution of instructions is solely determined based on the availability of input arguments to the instructions, so that the order of instruction execution is non deterministic.

2.1.3 Intel Threading Building Blocks

In 2006, two years after the introduction of its dual-core processors, Intel presented the Threading Building Blocks (TBB) [25] library. TBB is a template library for C++ and it offers various parallel data structures and algorithms for developing multi-threaded applications. Similarly to Cilk, the programmer uses some directives to mark parts of code that can be run in parallel. Examples of these directives are: parallel_for, parallel_reduce, parallel_while, parallel_do, etc. The runtime system transforms the work of the parallel regions into a set of tasks that must be dynamically assigned to the processors. During the execution of a program, graphs of task dependencies are created and synchronized to provide the proper scheduling. TBB library also includes a work-stealing algorithm for the tasks.

2.1.4 Satin

Satin [26] is a Java library for programming grid computing applications. Satin programs follow the "Divide and Conquer" model and are executed in wide area cluster computing systems. These systems are formed by heterogeneous platforms and mixed network infrastructures. Thus, special care is taken for the synchronization and co-

operation of the processors. Satin includes a set of keywords that denote the "Divide and Conquer" jobs. The runtime system is responsible to divide the program into tasks that can be run in parallel and to distribute these tasks to available resources. To extend the usage of Satin to heterogeneous systems, a low level library called Ibis was included in the infrastructure. Ibis offers services such as communication, cluster topology information, manipulation of the available resources etc.

2.1.5 Jade

Jade programming environment [27] was introduced in 1992 and targets both shared memory and message passing multicore systems. It is an extension of C language with new directives for declaring shared objects, for denoting code segments that can be run in parallel (tasks) and for defining dependencies between the data accessed by these tasks. The runtime system dynamically creates the task dependency graph and schedules the tasks to the available processors for execution. The language offers restricted parallelism expressiveness. The programmer cannot intervene to the runtime decisions and make low level optimizations, instead it must rely on automatic parallelism extraction.

2.1.6 EARTH-C

EARTH-C [28] is parallel programming language designed for developing applications for shared memory systems by extending serial programs written in C. The popular fork-join model is used and the new directives can be used to define the way code segments can be parallelized. The programmer can provide optimizing hints about the computation locality, for example it may propose candidate processors that can execute specific parts of code. The runtime system takes as input the code along with the optimization hints and it divides the program to groups of commands called execution threads. Each execution thread is represented as a node in the application dependency graph, where edges represent the dependencies between them. The threads are built in such a way so that the active ones could make independent computations.

2.1.7 SPLIT-C

Split-C [29] is a parallel extension of the C programming language that supports efficient access to a global address space on distributed memory multiprocessors. In Split-C programs, the compiler takes care of addressing and communication, as well as code generation. The language provides a small set of global access primitives and simple parallel storage layout declarations (spread arrays). In particular, Split-C follows the SPMD (single program, multiple data) model, where the processors execute the same code albeit they may operate asynchronously and on different data. Any processor may access any location in a global address space, but each processor owns a specific region of the global address space. Two kinds of pointers are provided, reflecting the cost difference between local and global accesses. Split-C offers split-phase assignment where a request to get a value from a location (or to put a value to a location) is separated from the completion of the operation (performed by a `sync` call). Additionally to data requests the programmer can trigger explicit data movements to distant locations with a so-called signaling store. An evolution of Split-C is the Unified Parallel C (UPC) language [30].

2.1.8 EM-C

EM-C [31] is another parallel extension of the C programming language designed for efficient programming in the EM-4 processor, a distributed memory multiprocessor which has a hybrid data-flow/von Neumann organization. Data-flow processors are designed to support small fixed-size messaging through the interconnection network. Those messages are interpreted as "data-flow tokes", which carry a word of data and a continuation to synchronize and invoke the destination thread in other processors. EM-C offers a global address space (through constructs for declaring global pointers) which spans the local memories of all processors. Using remote reads/writes the language supports direct access to remote shared data from any processor. The language provides the `parallel sections` construct that concurrently executes each statement in a compound block as a fixed set of threads. Additional constructs provide loop parallelism, such as the `iterate` and `everywhere` ones.

2.1.9 OpenMP

OpenMP [32] (Open Multi-Processing) is an application programming interface (API) that offers a parallel programming environment as an extension of C, C++, and Fortran languages. It is the de facto standard for shared memory systems and it is supported on most platforms, processor architectures and operating systems, including Solaris, AIX, HP-UX, Linux, OS X, and Windows. The API is managed by the OpenMP Architecture Review Board (defined by a group of hardware and software vendors, including AMD, IBM, Intel, Cray, HP and more) and includes a set of directives, library routines, and environment variables. In this thesis we focus our research on the OpenMP to provide an easy to use, albeit efficient model for programming multicore and embedded systems.

The OpenMP ARB published the first specifications for Fortran (1.0) in October 1997. In October 1998 they released the 1.0 version of the C/C++ standard. The execution model of an application follows the fork-join model: Programs start with just one thread, the master. Worker threads are spawned at parallel regions and together with the master they form the team of threads. In between parallel regions the worker threads are put to sleep. The parallelism must be declared explicitly using the set of the directives and speed up is achieved through the worksharing constructs. There, a code block that contains the total work is assigned to the team of threads. Three worksharing constructs were introduced: The `for` distributes the iterations of a loop over all threads in a team; scheduling of the distribution can be customized through special directed clauses. The `single` construct specifies that the enclosed structured block is executed by only one thread of the team. This is useful for set-up work, for example memory allocation and deallocation, etc. The `sections` identifies a non-iterative worksharing construct that specifies a set of jobs (sections) that are to be divided among threads in the team. Each section is executed once by a thread in the team. The API also defined synchronization constructs (`master`, `critical`, `barrier`, `atomic`, `ordered`) as well as the data scoping.

The version 2.0 of the Fortran specifications and the C/C++ specifications were released in years 2000 and 2002 respectively. The first combined version of C/C++ and Fortran specifications came in 2005 with the v2.5. Both these versions offered additions, clarifications and enhancements to the directives of the original version of the API. The power and expressiveness of OpenMP has increased substantially with the

```
void main(void)
{
  int res;

  #pragma omp parallel
  {
    #pragma omp single
    res = fib(40);
  }
}
```

```
int fib(int n)
{
  int n1, n2;

  if(n <= 1) return n;

  #pragma omp task shared(n1)
  n1 = fib(n - 2);

  #pragma omp task shared(n1)
  n2 = fib(n - 1);

  #pragma omp taskwait
  return n1 + n2;
}
```

Figure 2.1: OpenMP example using tasks

addition of tasking facilities. In particular in May 2008, v3.0 of the specifications was released and it included directives that allow the creation of a task out of a given code block. Upon creation, tasks include a snapshot of their data environment, since their execution may be deferred for a later time or when task synchronization/scheduling directives are met. The tasking model of OpenMP was initially presented in [33] and a prototype was implemented in the NANOS Mercurium compiler [34]. Currently, there exist several platforms that provide support of the OpenMP tasking model; these include both commercial (e.g. Intel, Sun, IBM), freeware, and research compilers, such as GNU gcc [35] and OpenUH [36].

The version 3.1 of the OpenMP specifications was released in July 2011. It included enhancements and extensions to the previous version, as for example new reduction operators and additions to the tasking infrastructure; the `final` and `mergeable` clauses that support optimization of task data environments, and the `taskyield` construct that allows user-defined task switching points.

In Fig. 2.1 we present an example of OpenMP code that employs tasks to calculate Fibonacci numbers recursively. The program starts at the `main` function where the initial is the only executing thread. The `pragma omp parallel` directive forms a team of threads that execute the following code block; one thread is allowed to execute the code following the `pragma omp single` directive, while the others wait at the end of

the parallel region. Each call of the `fib` function creates two tasks using the `pragma omp task` directive to recursively calculate the Fibonacci numbers of $n-1$ and $n-2$. Finally, the directive `pragma omp taskwait` is used to ensure that $n1$ and $n2$ have the correct values in order to proceed with the calculation. The tasks will be executed either by the thread that created them, or by any of the other threads.

2.2 Heterogeneous Computing Models

General-purpose graphics processing units (GPGPUs) and accelerators have been recognized as indispensable devices for accelerating particular types of high-throughput computational tasks exhibiting data parallelism. GPGPUs consist typically of hundreds to thousands of elementary processing cores able to perform massive vector operations over their wide vector SIMD architecture. Such devices are generally non-autonomous. They assume the existence of a host (CPU) which will offload portions of code (called kernels) for them to execute. As such, a user program is actually divided in two parts—the host and the device part, to be executed by the CPU and the device correspondingly, giving rise to heterogeneous programming. Despite the heterogeneity, the coding is generally done in the same programming language which is usually an extension of C. This section presents some important programming models for heterogeneous programming.

2.2.1 CUDA

CUDA [37] is a parallel computing platform and application programming interface (API) model created by NVIDIA. It allows software developers to use a CUDA-enabled GPU (built around an array of streaming multiprocessors) for general purpose processing. The initial release was announced in June 2007 and in September 2015 version 7.5 was introduced. The CUDA platform is a software layer that works with programming languages such as C, C++ and Fortran to offer direct access to the GPU's virtual instruction set and parallel computational elements, for the execution of compute kernels. CUDA allows the programmer to define the kernels, that, when called, are executed N times in parallel by N different CUDA threads. Threads are grouped into blocks, and blocks are grouped into a grid. Each thread has a unique local index in its block, and each block has a unique index in the grid.

Threads in a single block will be executed on a single multiprocessor, sharing the software data cache, and can synchronize and share data with threads in the same block. Threads in different blocks may be assigned to different multiprocessors concurrently, to the same multiprocessor concurrently (using multithreading), or may be assigned to the same or different multiprocessors at different times, depending on how the blocks are scheduled dynamically. Performance tuning on NVIDIA GPUs is not a trivial task and requires the following optimizations in order to exploit the architectural features: Finding and exposing enough parallelism to populate all the multiprocessors; finding and exposing enough additional parallelism to allow multithreading to keep the cores busy; optimizing device memory accesses to operate on contiguous data.

2.2.2 HMPP

In 2007 Dolbeau et al presented the HMPP [38] that targeted GPGPUs. Currently, HMPP includes a set of compiler directives, tools and runtime libraries that support parallel programming in C and Fortran in heterogeneous systems. HMPP provides a portable interface for developing parallel applications whose critical computations are distributed, at runtime, over the available specialized and heterogeneous cores, such as GPGPUs and FPGAs. There are directives that address the remote execution of a piece of code as well as the data movements to/from the hardware accelerator memory, in the spirit of the recent version of OpenMP, see Section 2.2.6.

When targeting CUDA-enabled GPUs, HMPP expresses which parts in the application source should be executed on an NVIDIA card. The offloaded code (codelet) is written in CUDA in a separate file, while keeping the original version of the code in the main source file. The developer also uses the NVIDIA provided tools, such as the CUDA runtime library and compiler to program the codelets. As such, the application source code is kept portable and a sequential binary version can be built using a traditional compiler. Furthermore, if the hardware accelerator is not available for any reason, the legacy code can be still executed and the application behaviour is unchanged.

2.2.3 OpenCL

OpenCL [39] (Open Computing Language) is an open programming environment for developing programs that execute across heterogeneous platforms consisting of CPUs, GPUs, DSPs, FPGAs and other accelerators. OpenCL specifies a language based on C99 and provides an API for parallel computing using task-based and data-based parallelism. The first version of OpenCL was published in August 2009 by the technology consortium Khronos Group and in March 2015 the OpenCL v2.1 was announced.

In OpenCL a computing system is a collection of compute devices including the central (host) processor to which a set of accelerators such as GPUs is attached. Each compute device typically consists of several compute units (CU), and each CU comprise multiple processing elements (PEs). For comparison, a CU can be thought as a CPU core but the notion of core is hard to define across all the types of devices supported by OpenCL. A PE can be seen as a SIMD element of the CU. Functions executed on an OpenCL device are called kernels and a single kernel execution can run on all or many of the PEs in parallel. The code for the host and for the accelerators must be written is separate files and special annotations must be made for the functions and data. The OpenCL API allows applications running on the host to offload kernels (set of functions with corresponding data) on the compute devices and manage the device memory, which can be separate from host memory. To maintain portability, the programs are compiled at run-time.

2.2.4 OMPSs

StarSs [40] is a family of research programming models developed at the Barcelona Supercomputer Center. Prototype implementations include: GRIDSs for grid computing, CellSs for the Cell processor, SMPSs for shared memory systems, COMPSs for distributed computing and cloud computing and OMPSs for HPC multicore and heterogeneous computing. The main characteristics of this family are: task-based programming with annotations for data movement directionality, a single flat logical memory space, and generation of a task-dependencies graph which dynamically guides task scheduling.

OMPSs [41] in particular is a programming model that is compatible to OpenMP and, among others, includes extensions to support heterogeneous environments. These

extensions were first introduced in 2009 as a separate model (GPUSs [42]) but now constitute part of OMPSs. It uses two directives: The first one allows the programmer to annotate a kernel as a task, while also specifying the required parameters and their size and directionality. The second one maps the task to the device that will execute it (in the absence of this directive, the task is executed by the host CPU). This directive also determines the data movements between the host and the device memories. The programmer needs to provide the code of the kernel in CUDA or OpenCL. Since the kernel code can be a part of a task, it can be executed asynchronously. Furthermore, OMPSs provides facilities for multiple implementations (versions) of a given kernel which target different devices. At runtime the scheduler can decide which version should be executed, taking into account parameters such as the execution time or the locality of data.

2.2.5 OpenACC

OpenACC [43] (Open Accelerators) is a programming model designed to simplify parallel programming of heterogeneous CPU/GPU systems and it is developed by Cray, CAPS, Nvidia and PGI. These have also worked as members of the OpenMP ARB to create a common specification which extends OpenMP to support accelerators in the v4.0. It is a directive-based model whereby simple annotations added to a sequential C/C++ or Fortran program are enough to produce decent parallelism without significant effort. The program starts in the host CPU and the source code can be annotated using directives to identify the areas that should be offloaded to an accelerator. Thus in contrast to OpenCL, OpenACC supports files with unified code. OpenACC is available in commercial compilers from PGI (from version 12.6), GCC, Cray, and CAPS. In November 2015, the OpenACC version 2.5 of specifications was presented.

2.2.6 OpenMP for Heterogeneous Systems

In July 2013 the version 4.0 of OpenMP specifications was released. In the spirit of OpenACC, it provides a higher level directive-based approach which allows the offloading of portions of the application code onto the processing elements of an attached accelerator, while the main part executes on the general-purpose host processor. Similarly to OpenACC, the application blends the host and the device code portions in a unified and seamless way, even if they refer to distinct address spaces.

V4.0 also adds or improves the following features: atomics; error handling; thread affinity; task dependencies; user defined reduction; SIMD support and Fortran 2003 support. Finally the ARB released the v4.5 in November 2015. This version includes several features of Fortran 2003; additions to loop and SIMD constructs; task loops and priorities; several enhancements to the accelerator directives and other extensions.

Although support for the OpenMP 4.0 device model has been slow to be adopted by both compiler and device vendors, it is gaining momentum. Currently, the Intel ICC compiler [44, 45] and GNU C Compiler, GCC (as of the latest version [46]) support offloading directives, with both of them only targeting Intel Xeon Phi as a device. GCC offers a general infrastructure to be tailored and supplemented by device manufacturers. Preliminary support for the OpenMP `target` construct is also available in the ROSE compiler. Chunhua et al [47] discuss their experiences on implementing a prototype called HOMP on top of the ROSE compiler, which generates code for CUDA devices. A discussion about an implementation of OpenMP 4.0 for the LLVM compiler is given by Bertolli et al [48] who also propose an efficient method to coordinate threads within an NVIDIA GPU. Finally, in [49] the authors present an implementation on the TI Keystone II, where the DSP cores are used as devices to offload code to.

2.3 Multicore Embedded Systems and Their Programming

Embedded systems usually include customized hardware and are traditionally programmed using vendor-specific tools. Higher-level models, as for example OpenCL, are generally lacking. We believe that OpenMP can form the basis for an attractive and productive programming model for multicore embedded systems. However providing OpenMP facilities on such devices is far from straightforward. We are not the first to propose OpenMP as a suitable model for accelerators or multicore embedded systems. Other efforts include [50] where the authors propose OpenMP extensions to provide a high level API for executing code on FPGAS. They propose a hybrid computation model supported by a bitstream cache in order to hide the FPGA configuration time needed when a bitstream has to be loaded. Sato et al [51] implement OpenMP and report its performance on a dual M32R processor, which runs Linux and supports fully the POSIX execution model. In [52] Hanawa et al evaluate the OpenMP model for multicore embedded Systems Renesas M32700, ARM/NEC MPCore, and Waseda

University RP1. Liu and Chaudhary [53] implement an OpenMP compiler for the 3SoC Cradle system, a heterogeneous system with multiple RISC and DSP-like cores. Additionally in [54] double buffering schemes and data prefetching are proposed for this system. In [55] Woo-Chul and Soonhoi discuss an OpenMP implementation that targets MPSoCs with physically shared memories, hardware semaphores, and no operating system. However, they cannot use the fork/join model for the parallel directive due to the lack of threading primitives.

Furthermore, extensions to OpenMP have been proposed to enable additional models of execution for embedded applications. Gonzàlez et al [56] extend OpenMP to facilitate expressing streaming applications through the specification of relationships between tasks generated from OpenMP worksharing constructs. Carpenter et al in [57] propose a set of OpenMP extensions that can be used to convert conventional serial programs into streaming applications. Chapman et al [58] describe the goals of an OpenMP-based model for different types of MPSoCs that takes into account non-functional characteristics such as deadlines, priorities, power constrains etc. They also present the implementation of the worksharing part of OpenMP on a multicore DSP processor. In [59], Burgio et al present an OpenMP task implementation for a simulated embedded multicore platform inspired by the STHORM architecture. Their system consists of doubly linked queues which store the tasks. They make use of task cut-off techniques and task descriptor recycling.

STHORM [2] was formerly known as the ST P2012 platform. The accelerator fabric consists of tiles, called clusters, connected through a globally asynchronous locally synchronous (GALS) network-on-chip. The architecture of a single cluster is composed of a multicore computing engine, called ENCore, and a cluster controller (CC). Each ENCore can host from 1 to 16 processing elements (PEs). A number of programming models and tools have been proposed for the STHORM architecture. Apart from the low-level system software which is rather inappropriate for general application development, STHORM comes with the Native Programming Model (NPM) which relies on the system runtime libraries and forms the base for component-based tools and applications. OpenCL is a major model implemented in STHORM, relying on the system runtime libraries, too. A number of high-level tools have also been ported, targeting the NPM or OpenCL layers. A prominent example is BIP/DOL [60] which generates platform-specific code using abstract application models based on communicating processes.

The second platform we are interested in this thesis is the popular Parallella board [5]. Together with the STHORM, we believe its architecture is representative of the architecture future many-core and MPSoC a platform will have. The Parallella board is a credit-card sized multicore system consisting of a dual-core ARM host processor and a distinct 16-core Epiphany co-processor. All common programming tools are available for the ARM host processor. For the Epiphany, the Epiphany Software Development Kit (ESDK) is available [61], which includes a C compiler and runtime libraries for both the host (EHAL) and the Epiphany (ELIB). As for high level tools, COPRTHR sdk [62], developed by Brown Deer Technology, provides OpenCL v1.5 support for Parallella and allows OpenCL kernels to be executed on both the host CPU and the Epiphany co-processor. In this thesis we present the first implementation of OPENMP for this system.

Chapter 3

OpenMP Tasking

3.1 Background

OpenMP was initially designed to support parallelization of loops found mainly in scientific applications. Since the first version of the specifications many features have been added expanding the capabilities for parallel expressiveness. Particularly in version 3.0 (2008) the addition of *tasks* gave the application programmer the ability to express recursive and irregular parallelism in a straightforward way. The tasking extensions of OpenMP have been implemented in many open-source and commercial compilers such as the well known gcc, the Intel icc, the Oracle suncc and the IBM XL [63] compilers. Some research compilers such as the OpenUH [36] and Nanos v4 [64] also support the tasking interface. The research compilation systems usually consist of a front-end source-to-source translator that takes as input OpenMP source code and outputs a transformed source code augmented with calls to a runtime support library. The library implements the OpenMP functionalities, by exploiting the power of multiple threads. The final program is compiled and linked using a standard, sequential back-end compiler.

The notion of tasking for expressing dynamic and irregular parallelism existed before the introduction of OpenMP tasks. In what follows we describe the examples of the Intel Workqueing Model [65] and Dynamic Sections [66].

Intel Workqueing Model

Since the release of the version 2.5 of OpenMP, it became clear that the model was not well suited for applications working with lists or tree-based data structures and for applications that included dynamic or irregular workloads. To deal with these scenarios the OpenMP community started an open discussion and encouraged its members to propose ideas on improvements for the model. In 1999 Intel proposed the Workqueing Model [65] which was included in its icc compiler in 2002 as an OpenMP extension. In essence this model supports the annotation of partly "schedule-independent" code blocks, represented as tasks. In contrast, the way the "parallel for" or "parallel sections" are scheduled for execution is strict and predefined. The Workqueing model introduces two directives; the `intel taskq` directive which creates an execution environment for a group of tasks, and the `intel task` directive which denotes the code block of a task.

Within a parallel region (where multiple threads execute the same code block) the `intel taskq` directive denotes a code block (the task execution environment) that can be executed by only one thread of the OpenMP team. The remaining threads are synchronized at the end of this block. Inside the block there may be other commands alongside with `intel task` directives, which denote the code of the tasks. Each thread is equipped with a queue accessible by all threads. The thread that meets these directives creates the tasks (data environment and code) and stores them as nodes in the shared queue. Other threads (waiting at the end of the block) can extract the nodes which are stored in a sibling thread's queue and execute the corresponding tasks.

Dynamic Sections

In 2004, Balart et al [66] presented the Nanos Mercurium research OpenMP compiler. In that work they propose a possible extension to the standard that targeted applications with dynamic and irregular parallelism. Their approach, called *dynamic sections*, is based on the `omp sections` and `section` directives and they suggest to loosen

their functionality restrictions. In their proposal they allow commands to be placed lexicographically between the definitions of `section` directives. These commands are executed by only one team thread. Furthermore they allow the recursive definition of the same section, for example inside a `while`-loop. The thread that executes the `sections` code block inserts each node that represents a `section` into an internal list shared by the threads of the team. The remaining threads extract the nodes and execute the work.

A comparison between the dynamic sections and the Intel Workqueing Model reveals a direct correspondence. The latter is equipped with the `intel taskq` and `intel task` directives which are replaced by the `sections` and `section` respectively. All the internal functionalities between the two proposals are the same. Nevertheless, the dynamic sections approach has the advantage of expanding the functionalities of already present directives, instead of introducing new ones.

3.2 OpenMP Tasks

The notion of tasks, as independent code blocks that can be run in parallel, provides great amount of flexibility regarding their scheduling and execution. An OpenMP task is a unit of work scheduled for asynchronous execution by an OpenMP thread. Each task has its own data environment and can share data with other tasks. Tasks are distinguished as *explicit* and *implicit* ones. The former are declared by the programmer with the `task` construct, while the latter are created by the runtime library, when a `parallel` construct is encountered; each OpenMP thread corresponds to a single implicit task. A task may generate new tasks and can suspend its execution waiting for the completion of all its child tasks with the `taskwait` construct. Within a parallel region, task execution is also synchronized at barriers: OpenMP threads resume only when all of them have reached the barrier and there are no pending tasks left. Tasks are also classified as *tied* and *untied*, depending on whether resumption of suspended tasks is allowed only on the OpenMP thread that suspended them or on any other thread.

In some application classes, programming in terms of threads can be a poor way to provide concurrency. In contrast, formulating a program in terms of tasks, not threads, can be quite beneficial [67]:

* When exploiting a threading package the programmer maps threads onto the cores of the hardware. Highest efficiency usually occurs when there is exactly one running thread per core. Otherwise, there can be inefficiencies from the mismatch. Undersubscription occurs when there are not enough threads running to keep the cores occupied. Oversubscription occurs when there are more running threads than cores.

* The key advantage of tasks versus threads is that tasks are much lighter weight than threads. This is because a thread has its own copy of a lot of resources, such as register state and a stack.

* The use of tasks offers a great deal of load balancing opportunities to the underlying runtime infrastructure. As long as a program is broken into enough small tasks, the scheduler usually does a good job of assigning tasks to balance load. With thread-based programming, the programmer has to deal with the load-balancing designing.

In the next paragraphs we analyse some issues regarding the implementation of OpenMP tasks and the way some compilers have dealt with them.

3.2.1 Task Scheduling

Task scheduling is a major building block of an OpenMP runtime infrastructure. The scheduling problem has two parameters: a) The description of a multithreaded application which is split into a set of tasks and b) A shared memory multicore system with a given hardware architecture. The goal of the scheduler is to minimize execution time of an application. To accomplish that, the scheduler has to equally assign the workload to the processors so as to minimize their idle times (or in other words perform load balancing). Furthermore, the task assignment has to favour data locality, otherwise the application may suffer from the time consuming operations caused by the cache coherent protocols, degrading the overall performance.

Korch and Rauber [68] study applications that follow the tasking model. The set of tasks are represented by a directed acyclic graph (DAG), where the nodes and the edges represent the tasks and their dependencies respectively. The task scheduling can be seen as the problem of dynamically processing all nodes of a DAG by a limited set of processors, in a way to minimize the total execution time. The latter is proved

to be NP-hard [69]. The fact that the task dependency graph is created dynamically along with a limited number of processors lead to the solution of heuristic methods for the scheduling of the tasks.

The data structure used for storing the pending tasks is an important component of the scheduler, and may have a significant impact on its performance. Here we present examples of data structures which are studied in [68]:

central queue The most trivial solution is the use of a single central queue protected by a lock, where all threads can insert and extract tasks.

randomized task pools An extension to the central queue is the utilization of multiple queues which can reduce the racing between the threads, nevertheless this solution adds complexity.

distributed task pools Another approach is to use a private list for each thread, which eliminates the need for locks but also complicates the load-balancing algorithms.

combined central and distributed task pools A more efficient solution is the combination of a central queue with a set of private queues. A thread starts inserting tasks in its private queue; when this queue becomes full, the thread inserts new tasks in the central queue. Correspondingly, when executing tasks, if the private queue is empty then a thread can try to extract a task from the central.

workstealing pools The last proposal refers to the utilization of private queues for each thread, but in the case where a queue is empty the thread is allowed to steal a task from a sibling's queue. A variation of this solution is to equip each thread with two queues; a private queue where stealing is not allowed and a shared one where other threads can steal from.

The selection of the proper data structure depends on the capabilities and characteristics of the underlying system and the applications that the scheduler targets.

After selecting a suitable data structure, the designer of the scheduler has to decide about the order that the tasks are being retrieved from the thread queue(s); in LIFO (last in first out) order the newest-entered task is dequeued while in FIFO (first in first out) order the oldest-entered task is dequeued. The decision on the actual policy to use must take into account the application characteristics. A usual case for task-based applications it to initially create tasks carrying big workloads, or tasks that hide

recursive creation of other tasks, for example applications that follow the "Divide and Conquer" paradigm. FIFO is the preferred way to steal tasks from a sibling thread's queue [68]. This way an idle thread that steals a task will benefit from by acquiring a larger workload (in other words it will be responsible for a larger part of the task graph). Furthermore the number of steal operations (and the corresponding traffic between the processors) would be reduced. While FIFO order is generally suitable for stealing tasks, when a thread decides to retrieve a task from its private queue, then the LIFO order is preferred, since the system will be benefit from better data locality [68]. Stealing multiple tasks instead of one at a time is another technique that may be used in certain application scenarios.

Clet-Ortega et al [70] propose a set of strategies for choosing an appropriate victim task queue where task(s) will be stolen from.

hierarchical In the hierarchical strategy the search starts from the closest queue in the hierarchical order, determined by the hardware architecture. For example, in the case where each thread has a separate queue, threads will try to steal a task from queues of threads running on the cores of the same processor before looking further.

random The random strategy simply checks randomly for tasks among all queues.

random order In this strategy each thread follows a randomly defined order of queues.

round robin For this strategy threads follow a different, predefined sequence of queues according to the hardware architecture.

producer The producer is a strategy that targets producer/consumer applications, here each thread selects the queue that stores the largest number of tasks.

producer order This is a variation of producer, where threads create a sequence of task queues according to the number of stored tasks.

Duran et al [71] compare the policies for creating OpenMP tasks; namely the depth- and breadth-first policy. In depth-first (DFS), a newly created task will be executed immediately, similarly to the way Cilk does. According to the breadth-first (BFS) policy the creation of a task in nested level N imposes that all tasks in level $N-1$ have already been created (If a task belongs in level is X then the tasks that it creates

belong to level $X + 1$). The results show that creating tasks in depth-first policy is very efficient since the application exploits data locality, a result confirmed also by [72]. The use of depth-first policy is proved more efficient when the application uses untied tasks. When tied tasks are used then the breadth-first policy results in better performance. Also, this policy is preferred in applications where a single thread produces all the tasks while the remaining threads execute them. Since the default in OpenMP is to create tied tasks then breadth-first policy for creating tasks is preferred.

An efficient work-stealing algorithm for multi-socket multi-core shared memory systems is proposed in [72]. It exploits data locality in order to reduce the time consuming steal operations. Threads executed in the same socket form a group and share a queue of tasks. This queue is processed in a LIFO way, so that the threads will be favoured by data locality, since they share some levels of cache memory. In the case where a task queue is empty, then a thread of a group steals a batch of tasks from another queue to supply the whole group, hence reducing the task stealing attempts. The shared queues are protected by locks, an acceptable solution since each lock will be mostly claimed by threads which execute on cores in the same socket, thus sharing same levels of cache. Authors in [70] propose a topology aware tree-based representation of the computing platform produced by the `hwloc` package [73]. Having this representation the runtime can choose the appropriate number of the task queues and their placement level; one queue per thread, one queue per group of threads that share L2/L3 cache or one queue per NUMA node.

3.2.2 Task Cut-Off

The mechanism that decides whether OpenMP tasks are to be stored for future execution or to be executed immediately is called task cut-off mechanism. A large number of pending tasks would increase the parallelism potential by keeping processors busy and favour load balance. On the other hand a smaller number of pending tasks would reduce the costs of creating, storing and extracting/stealing operations (here lie high communication and cache protocols cost). Hence, the goal of task cut-off mechanism is to strike a balance between tasking costs and their parallelization benefits. This can be achieved by using some kind of threshold limiting the creation of pending tasks. When this threshold is exceeded all newly created tasks are executed immediately.

Cut-off thresholds can be either explicitly set by the programmer or provided

transparently from the runtime infrastructure. Some proposed cut-off thresholds include [71, 36]:

total number The system will start executing tasks immediately when the total number of pending tasks exceeds a predefined limit.

nested level The use of the nested level of tasks is proposed as a limit that better fits with applications that recursively create tasks using the "Divide and Conquer" model.

dynamic The dynamic cut-off mechanism utilizes profiling techniques in order to collect information about the task patterns of "Divide and Conquer" applications.

depthmode In this proposal each thread decides whether it would create a pending task or not based on the result of $depth \mod N$, where $depth$ is the nested tasking level and N is a parameter.

As a conclusion, designing an optimal cut-off mechanism is a very difficult procedure since its functionality highly depends on a given application, the input data and the task scheduling. Furthermore, the lack of cut-off mechanism, or the choice of an inappropriate limit would greatly degrade the efficiency of a tasking runtime [74].

3.2.3 OpenMP Compilers

GNU GCC

The GNU Compiler Collection (GCC) is a compilation ecosystem developed by the GNU Project for various programming languages. Support for OpenMP started with version 4.2 while version 4.4 was the first to implement tasking. Versions 4.4.5 and 4.7.2 of GCC were available at the time the work of this thesis was performed. In particular we are interested in the library that implements the runtime part (libgomp), and the description that follows applies to these versions. The tasking system of libgomp utilizes a per-team, lock-protected, doubly-linked list. All threads use this list to store pointers to pending tasks. Furthermore, each thread executing a task T, includes a separate doubly-linked list with pointers to the T's child tasks. This feature turns the task queue into a performance bottleneck when a large number of threads execute tasks, since many of them will try to acquire the lock to create or to execute a task. Tasks are created using the BFS approach, but when the total number of pending

tasks in the team list exceeds a specific limit, then the DFS approach is activated and tasks are executed immediately.

Intel ICC

The Intel Compiler Collection, also known as ICC, is a set of C and C++ compilers from Intel available for Windows, OS X, Linux and Intel-based Android devices. The compilers generate optimized code for 32- and 64-bit Intel and compatible architectures. The last release is 15.0 which includes support for OpenMP v4.1. The tasking library (libomp) of this version utilizes a per-thread, lock-protected queue of predefined size, used to store pointers to pending tasks. Tasks are created using the BFS approach and are enqueued at the head of the queue. In the case where the queue is full, then the encountered tasks are executed immediately (DFS). Work-stealing is also implemented: An idle thread traverses the queues of its siblings one after another, and tries to steal tasks from the tail of their queues.

IBM XL

The family of IBM XL compilers [63] are designed for the PowerPC architecture and support the OpenMP base languages. The implementation is based on POSIX threads, where each parallel team owns a shared queue of predefined size for storing pending tasks. When this queue is full, all new tasks are executed in a serial fashion. The runtime utilizes a recycle list for tasks in order to achieve better memory management. To execute a task, a thread extracts it from the queue and places it in its private execution memory. The tasks are extracted from the queue in a FIFO manner. If the execution of a task is suspended, it remains in the thread's private memory, hence all the tasks are treated as tied. All tasks form a parent-child hierarchy which is utilized in task synchronization points.

OpenUH

OpenUH [36] is a research compiler developed at the University of Houston and supports OpenMP for C/C++ and Fortran 95. It is based on the open source Open64 compiler and utilizes a user-level threading library named PCL (Portable Coroutines Library [75]), to support both tied and untied tasks. Each thread owns two queues for pending tasks. The first is a public one, shared with sibling threads and is used

to store newly created tasks as well as suspended untied tasks. The second is private and is used to store suspended tied tasks.

The task scheduler is influenced by Cilk but it is adapted to better fit the OpenMP model. Task creation occurs in a breadth-first manner while the extraction of tasks from the queues is done in LIFO order. When a thread meets a `taskwait` construct then it first executes the most recently created task, resulting in a depth-first task execution, similarly to Cilk. In task scheduling points (`taskwait`, `barrier`) threads first search in their private queues for suspended tasks; if empty, they resort to their public queue. If all queues are empty, they randomly select a victim queue and try to steal tasks from there. The parent-child relationships are also utilized here to ensure correct synchronization. The cut-off mechanism utilizes the following criteria in order to reduce the number of tasks: the total number of pending tasks, the nesting level of tasks and some upper and lower limits regarding the number of tasks stored in each queue.

Nanos v4

Nanos V4 RTL [64, 71] is an OpenMP runtime library based on the nano threads programming model [76]. It makes use of user-level threads executed on top of posix threads (termed virtual processors). This library constitutes the runtime system of the Mercurium research compiler, which is also based on Open64. The library utilizes three types of queues. The first type is a global queue used to store the user-level threads that can be executed by all virtual processors. Each virtual processor owns a local queue (second type) that stores the user-level threads which can be executed only by this particular virtual processor. Finally, there is a team queue (third type) which enforces a logical grouping of user-level threads that represent the tasks of a team. Upon the creation of a task the runtime system decides whether to execute it immediately or to create a new thread and manage it through the user level runtime scheduler.

Two families of schedulers are implemented: Breadth-first schedulers and work-first schedulers. In breadth-first scheduling, every created task is placed in the team queue and execution of the parent task continues. When a tied task is suspended, it goes to the private queue of the thread which was executing it. A suspended untied task is queued into the team queue. Virtual processors will always try to schedule a task from their local queue first. If it is empty then they will try to obtain tasks from

the team queue. Two access policies are implemented: LIFO (i.e. the last enqueued task will be executed first) and FIFO (i.e. the first queued task will be executed first).

Work-first scheduling tries to follow the serial execution. Whenever a task is created, the creator is suspended and the executing thread switches to the newly created task. When a task is suspended it is placed in the local queue. Again, this queue can be accessed in a LIFO or FIFO manner. When looking for tasks to execute, threads check their local queue first. If it is empty, they try to steal work from other threads. In order to minimize contention they used a strategy where a thread traverses the queues all other threads sequentially, starting by the next thread. The access to the victim's queue can also be LIFO or FIFO. The cut-off mechanism utilizes the total number and the nesting level of tasks.

3.3 Tasking in OMPi

In this section we present our design and implementation of tasks in the context of the ompi OpenMP compiler. The modular architecture of ompi's runtime system allows a wide range of choices for experimenting with OpenMP structures. We discuss the source code transformations and optimizations performed by the compiler and the runtime support for tasks. The experimental assessment with benchmarks demonstrate the efficiency of ompi, which attains highly competitive performance, sometimes surpassing that of commercial OpenMP compilers. The rest of the section is organized as follows: In Section 3.3.1 we give an overview of the ompi compilation environment and present the compiler support of OpenMP tasks. In Section 3.3.2 we discuss the runtime design. Finally experimental evaluation is reported in Section 3.3.4.

3.3.1 Compiler Transformations

The compiler uses *outlining* [77] to move the code residing within a `parallel` or a `task` region to a new function and then, depending on the construct, inserts calls to create a team of threads or a task to execute the code of the new function. In more detail, upon encountering an OpenMP `task` construct, the compiler moves the code residing within the `task` region to a new function. Each task is a block of code that may be executed either directly or asynchronously at a later time. As such, its data environment must be captured at the time of task *creation*. Consider the following general OpenMP task

statement, which provides code to be executed as a task, requiring a snapshot of variables `x` and `y`. The `if` expression forces immediate execution if `cond` evaluates to false.

```
#pragma omp task if(cond) firstprivate(x,y)
{
   <CODE>
}
```

An outlining of the code produced by the compiler is shown in Fig. 3.1. The original code gets replaced by lines 2–12. In essence, there is a new structure declared that captures the data environment (firstprivate variables) that will be used by the task, and then the runtime system is instructed to simply create a new task (`ort_new_task()`) or create it and execute it immediately (`ort_new_task_exec()`), depending on the given condition (`cond`). The actual task code is moved to a new function (`taskFunc0()`), shown in lines 15–31. The task function has exactly one argument, its data environment. Notice that the data environment is allocated at line 5, using `ort_taskenv_alloc()`, and the size (in bytes) of the environment. The data is deallocated at the end of the task function, through `ort_taskenv_free()`.

Optimized Execution Path

It should be clear from the above transformation that preparing a task for execution incurs the overheads of

* allocating the data environment
* invoking a driver function (`ort_new_task()`)
* calling the actual function that includes the original code (`taskFunc0()`)
* copying the data environment
* freeing the data environment
* management of runtime queues that store pending tasks
* task scheduling overheads

Such overheads become significant especially for very fine-grain tasks. There may be cases (e.x. the fibonacci sequence recursive computation) where the actual task

```
1  /* Tranformed code */
2  {
3    struct taskenv { int x; int y; } *tenv;
4
5    tenv = ort_taskenv_alloc(sizeof(struct taskenv));
6    tenv->x = x;
7    tenv->y = y;
8    if (cond)
9      ort_new_task(0, taskFunc0, (void *) tenv);
10   else
11     ort_new_task_exec(0, taskFunc0, (void *) tenv);
12 }
13
14 /* Produced task function */
15 void taskFunc0(void *tdata)
16 {
17   struct taskenv { int x; int y; } *tenv = tdata;
18   int x = tenv->x;
19   int y = tenv->y;
20
21   <CODE>
22
23   ort_taskenv_free(tenv, sizeof(struct taskenv));
24 }
```

Figure 3.1: Source code transformation for tasks

```
1  if (!cond || ort_task_throttling())
2  {
3    int _fy=y, y=_fy, _fx=x, x=_fx; /* Capture initial values */
4    ort_task_immediate_start();
5    <CODE>
6    ort_task_immediate_end();
7  }
8  else
9    <standard transformation of Fig. 3.1>
```

Figure 3.2: Optimized code for immediately executed tasks

workload (sum of two integers) is smaller when compared to the task creation/execution overheads. For these kind of computations a bad runtime design would result to parallel programs that perform worse when compared to the corresponding serial alternatives. While one cannot avoid the task management overheads in the general case, we introduce an optimization so as to alleviate most of them. Our implicit task cut-off mechanism is triggered when either the condition `cond` evaluates to false, or the runtime system has a sizable backlog of tasks waiting for execution, and *throttles* the creation of new tasks. Our goal is to minimize the execution overheads of the tasks that are to be executed immediately. To achieve this, we redesigned the compiler to produce a new style of transformed code for the task generation. The redesigned transformed code is augmented with the *fast execution path* which holds a copy of the original code verbatim. The transformation is shown in Fig. 3.2; local variables are declared to capture the firstprivate ones and the original code is executed unmodified. The two runtime functions (`ort_task_immediate_start/end()`) are required for internal bookkeeping.

Notice that the optimization introduces code duplication, thus increasing the size of the executable. This may be an undesirable situation, especially in embedded platforms. For size-critical applications, we provide a compilation flag that disables the production of the fast execution path.

3.3.2 Runtime Organization

This section describes our design and implementation of tasking support in the runtime system of the OMPi OpenMP C compiler, based on the default POSIX threading

library.

Key Structures and Memory Management

The most important data structure is the task 'node', which holds the task descriptor, i.e. all the information needed for the execution of a task. Every OpenMP thread owns a list where task nodes can be stored. This list is called task_queue and task nodes that are stored there represent tasks whose execution is pending. task_queues are implemented as special double-ended queues (deques); the owner thread can add and remove elements from the top of the queue, while any other thread can only extract elements from the bottom of the queue. In addition, ompi makes use of per-thread recycle bins which store task node structures that have been deallocated. This allows task nodes to be reused, leading to better memory utilization and faster allocation.

When preparing for a new task, the need for memory allocation is two-fold:

a) allocate a new task node

b) allocate space for the task data environment

The latter refers to memory allocated by the compiler-produced code (see Fig. 3.1, line 5). In contrast to the task node structure, which has a fixed size, the size of task data environments varies, depending on the size of the captured firstprivate variables of each task. However, the different memory sizes needed for task data environments are few in number, and equal to the number of lexical task regions in the program. Thus, the memory allocator is required to handle only a few different memory sizes. As a result, the allocator used in `ort_taskenv_alloc()` (see Fig. 3.1) was designed as a list of buckets; each bucket stores memory regions of a particular size. Given the size of the task data environment (say s), the allocator searches the list until it hits the bucket responsible for size s. If such a bucket does not exist, it gets created and a certain number of memory regions are allocated and added to the bucket. Finally, a memory region out of the bucket is returned. Upon deallocation (`ort_taskenv_free()`), the memory region is returned to a bucket of the specified size, making it possible to recycle memory regions for subsequent tasks.

Task Scheduling and Work-Stealing

Our task scheduler is based on work stealing [78], whereby idle threads try to execute tasks created by other threads. After a new task is created, it is placed in a thread's queue until some thread decides to execute it. TASK_QUEUEs have fixed length, i.e. they can store up to a certain number of pending tasks. This number is one of OMPi's runtime parameters, controlled through an environmental variable (`OMPI_TASKQ_SIZE`). The manipulation of TASK_QUEUEs utilizes a highly efficient lock-free algorithm based on [79]. Because the original algorithm implements an unbounded deque, we have modified it in order to handle our fixed-length TASK_QUEUEs. The modifications included new functions to handle enqueue requests for a full queue and code adjustments to the original functions in order to simulate an unbounded queue from one that has a fixed-length.

When a thread is about to execute its implicit task (parallel region), a new task node is allocated, the code of this task is executed immediately and finally the task node is deallocated. Whenever a thread reaches an explicit `task` construct, it can either allocate a new task node and submit the corresponding task for deferred execution, or it can suspend the execution of the current task and execute the new task immediately; the default behaviour is to choose the former. That is, we implement a *breadth-first* task scheduling policy (BFS). We resort to the second alternative (*depth-first* task execution) when the TASK_QUEUE is full. In that case the thread enters *throttling* mode, where every encountered task is executed immediately to completion. Notice that in this case the current task (although temporarily suspended in favour of the new task) does not enter the TASK_QUEUE, so it can never be resumed by another thread. In effect, all tasks are *tied*.

A thread's entrance in throttling mode is one of the runtime objectives. However, a thread operating in throttling mode does not produce deferred tasks, which results in a reduction of available parallelism. To strike a balance, before a throttled thread executes a new task, it checks its TASK_QUEUE free space. If the queue has become at least 30% empty then throttling is disabled and task creation policy returns to breadth-first. As shown in Fig. 3.3 each entry in the TASK_QUEUE is a pointer to a task descriptor (Td), which stores all the runtime information related to the task execution as well as the task data environment. The descriptor is obtained out of the thread's descriptor pool. This pool contains an array of pre-allocated descriptors (in

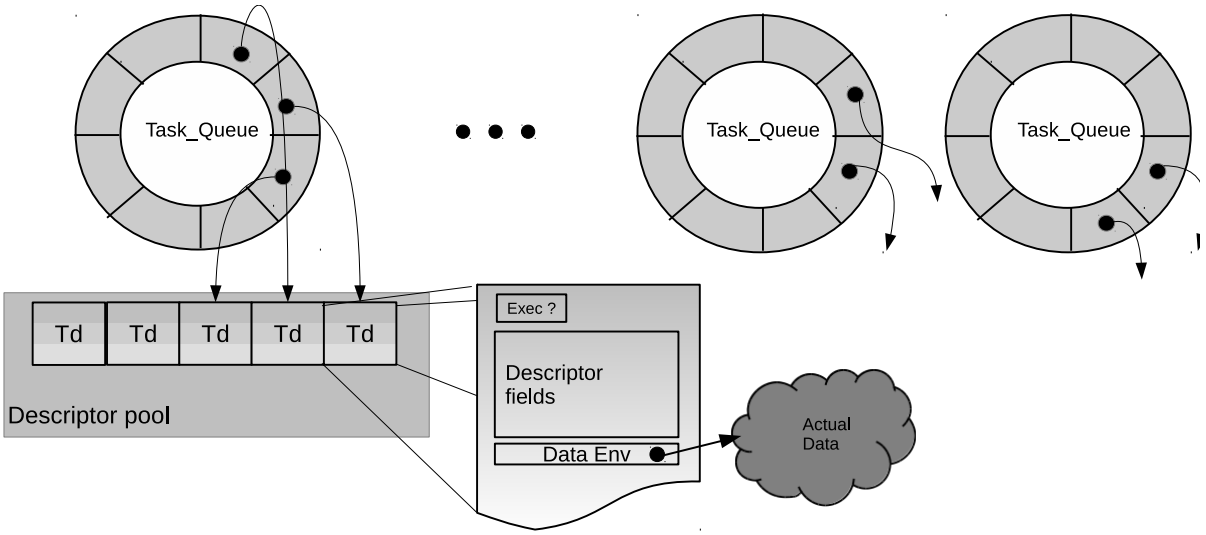


Figure 3.3: Task queues organization. Each thread owns a circular queue(TASK_QUEUE) where pointers to task descriptors (Td) are inserted. Each Td carries bookkeeping information, a special flag (Exec) and a pointer to the task data.

order to speed up the allocation process) and a dynamic overflow list for the case the array becomes empty. Whenever a task finishes its execution, the corresponding Td is returned to a descriptor pool, recycled for future reuse.

The execution of pending explicit tasks takes place in two different cases. The first occurs when the thread reaches a `taskwait` construct, where it executes all the pending tasks it has created. When a thread reaches a barrier, it will additionally try to execute pending tasks of every sibling in its team of threads. In both cases a thread repeatedly checks its queue for pending tasks; if there is one, it executes it. Otherwise, it tries to steal and execute a task from a sibling's TASK_QUEUE. To reduce the repeated hits in empty TASK_QUEUEs we used the following steal search policy: Initially a thread traverses the TASK_QUEUEs of its sibling threads serially, starting from the TASK_QUEUE of the thread with the next id in the team. When it finds a non-empty queue it dequeues and executes a task. The next time it searches for a task it tries to steal from the last queue it proved to be non-empty in the past.

For comparison purposes, we have summarized the tasking architecture of well known compilers (discussed in Sections 2.1.1 and 3.2.3) in Table 3.1. We have also included the organization of Cilk as a reference even though it is not compatible with the OPENMP model. The second column shows the different types of queues utilized by each tasking runtime in order to store pending tasks. The different approaches include one or two queues per thread, a global queue per team, a set of hierarchical

Table 3.1: comparison between tasking implementation policies

Compiler	Queue	Enqueue	Steal strategy	Cut-0ff
GCC	Dequeue (team/thread)	BFS/DFS	Random	Total number
ICC	Dequeue (thread)	BFS/DFS	Fixed order	Full queue
IBMXL	Queue (team)	BFS/DFS	Random	Total number
OpenUH	2 Dequeues (thread)	BFS/DFS	Random	Combined
Nanos V4	Hierarchical queues	BFS/DFS	Serial search	Total number/nested level
ompi	Dequeue (thread)	BFS/DFS	Last victim	Full queue
Cilk	Dequeue (thread)	DFS	Random	-

queues or combinations of them. Next, we show the approach taken upon the creation of new tasks; all compilers but Cilk mark them as pending (BFS) and when cut-off is activated they execute them immediately (DFS). In the fourth column we present the order followed by thief threads in the cases where they have to choose a victim queue. The alternatives are random or fixed order, serial search or the last victim technique of ompi. The final column depicts the various cut-off mechanisms employed by the tasking infrastructures. ompi and icc enable the mechanism when the task_queue is full of tasks, gcc and IBMXL use the total number of pending tasks as a limit, whereas OpenUH and Nanos V4 runtimes utilize combinations of the policies presented in Section 3.2.2.

Fast Execution Path

Since OpenMP V3.0, every task is required to keep a private copy of certain user-modifiable variables known as internal control variables (ICVs). When a thread is in throttling mode, as described above, it executes any encountered tasks immediately. Normally, because of the ICVs storage requirements, the newly executed task will suffer all the overheads of task node allocation and deallocation.

As an optimization, when in throttling mode, we follow a lazy policy, whereby no task node is allocated until actually needed. Tasks are executed without a descriptor, as long as there is no access to the ICVs; a new task node will be allocated upon the first modification of the task's ICVs. The optimization is enabled in the code generated by the compiler shown in Fig. 3.2, lines 3-6. The `ort_task_immediate_start()` call is needed to allow the suspended task remember that it is about to start a new task which has no allocated task node.

3.3.3 Final and Mergeable clauses

This work preceded the V3.1 specifications of OpenMP, where the `final` and `mergeable` clauses for the task directive were introduced. The following code is an example of the new clauses:

```
#pragma omp task if(cond1) final(cond2) mergeable firstprivate(x,y)
{
   <CODE>
}
```

If the `cond2` condition in the final clause evaluates to true, the newly created task is declared as final and forces all of its child tasks to become final and to be executed immediately by the encountering OpenMP thread. When a mergeable clause is present, and the generated task is undeferred, the implementation might generate a task whose data environment, inclusive of ICVs, is the same as that of its generating task region. These new clauses provide new possibilities towards explicit task cut-off and allow the runtime implementations to reduce the overheads of task manipulation.

However, the introduction of the mergeable clause does not have a major impact for our task implementation, since we already implement its functionality within the libraries of the fast execution path. Our solution performs this optimization transparently when applicable without requiring extra code from the user. Nevertheless, to support the new clauses we altered the compiler to include another optimized path for the task creation in the transformed code. In Fig. 3.4 we present the transformed code produced by the compiler for the previous code example. The new optimized path for the mergeable clause is shown in lines 1-5. This code is enabled when a task is denoted as final, and it differs to the one executed in the fast path, due to missing calls of `ort_task_immediate_start()\end()`. These calls are not needed anymore because the user specifies that the new task can be safely executed in its parent data environment. What is important here, is the fact that this path incurs no overheads (besides the condition in line 1) regarding the execution of a task.

3.3.4 Performance Experiments

In this section we present experimental results on a server with 4 dual-core Intel Xeon Paxville 3.0GHz CPUs running Debian Linux 2.6. We have used the BOTS benchmark suite [80], which has been developed for testing and evaluating OpenMP task im-

```
1   if (in_final())
2   {
3     int _fy=y, y=_fy, _fx=x, x=_fx; /* Capture values */
4     <CODE>
5   }
6   else if (!cond || ort_task_throttling())
7   {
8     int _fy=y, y=_fy, _fx=x, x=_fx; /* Capture values */
9     ort_task_immediate_start();
10    <CODE>
11    ort_task_immediate_end();
12  }
13  else
14    <standard transformation code for a task>
```

Figure 3.4: Transformed code for tasks that utilizes mergeable and fast execution path

plementations. We present here results for four BOTS applications: Alignment, FFT, Sort and NQueens, but similar results were also found in the rest of the applications. Alignment aligns all protein sequences from a big set against every other sequence using the Myers and Miller algorithm. The alignments are scored and the best score for each pair is provided as a result. FFT computes the one-dimensional Fast Fourier Transform of a vector of complex values using the Cooley-Tukey algorithm. Sort sorts a random permutation of a sequence of 32-bit numbers with a fast parallel sorting variation of the ordinary merge sort. Finally, NQueens computes all solutions of the n-queens problem, whose objective is to find a placement for n queens on an $n \times n$ chessboard such that none of the queens attack any other. It uses a backtracking search algorithm with pruning.

Besides OMPi with the default threading library (POSIX) we included the PSTHREADS user level library. We compare OMPi's results with the results for three freely available OPENMP compilers that support tasks, the Intel C++ 12.0 compiler (ICC), Sun Studio 12.2 (SUNCC) for Linux, and GNU GCC 4.4.5. We have used the default settings of the OPENMP runtime libraries and the $-O3$ optimization flag in all experiments.

Regarding our tasking library, TASK_QUEUE sizes were set to 24; if a thread tries to store more tasks in its queue it switches to throttling mode. In addition, throttling is

Table 3.2: Serial execution time of benchmarks

Compiler	Alignment	FFT	Sort	Nqueens
ICC	55.94	41.94	8.07	9.71
SUNCC	66.44	43.14	7.96	10.51
GCC	55.44	46.73	7.72	9.30

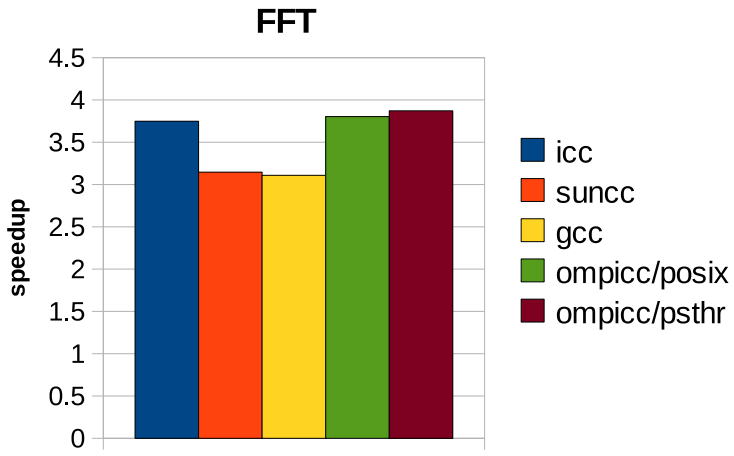


Figure 3.5: FFT (matrix size=32M)

disabled when a thread's queue has more than $24 \times 30\% = 7$ available nodes.

Table 3.2 presents the sequential execution (ignoring OpenMP pragmas) time in seconds of the four BOTS benchmarks with their default runtime parameters. Because we use GNU GCC as the native back-end compiler of OMPi, we use the sequential execution times of GCC as a baseline for calculating the speedups for OMPi. With the exception of SUNCC and the protein Alignment benchmark, the compilers exhibit similar execution times.

Figs. 3.5 to 3.8 present the speedups of the four BOTS benchmarks when running on the 8 cores of our server. Each benchmark was run several times and the average execution time was calculated. For the FFT benchmark (32M matrix size) we observe that for OMPi both threading libraries achieve higher speedup as compared to all other compilers. For Alignment (100 protein sequences) all compilers exhibit comparable performance. As far as the Sort benchmark (32 matrix size) is concerned, OMPi outperforms GCC and SUNCC, but here ICC exhibits the best speedup. Fig 3.8 shows the experimental results of the NQueens benchmark when manual cut-off of task par-

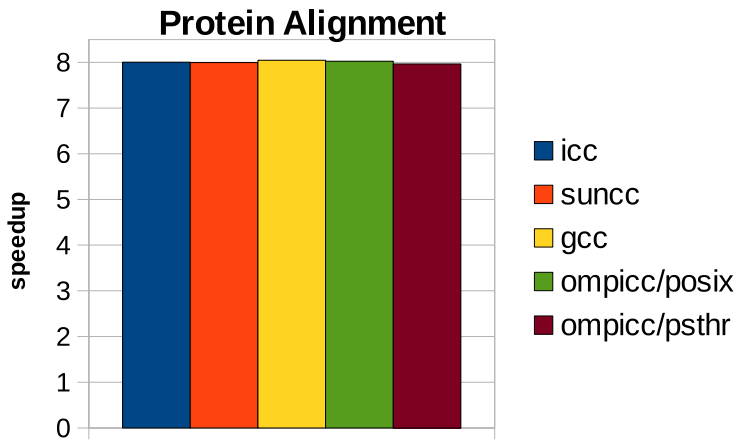


Figure 3.6: Protein Alignment (100 protein sequences))

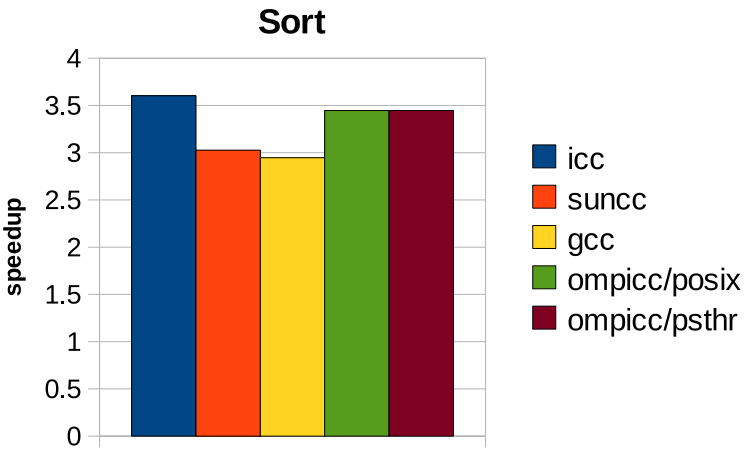


Figure 3.7: Sort (matrix size=32M)

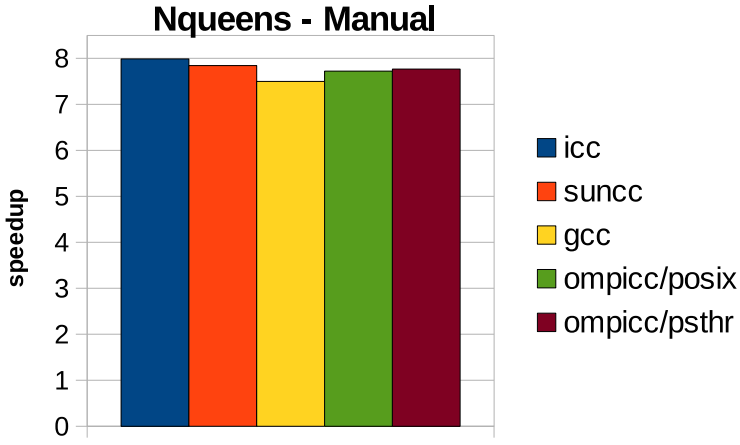


Figure 3.8: NQueens with manual cut-off (board size=14)

allelism is employed by the application itself. These results are for board size equal to 14; we experienced similar behaviour for bigger board sizes. The cut-off value is set to 3 which means that only tasks up to nested level 3 are created. For this particular experiment, ICC and SUNCC exhibit slightly better performance than OMPi, which outperforms GCC. We also measured the performance of the compilers in the case where no cut-off is employed for a board with size 13. OMPi exhibits more that two times better speedup compared to ICC (2.78× over 1.23×). Finally for that scenario the results for SUNCC and GCC are unexpected as the exhibit very poor performance.

3.4 OpenMP Tasking for NUMA Architectures

Many contemporary multi-core systems follow the NUMA (Non Uniform Memory Access) architecture. They usually consist of a small number (2-8) of blocks (NUMA nodes) interconnected with high bandwidth point to point networks that provides access to a logical shared memory. Within each NUMA node there is a relatively small number of cores (2-16) which share an hierarchy of caches and memory. All cores have direct access to the entire memory space but with varied delays, depending on the placement of the data. This means that a core has fast access to the data stored in its NUMA node, but it suffers noticeable delays otherwise. NUMA systems are equipped with multilevel cache memories (usually 3 levels) and a set of complex coherency protocols. Developing applications that fully exploit the capabilities of these systems is a challenging task.

In this section we present the design and implementation of a tasking runtime that targets systems with NUMA characteristics. The general rule is to minimize the delays related to data transfers between the system nodes. Towards this aim, the algorithms and data structures are designed to minimize the access of data located at distant nodes. Furthermore, we avoid the communication based on shared variables for threads executed on different nodes. The runtime functionalities are designed to favour extensive processing of local data (data of the local node) instead of accessing distant data.

In multicore platforms, threads use shared objects in order to achieve mutual exclusion and cooperate throughout a program's execution. Examples of shared objects are locked-protected variables, shared lists, message queues etc. To implement a

shared object, other lower level primitives are utilized, provided either by hardware or software. For example a CAS (Compare and Swap) is an atomic instruction used to achieve synchronization. It compares the contents of a memory location to a given value and, only if they are the same, modifies the contents of that memory location to a given new value. This is done as a single atomic operation. The atomicity guarantees that the new value is calculated based on up-to-date information; if the value had been updated by another thread in the meantime, the write would fail. Nowadays the majority of multiprocessor architectures (e.x. x86, IBM) implements CAS, while systems running on SPARC architectures must implement this instruction in software. This instruction can be used to implement semaphores and mutexes, which in turn is used to protect a shared variable, or a shared list.

A universal synchronization algorithm is a generic mechanism which can be used to implement any shared object, for example a shared counter. The mechanism utilizes some hardware or software synchronization primitives like CAS or FAA (Fetch and Add). It takes as input the sequential implementation of any operation of the shared object (in our example the counter's addition by 1) and simulates a concurrent implementation of it. Thus, the availability of an efficient universal algorithm can provide a thread-safe implementation of any shared object with minimal programming effort.

The work-stealing mechanism is a crucial component of an OpenMP runtime and should thus be designed in a way to be efficient and scalable in cases of high contention. A number of work-stealing algorithms with various characteristics has been proposed, such as Intel TBB's AP/SP and Lazy Binary Spliting [81, 82] which are targeting tasks generated by do-all loops. Cilk's work-stealing infrastructure [78] is another well-known example; however Cilk's runtime is not directly applicable to OpenMP since the latter allows barriers among team threads. To provide an efficient work-stealing mechanism we utilized a Universal Synchronization Algorithm [83], in order to minimize the thread communication and data transfer overheads. We implemented our runtime in the environment of the ompi compiler and tested our system exhaustively. Using a synthetic benchmark we reveal a very significant— up to 6x—increase in attainable throughput (tasks completed per second), as compared to other OpenMP compilers, thus enjoying scalability under high task loads. At the same time applications from the BOTS tasking suite [80] experience reduced execution times (up to 87%) in comparison to the rest of the available OpenMP systems.

The rest of the section is organized as follows: in Section 3.4.1 we present in detail the organization of the optimized runtime system. The work-stealing subsystem, is discussed separately in Section 3.4.2. Finally Section3.4.3 is devoted to the experiments we performed in order to assess the performance of our implementation.

3.4.1 Optimized Runtime

Our runtime organization is based on distributed TASK_QUEUES, one for each OPENMP thread as explained in 3.3. Each entry in the TASK_QUEUE is a pointer to a task descriptor (Td), which stores all the runtime information related to the task execution as well as the task data environment. A key point in our design for the scheduling of the tasks is the aforementioned throttling mode; while in throttling mode all descendant tasks are executed immediately in the context of parent task, favouring data locality.

A task created by a thread might be stolen and executed by another thread in its team. When the task finishes and the descriptor must be recycled, a decision has to be made as to which pool the descriptor should return to. If it enters the pool of the thread that executed the task, severe memory consumption is possible in cases where only few threads create a big number of tasks while the rest execute them. On the other hand, this option is a local operation, enjoying lack of contention. Memory consumption is reduced if the descriptor is put back to the task creator's pool, and this is what we propose. Notice though that synchronization needs arise since threads that stole tasks from the same thread may try to restore it concurrently.

In order to avoid the aforementioned synchronization overheads, we propose a garbage-collecting strategy, shown in Fig. 3.9. Each thread t maintains a private set of pointers (PENDING_QUEUE) to the task descriptors it has created and are either stored for deferred execution or are currently executing. When a task is dequeued for execution (e.g. because a thief stole it), the Td pointer is removed from TASK_QUEUE but remains intact in PENDING_QUEUE. The descriptor contains a special flag ('Exec' in Fig. 3.3). When the task completes its execution, the executing thread sets this flag to announce that the descriptor can now be recycled. On specific occasions thread t traverses its PENDING_QUEUE to find Tds that represent executed tasks and returns them to its pool for future use.

The PENDING_QUEUE plays a central role in the implementation of the `taskwait` and `barrier` constructs, too. Whenever a task meets a `taskwait`, it must wait until

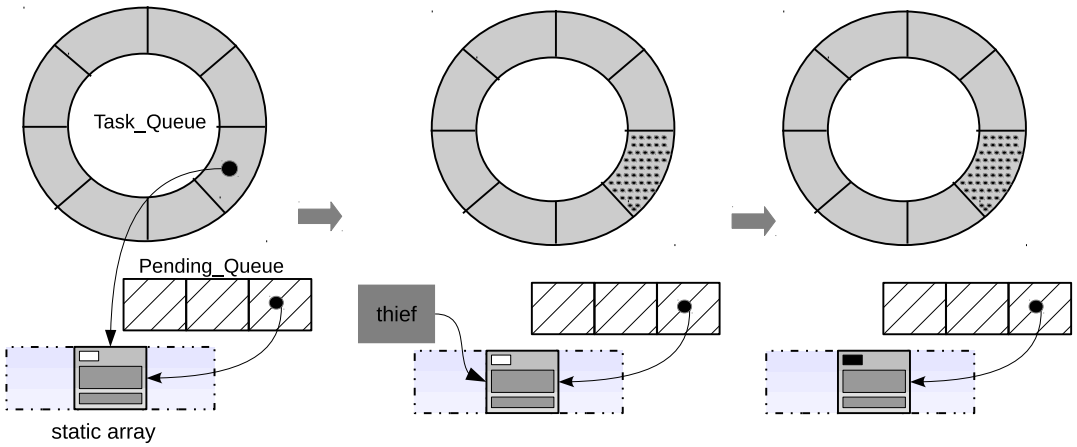


Figure 3.9: Pending, executing (stolen) and finished task. When a task is pending for execution then corresponding entries in TASK_QUEUE and PENDING_QUEUE point to its Td. Upon dequeing, only the link in TASK_QUEUE is removed, freeing one slot. When the task is finished, the executing thread sets the Exec flag to announce that the descriptor can be recycled

the completion of all tasks it created (it is actually then that the task execution and stealing mechanism is triggered). This completion condition is fulfilled simply when all the descriptors in the thread's PENDING_QUEUE have been flagged as executed. Upon meeting a `barrier`, a thread must wait until:

(i) all its siblings reach the barrier and

(ii) all team-generated tasks are executed

For the first condition an atomic counter is employed, getting increased by every thread reaching the barrier. For the second condition each thread contiguously executes/steals pending tasks from all TASK_QUEUE's within its team until all team PENDING_QUEUES become empty.

Our runtime design aims at using as little shared data as possible, so as to reduce atomic operations and minimize thread synchronization. It is worth noting that in our tasking system thread synchronization occurs only in two cases. The first is during the unavoidable barrier construct and the second is during the work-stealing operations, as described in the next section, for which a very fast algorithm is employed. All data structures (e.g. Td's in the descriptors pool) are cache line-size aligned so as to eliminate false sharing phenomena and avoid triggering coherency protocol actions, which deteriorate the performance, especially in NUMA platforms.

3.4.2 A Novel, NUMA-driven, Fast, Work-Stealing Algorithm

In many applications task creation is unbalanced and it is a very common phenomenon few threads to produce many tasks and all other threads to consume them. In such cases contention could be lowered if threads cooperated instead of competed for obtaining the next tasks to execute. In our OpenMP tasking runtime each thread maintains (owns) a task_queue, as explained above. A task_queue is a shared object similar to the shared queue [84] supporting two operations: `OwnerEnqueue` and `Dequeue` for inserting and removing tasks, correspondingly. `OwnerEnqueue`(q, t) inserts a new task t in queue q in case there is enough free space and returns true; otherwise, `OwnerEnqueue` fails and returns false. In contrast to `Enqueue` of a conventional shared queue, `OwnerEnqueue` is executed only by the thread that owns q. `Dequeue` is executed by any thread and removes the oldest inserted task of q. Recently, Fatourou and Kallimanis [83] presented `CC-Synch`, an object which is able to implement (simulate) any shared object very efficiently. For example, to implement a shared queue, it is enough to use one instance of `CC-Synch` and to supply the sequential code for the `Enqueue` and `Dequeue` operations. `CC-Synch` supports only one operation called `ApplyOp`(*sfunc*, *arg*, *th_id*); *sfunc* is the serial code of the operation, *arg* is the argument of the operation and *th_id* is the id of the thread that executes the operation.

In [83], it is shown that `CC-Synch` significantly outperforms the state-of-the-art synchronization techniques. This is a result of the efficient implementation of the *combining* technique whereby, one thread (the *combiner*) holds a coarse lock, and additionally to the application of its own operation, serves the operations of all other active threads. Whenever a thread executes an operation using a conventional synchronization technique (such as spin-locks), it causes cache misses by fetching part of a shared object's state to the local processor cache in order to apply its operation. In the combining technique, only the combiner fetches parts of object's state and applies the operations of all active threads. Therefore, a lot of cache misses are avoided and the communication overheads among processors are much lower.

In the same work ([83]) `H-Synch` universal object was presented. `H-Synch` is a hierarchical approach of `CC-Synch`. It targets numa architectures where each node includes a multicore processor capable of supporting a substantial number of hardware threads. An example of such an architecture is the SUN UltraSPARC T2 cpu. It is proven that `H-Synch` outperforms `CC-Synch` in machines with such characteristics. However, in

this work we target machines where each NUMA node supports a smaller number of threads (< 32) where CC-Synch is more appropriate and exhibits better performance.

Using CC-Synch to implement an operation that is executed only by a single thread in any point of time is rather expensive. Thus, in our work-stealing queue implementation, we designed OwnerEnqueue (which is executed only by the owner of the work-stealing queue) in a way that it does not make calls to ApplyOp. Thus, we avoid making the expensive calls of CC-Synch, wherever possible. It is noticeable that CC-Synch is better suited for cache-coherent NUMA machines, which constitute the majority of modern multicore multiprocessors.

We now give more details for our work-stealing implementation. Our work-stealing TASK_QUEUE (Fig. 3.10) consists of:

(i) a shared array of pointers to TASK structs

(ii) a shared integer *Top* which points to the topmost element of the queue

(iii) a shared integer *Bottom* which points to the bottommost element of the queue, and

(iv) an instance of CC-Synch

Since the OwnerEnqueue operation is executed only by the owner of the queue, no synchronization is needed. Whenever a thread p executes an OwnerEnqueue operation, it firstly executes a read on *Bottom* and after that a read on *Top*. If there exists free space, p inserts the new task and increases Top by one; otherwise, OwnerEnqueue returns false. Since p is the owner of the work-stealing queue and OwnerEnqueue is executed only by the owner, p is the only thread that modifies the shared variable *Top*. Therefore, no synchronization is needed while modifying *Top*.

Whenever p wants to execute a Dequeue operation, it first checks if at least one element exists in the queue and in that case increases *Bottom* by one. Many threads may access *Bottom* simultaneously, since any thread is able to execute Dequeue in any TASK_QUEUE. We implement Dequeue using an instance of the CC-Synch synchronization queue and providing it the serial code for the Dequeue operation. Since CC-Synch is a synchronization technique that serves operations with FIFO order, threads that execute Dequeue operations are also served with a FIFO order. Thus, our implementation satisfies strong fairness properties.

```
typedef struct WSQueue {
   int Bottom, Top;
   TASK *QArray[m];
   an instance of CC-Synch synchronization technique;
} WSQueue;

bool OwnerEnqueue(WSQueue *l, TASK *arg, int pid) {
   int top = l->top, bottom = l->bottom;
   int new_top = (top + 1) % TASKQUEUESIZE;

   if (new_top == bottom) return false;
   else {
      l->QArray[top] = arg;
      l->top = new_top;
      return true;
   }
}

TASK *Dequeue(WSQueue *l, int pid) { // Serial code for Dequeue, the concurrent
   void *ret;                        // version is implemented using CC-Synch.

   if (l->bottom == l->top) ret = NULL;
   else {
      ret = l->QArray[bottom]
      l->bottom = (l->bottom + 1) % TASKQUEUESIZE;
   }
   return ret;
}
```

Figure 3.10: Pseudocode for the work-stealing queue implementation

3.4.3 Performance Evaluation

In this section we evaluate the efficiency of our OpenMP tasking implementation. A synthetic producer/consumer benchmark was used to measure the task creation and the task execution throughput. Furthermore, the Barcelona OpenMP Tasks suite (BOTS) [80] was utilized in order to test our system in a broad range of task applications. All experiments were run on a 16-core machine equipped with two 8-core AMD Opteron 6128 CPUs running at 2.0GHz (two NUMA nodes) and with a total of

```
main()
{
  #pragma omp parallel num_threads(nthr)
    if(omp_get_thread_num() < nprod) {
      for (int i=0;i<16E6/nprod);i++)
        #pragma omp task
        do_random_work();
    }
}
```

```
do_random_work()
{
  volatile long i;

  for (i=0; i<RandomRange(0,maxload); i++)
    ;
}
```

Figure 3.11: Code for synthetic microbenchmark

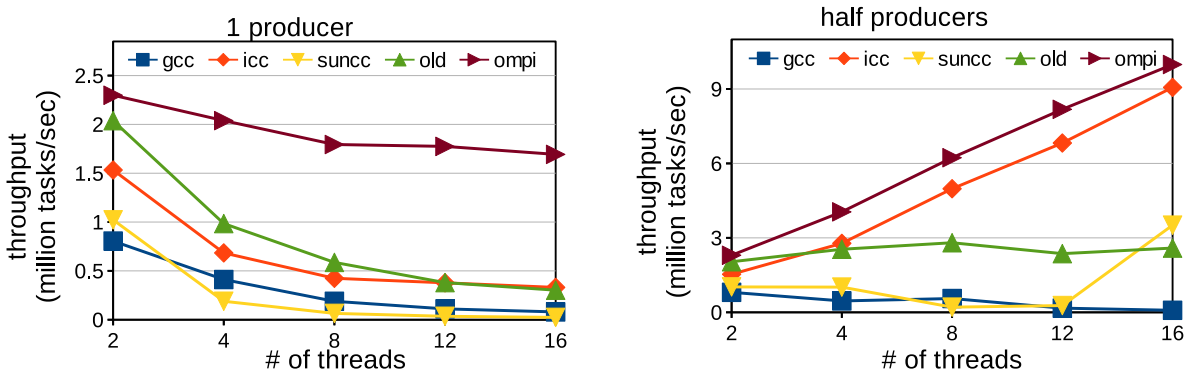
16GB RAM. The system runs Debian Squeeze based on Linux kernel 2.6.32.5. We compare the performance of our compiler with GNU GCC (version 4.4.5-8), Intel ICC (version 12.1.0) and Oracle SunStudio SUNCC (version 12.2). For reference the initial unoptimized implementation of OMPi in [8] is also included, labeled as 'OLD'.

In [71], it is shown that choosing the appropriate limits to enable and disable task cut-off is not an easy task. When dealing with task cut-off, it is required to have good knowledge of application's behaviour for a specified input size, and of the runtime 's tasking implementation. We thus chose to deactivate all manual cut-off techniques in all our benchmarks and let the OPENMP implementation operate under its default settings. As far as OMPi and OLD compilers are concerned, we used the default values for the size of TASK_QUEUES which is 24.

We used GNU GCC with the "-O3" flag as a back-end compiler for OMPi. The corresponding flags for GCC, ICC and SUNCC were "-O3 -fopenmp", "-fast -openmp" and "-fast -xopenmp=parallel". We experimented with a lot of other flag combinations for all compilers but we didn't notice significant performance differences. All experiments were executed twelve times each, then the best and worst runs were discarded; from the ten remaining executions average values were calculated and reported.

Synthetic Benchmark

In order to evaluate the performance of OMPi, a synthetic benchmark with a controllable number of task producers and task consumers was used, as shown in Fig. 3.11. In this benchmark, a parallel region is created and a specified number of threads (equal to `nthr`) is created. Only `nprod` threads become producers and are allowed to create tasks. The rest of threads simply reach the end of parallel region and become consumers (executors) of the created tasks. Each run of the specified benchmark cre-



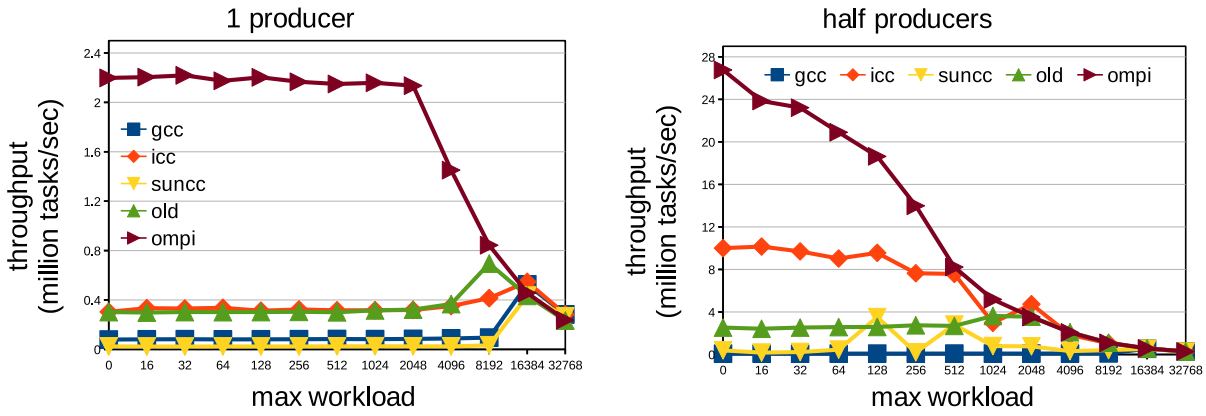
(a) (b)

Figure 3.12: Synthetic benchmark, `maxload`=128

ates 16×10^6 tasks, the creation of which is equally assigned to producer threads. Each task consists of a dummy loop used to simulate workload that a task may have to execute in a way similar to [84, 85, 83]. The number of iterations is a random number between 0 and `maxload`, a variable controlling the task granularity. Iterator variable i is annotated as volatile in order to avoid compiler code elimination optimizations. This benchmark aims to stress the runtime's ability to create, steal and execute tasks.

We run several tests for different values of `nthr`, `nprod` and `maxload`. In Figs. 3.12–3.13 we present each implementation's throughput, measured as the number of tasks completed per second. For Fig. 3.12a we employed one producer and `nthr`-1 consumers. In this experiment `maxload` was chosen to be equal to 128, representing fine-grain work. Some lock-free shared objects show unrealistic high performance when choosing a `maxload` value equal to 0, thus it is a common benchmarking strategy [84, 85, 83] to choose a small value for `maxload`, but not equal to 0. In this experiment, as more threads try to steal from the task queue of the producer, task throughput decreases. This is the result of extra synchronization overhead added, since more threads compete to get shared access to the same TASK_QUEUE. It can be seen that OMPi exhibits up to 5 times higher task throughput (at 16 threads) compared to ICC which is ranked as second best. The situation is similar when compared to the initial OMPi task implementation and the reasons are twofold:

* In the new implementation we redesigned the data structures in order to eliminate false sharing effects. We altered the operations that included thread communication so as to benefit from local operations and we minimized accesses to remote memories in order to reduce the synchronization overheads.



(a) (b)

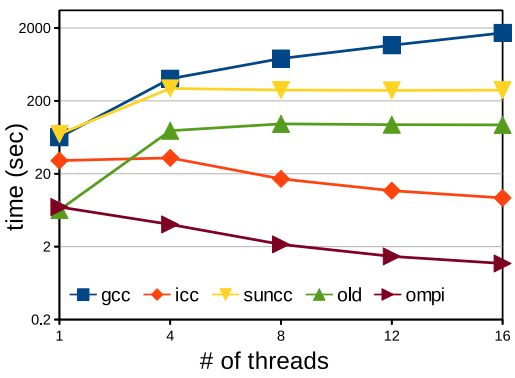
Figure 3.13: Synthetic benchmark, `nthr=16`

* The combining technique used in our work-stealing implementation, emphasizes thread cooperation. A single thread holds a coarse lock on the TASK_QUEUE and along with its `Dequeue` operation it performs the `Dequeues` of its siblings. This way all operations in the shared TASK_QUEUE are executed very fast (almost as fast as a serial execution) and fewer accesses to shared variables are performed.

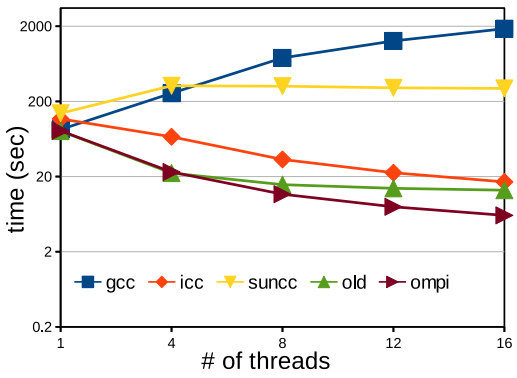
Due to our careful designed mechanisms OMPI outperforms all other compilers even in cases with very high contention and has the best scalability among them. The original OMPI implementation performs well only when 2 threads are used but its throughput quickly decreases. This behaviour is expected since throughout the experiments we observed that OLD suffered from communication overheads during the task creation and execution. In contrast, the new tasking runtime is designed to greatly reduce these overheads.

In Fig. 3.12b, we study the behaviour for different `nthr` values when `nprod=nthr/2`, while `maxload` is still equal to 128. The results are similar, confirming OMPI's superiority.

In Fig. 3.13a, the performance results for different values of `maxload` and for a total of 16 threads (one of which produces tasks) are displayed. In this benchmark, our runtime exhibits higher throughput when compared to all other compilers for almost any `maxload` value. For values of 8192 or less, the work that each task executes is quite small and is overwhelmed by the contention that the work-stealing part induces. Since OMPI exploits the combining technique in its work-stealing queue, the synchronization overheads between threads are vastly minimized and the performance advances a lot. We achieved a little more that 6 times better performance compared to ICC and even



(a) Fibonacci



(b) Nqueens

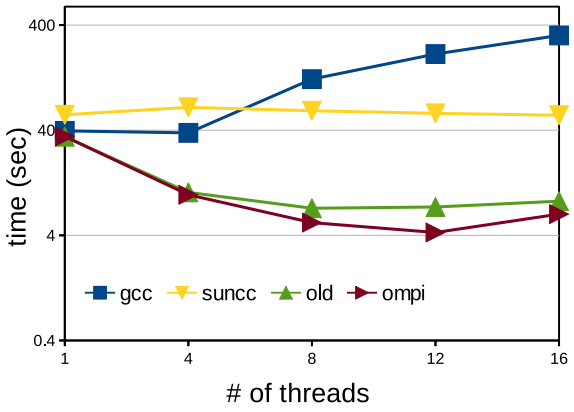


Figure 3.15: Floorplan

better compared to GCC and SUNCC when application produces fine-grain tasks. When the task's granularity becomes coarser (`maxload` values greater than 8192), synchronization overheads between threads are not a bottleneck anymore and all compilers tend to exhibit similar behaviour. Similar observations can be made with the results in Fig. 3.13b, where 8 out of the 16 threads produce tasks for different values of `maxload`. For `maxload` values between 0 and 256 our new runtime achieves from 2.6 to 1.8 times higher throughput than the second best (ICC).

Performance of the BOTS Application Suite

The Barcelona OPENMP Tasks Suite (BOTS) v.1.1.1 was used for evaluating our tasking environment's efficiency in a wide range of tasking scenarios. We present detailed results for the Fib, NQueens and Floorplan applications, while a brief discussion is made for Alignment, FFT, Health, Sort, SparseLU and Strassen. In order for every compiler to have full scheduling opportunities, we run both the tied and the untied

task versions of the applications (while ompi always utilizes tied tasks). We report the best execution times observed, although there were no significant performance differences as noted also in [80].

The Fib application computes the nth Fibonacci number using a recursive paralellization producing a very large number of fine-grain tasks. In Fig. 3.14a, execution time results for the 40th Fibonacci number are shown. Since it was a very common phenomenon for ompi to outperform some compilers by a factor of ten or more, a logarithmic scale is used in y-axis. ompi appears to be from 4 to 8 times faster than icc and 20 to 80 times faster than the original (old) implementation. Since Fib exploits nested task parallelization which creates a deep tree of small tasks, it is a common phenomenon some threads to fill their queues. ompi has a significant performance advantage by leveraging the new work-stealing implementation and the fast execution path produced by the compiler; task load is quickly balanced between threads, and the application delves into throttling mode. Moreover, ompi, along with icc, scales up with the number of threads.

Similarly to Fib, Nqueens exploits nested task parallelization which creates a deep tree of tasks. In the NQueens benchmark displayed in Fig. 3.14b, for an input of 14 queens we get similar results to Fib and ompi gives the best times. ompi is up to 2 times faster than old and up to 3 times faster than icc (not shown clearly in the logarithmic scale).

Floorplan calculates the optimal floor plan distribution of a number of cells. Tasks are hierarchically generated for each branch of the solution space. This application induces many data synchronizations and comes with a very irregular and aggressive pruning mechanism, which results in a heavily unbalanced task tree. Fig. 3.15 displays results of the application when the input.20 file is used; icc is not included here because the application could not execute properly when compiled with this compiler. ompi achieves the fastest times and our original implementation follows. Since Floorplan generates deep nested tasks, ompi performs well due to the work-stealing implementation along with the efficient fast path execution. suncc cannot exhibit speed-up, while gcc experiences significant slow-down when more threads are used.

Results from the rest of BOTS applications are given in Table 3.3, for the case of 16 threads. In this table we included results from ompi when using icc as back-end compiler, which in many situations produces faster code for the sequential part of the

Table 3.3: Execution time (sec) of BOTS using 16 threads

Compiler	FFT	Health	Sort	SpLU	Str.	Align.
GCC	17.571	141.85	2.007	1.679	24.602	1.576
ICC	2.086	**4.778**	0.621	1.676	20.641	1.338
SUNCC	2.473	15.694	0.652	1.835	21.619	1.218
OLD	2.086	7.114	**0.591**	1.766	21.589	1.587
OMPi	1.918	5.327	0.610	1.668	22.368	1.604
OMPi_ICC	**1.889**	4.787	0.621	**1.667**	**20.524**	**0.957**

application. In FFT, SparseLU, Strassen and Alignment applications OMPi with ICC as backend proves to be faster, while performing second best only in two applications with very small margins (3% in Sort and 0.2% in Health). ICC has the best behaviour in Health application, while our OLD system is the fastest as far as the Sort application is concerned. Thus, OMPi proves to perform consistently well in many different application scenarios, and especially when it uses an efficient back-end compiler, giving it a serious performance advantage. In general, ICC and SUNCC perform quite well with few exceptions. The version of GCC we had available does not perform up to par.

Chapter 4

Transforming Nested Workshares into Tasks

In this chapter we present a novel way of avoiding nested parallel loop overheads through the use of tasks. In particular, as our first contribution, we show that it is possible to replace a second-level loop by code that creates tasks which perform equivalent computations; the tasks are executed by the first-level team of threads, completely avoiding the overheads of creating second-level teams of threads and oversubscribing the system. We use the proposed method to show experimentally the performance improvement potential. At the same time we observe that our techniques require sizeable code changes to be performed by the application programmer, while they are not always applicable for arbitrary loop bodies. Thus, we propose a way to automate the whole procedure and provide transparent tasking from the loop nests, which, except the obvious usability advantages, does not have the limitations of the manual approach. We present the implementation details of our proposal

in the context of the ompi [7] compiler. Finally, we perform a performance study using synthetic benchmarks as well as a face-detection application that utilizes nested parallel loops; all experimental results depict the performance gains attainable by our techniques.

The main advantage of our technique is the avoidance of thread creation and manipulation overheads, in applications that need to use nested teams in order to maximize their performance. The oversubscription problem induced by nested teams may not be so severe in larger systems (e.g. many-core chips with hundreds of cores). Our proposal is general and can also be applied in such systems: while embedded hardware architectures can easily allow the execution of a team of threads, supporting nested parallelism may be hard to be handled in an efficient manner due to resource limitations. Automatically replacing nested parallel loops with tasks, gives the possibility to express nested parallelism and execute it efficiently in those systems, too.

Among the important features included from the very beginnings of OpenMP was *nested parallelism*, that is the ability of any running thread to create its own team of child threads. Although actual support for nested parallelism was slow to appear in implementations, nowadays most of them support it in some way. However, it is well known that nested parallelism, while desirable, is quite difficult to handle efficiently in practice, as it easily leads to processor over-subscription, which may cause significant performance degradation.

The addition of the `collapse` clause in V3.0 of the specifications can be seen as a way to avoid the overheads of spawning nested parallelism for certain nested loops. However, it is not always possible to use the `collapse` clause since:

* the loops may not be perfectly nested
* the bounds of an inner loop may be dependent on the index of the outer loop
* the inner loop may be within the extend of a general parallel region, not a parallel-loop region.

The nature of OpenMP loops is relatively simple; they are basically do-all structures with independent iterations, similar to what is available in other programming systems and languages (e.g., the FORALL construct in Fortran 95, the `parallel_for` template in Intel TBB [81], or `cilk_for` in Cilk++ [86]). What is interesting is that

some of these systems implement DO-ALL loops without spawning threads; they are mostly creating some kind of task set to perform the job. Can an OpenMP implementation do the same? While this seems rather useless for first-level parallel-for loops (since there is no team of threads to execute the tasks; only the initial thread is active), it may be worthwhile in a nested level.

The version V4.1 of the OpenMP specifications [87], includes the `taskloop` construct. This construct specifies that the iterations of one or more associated loops should be distributed and executed in parallel through the generation of appropriate OpenMP tasks. The `taskloop` construct, however, is quite different from what we propose here:

(i) It forces the programmer to re-formulate existing code so as to utilize the new directive, while our proposal requires no code modifications whatsoever.

(ii) It does not allow user-directed scheduling of the loop iterations; in particular it does not contain loop `schedule` clauses. It also does not allow the `ordered` clause for serializing iterations.

(iii) Our techniques can be extended to non-loop worksharing constructs such as `sections`.

(iv) A significant difference is that it may not utilize all available resources, even if they are available. The reason is that it does not generate additional parallelism levels; the generated tasks must be executed by the team of threads that met the `taskloop` construct. Our techniques, in contrast, may choose to utilize more threads, depending on the execution conditions.

4.1 Proof of Concept: Re-Writing Loop Code Manually

In this section we give two OpenMP examples where a code block that includes nested parallelism can be rewritten and replaced by a set of tasks, avoiding the creation of second level threads. In the first example we show the necessary steps to rewrite a loop with a static schedule, whereas in the second example we show the corresponding steps for the dynamic schedule. We then explain what makes these transformations possible. Finally we compare the performance of the transformed code to the original (with nested parallelism).

```
#pragma omp parallel num_threads(M)
{
  #pragma omp parallel for\
  schedule(static) num_threads(N)
  for (i=0; i<K; i++) {
    <body>
  }
}
```

(a) Original

```
#pragma omp parallel num_threads(M)
{
  for (t=0; t<N; t++)
    #pragma omp task
    {
      sta_calculate(N,0,K,&lb,&ub);
      for (i=lb; i<ub; i++) {
        <body>
      }
    }
  #pragma omp taskwait
}
```

(b) Transformed

Figure 4.1: Nested parallel loop using static schedule

4.1.1 Transforming a Loop With a Simple Static Schedule

Consider the sample OpenMP code shown in Fig. 4.1(a), where a team of M-first level threads is created[1]. Each one of these threads executes a nested parallel for-loop which will normally spawn a team of N threads. Consequentially, a total of $M \times N$ threads may be simultaneously active in the system executing the second-level parallelism body of code. This number of threads may be large enough to lead to excessive system oversubscription. Otherwise, the programmer must explicitly specify the number of threads for each parallel region according to the number of available cores. This, however, reduces the performance portability and hardware abstraction that OpenMP provides.

Fig. 4.1(b) illustrates how the code of Fig. 4.1(a) can be re-written so as to make use of OpenMP tasks instead of nested parallelism. The idea is conceptually simple: each first-level thread creates N explicit tasks (i.e. equal in number to the original code's second-level threads), and then waits until all tasks are executed. The code of each task contains the original loop body. In order to perform the same work the corresponding thread would perform, it is necessary to calculate the exact iterations that should be executed, hence the `sta_calculate()` call. In essence, in `sta_calculate()`

[1]Even if `num_threads(M)` is absent from the outer parallel construct, the standard practice of many OpenMP implementations is to produce as many threads as the number of available processors. Assume, without loss of generality, that they are equal to M.

```
#pragma omp parallel num_threads(M)
{
  #pragma omp parallel for\
  schedule(dynamic) num_threads(N)
  for (i=0; i<K; i++) {
    <body>
  }
}
```

(a) Original

```
#pragma omp parallel num_threads(M)
{
  for (t=0; t<N; t++)
    #pragma omp task
    {
        do {
         dyn_calculate(N,0,K,&lb,&ub);
         for (i=lb; i<ub; i++)
           <body>
       } while(lb<ub);
    }
  #pragma omp taskwait
}
```

(b) Transformed

Figure 4.2: Nested parallel loop using dynamic schedule

the user must write code in order to pre-calculate statically the exact for-loop iterations handled by each explicit task. In the original code a nested team has N threads and each thread will have to execute approximately K/N iterations. Consequently, in the transformed code, `sta_calculate()` function has to supply each task with the corresponding K/N iterations.

4.1.2 Transforming a Loop With a Dynamic Schedule

In Fig. 4.2(a) the nested part of the code creates N threads that have to execute a total K iterations which are distributed through a dynamic schedule. In Fig. 4.2(b) we show the necessary modifications in order to use tasks. The calculation of the iterations that each task should execute is done in `dyn_calculate()`. However, in contrast to Fig. 4.1 the calculations have to be done within a while loop (i.e. they cannot be predetermined and are dynamically given at runtime). Furthermore the implementation of the `dyn_calculate()` function itself differs, since now it has to emulate the execution of the dynamic schedule where the iterations are distributed in chunks as the tasks request them. After a chunk is executed, the task requests another chunk, until no chunks remain to be distributed.

The implementation of the `dyn_calculate()` function is not trivial. The code must include some kind of synchronization, in order for the tasks to continuously content

for fetching and executing the loop iterations. Similar rules apply also to the case of loops with *guided* schedules. This synchronization is not necessary in the static schedule because calculations regarding the distribution of iterations are done statically and independently before the actual execution of the loop.

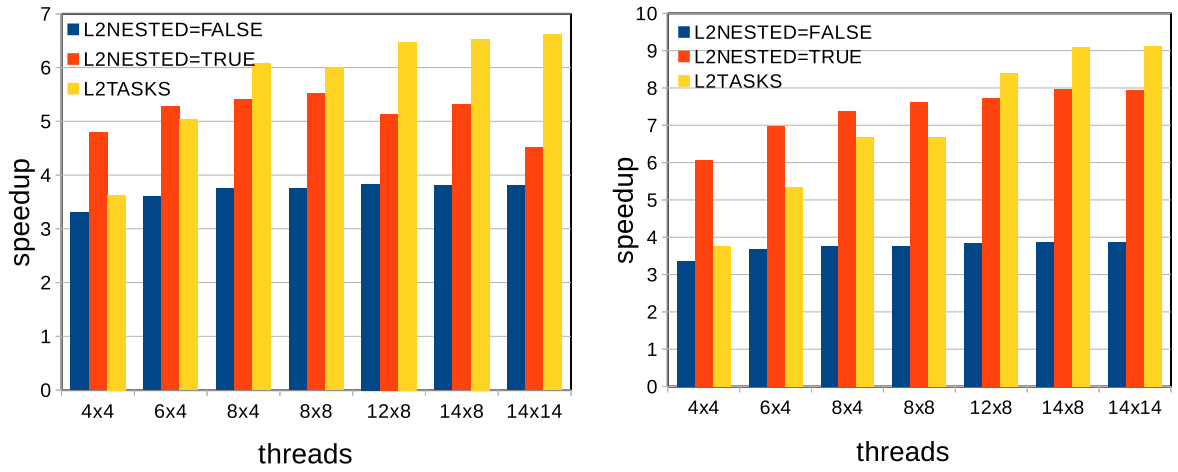
4.1.3 Comparing the Original and the Transformed Code

The main question at this point is: Why does this technique work? According to OpenMP terminology, the original code creates M sets of N *implicit* tasks, one set for each of the M first level threads. Afterwards, the system creates $M \times N$ OpenMP threads in order to execute these *implicit* tasks. On the other hand, the rewritten code emulates these implicit tasks through the use of *explicit* tasking. Here each one of the first level threads creates N *explicit* tasks to represent the work of the nested for loop. The main difference is that in this case the work is carried by the M first level threads and no new OpenMP threads need to be spawned.

Nevertheless, according to the OpenMP specifications, there are some fundamental differences between *implicit* and *explicit* tasks. Implicit tasks may contain barriers which are used to synchronize threads; this possibility is not allowed within the body of explicit tasks. In our case this is not an actual problem because implicit tasks that are created by Fig. 4.1(a) and Fig. 4.2(a) execute independent loop iterations, and within the loop body there can not exist a barrier closely nested.

Another difference between the two types of OpenMP tasks, are the data sharing attributes of the referenced variables. In implicit tasks all referenced variables default to `shared` unless the user specifies otherwise. In contrast, in explicit tasks all referenced variables default to `firstprivate` (unless global or user specified), which means that each task handles a separate copy of the variable. Consequentially, when utilizing the proposed technique, the user must explicitly *set the data sharing attributes of all referenced variables* in the new task-based code to guarantee the correctness of the code.

To summarize, the programs in Figs. 4.1(a),(b) and 4.2(a),(b) are equivalent, and no other changes apart from explicitly setting the sharing attributes are required. The important difference is that the codes in Figs. 4.1(b) and 4.2(b) do *not* spawn threads at the second level of parallelism. They utilize the tasking subsystem of the compiler and use only the available M threads to execute the $M \times N$ tasks generated in total. While task creation and execution is not without overheads, it remains mostly in the



(a) GCC

(b) ICC

Figure 4.3: Performance of the proposed technique on a 24-core system; speedup for a face detection algorithm applied on an test image with 57 faces. In an $M \times N$ configuration M (N) are used in the first (second) level of parallelism. Speedups are calculated for each compiler based on its own sequential times

realm of the OpenMP implementation to deal with it efficiently. On the other hand controlling a large amount of threads resulting from nested parallel regions may not be possible, especially when the OpenMP runtime relies on kernel-level threads (such as posix threads, which is a usual case).

4.1.4 Confirming the Performance Gain

We applied the above technique to the parallel version of a high-performance face detection application [88]. The application itself, the 24-core system, as well as the software configuration we used for our experiments are described in more detail in Section 4.4 so we will avoid repeating it here. The important issue is that the application contains a first-level parallel loop with unbalanced iteration load. The number of iterations depends on the image size and is usually less than 14. Inside this loop, there exist multiple second-level parallel for-loops, which clearly need to be exploited in order to increase performance. We re-wrote these second-level loops according to the method described in the previous section. We compiled the code with various OpenMP compilers and in Fig. 4.3 we show the execution results obtained using GNU gcc and Intel icc on a particular image containing 57 faces; similar results were observed with other compilers, too. In the figure, the new application code is designated

as L2TASKS. The original code which utilized nested parallelism is the L2NESTED=TRUE part. For comparison, we include the L2NESTED=FALSE bars which represent the original code executed with nested parallelism disabled (i.e. the environmental variable `OMP_NESTED` was set to false).

In the plots we vary the number of participating threads per level using up to $M = 14$ threads for the first level and up to $N = 14$ threads in the second level, denoted as $M \times N$ configuration. For both compilers nested parallelism (L2NESTED=TRUE) boosts performance as long as processors are not heavily oversubscribed. It can be seen that GCC's performance drops for large number of threads, while ICC seems to handle the situation much better, although its performance get slightly worse after the 14×8 configuration. Our approach results in better speedups after 8 or more first-level threads for GCC and after 12 or more first-level threads for ICC, confirming the validity of our approach. The lower performance shown in smaller configurations is expected since we only rely on the few first-level threads while nested parallelism is able to utilize all processors in the system. Finally, in the larger configurations, notice that while the L2NESTED=TRUE code utilizes all the 24 available processors (albeit with increased overheads), we obtain better speedups with only 14 threads.

4.1.5 Limitations

In the previous sections we presented the core idea behind our method, demonstrating the necessary transformations for loops with different schedule types. Consider for example the `dynamic` schedule case. In this case the implementation of the `dyn_calculate()` function (this is the function that determines which iterations should be executed by which task) can be particularly complex. This is because this function on one hand must calculate the iteration bounds for each task, in order to hand out the iterations in accordance to the loop schedule, and on the other hand must handle the competition/synchronization among tasks which is crucial for efficiency. Similar issues arise for the guided schedule type. In essence, the user has to re-implement a mini worksharing runtime subsystem in order to cover all possible schedule configurations. This is clearly both undesirable for the user and redundant as far as the compiler is concerned, since all this functionality should already be present in its OPENMP runtime library.

Another important issue is that even if the user is determined to do all this

work, this will not be enough to make it applicable to all possible cases. The reason is that within the loop body there may exist references to thread-specific quantities; for example, the loop body may contain calls to the OpenMP runtime function `omp_get_thread_num()` and utilize the thread's ID in computations. Clearly this problem demands serious programming effort and additional bookkeeping. Furthermore the loop body may access `threadprivate` variables. These are variables that are private and bound to OpenMP threads and according to the specifications, may retain their values throughout the whole program execution, even between successive creation and destruction of thread teams. The above makes it almost impossible to move the loop's body to independent tasks, as there is no guarantee as to which threads will execute what tasks.

In conclusion, the manual code transformations need extensive programmer involvement and are not applicable in the general case. On the other hand, all the required functionality is already implemented within the runtime library of any OpenMP system. Additionally, the runtime system has access to all the stored thread-specific quantities. It should thus be in position to support the required transformations seamlessly.

4.2 Automatic Transformation

In this section we consider the automation of the code transformations presented earlier, so as to be performed transparently by the compiler. Implementation details within a particular compiler are discussed in Section 4.3. In our discussion here we consider compilers that utilize the *outlining* [77] technique to translate OpenMP parallel and task regions, which is quite common a case (e.g. [89, 90, 66, 63, 35, 91]). We show that translator-related requirements are minimal; most of the hard work is delegated to the runtime library.

The outlining technique involves moving the code residing within a block to a new function; the original block is then replaced by a call to the new function. This method can be applied to `parallel` and `task` OpenMP regions. Concentrating on the `parallel` regions, the body of the construct is extracted and placed in a new function; in its place in the original block, the translator inserts runtime calls to create a team of threads which will execute the code in the new function. A major issue here is

setting correctly the data environment according to the visibility of variables in the original code block and any OpenMP data clauses that may appear in the construct.

Notice that the outlined code contains all the transformations performed by the compiler. In particular, any OpenMP loop is replaced with new code that handles the distribution of loop iterations among threads (quite similar in nature to the transformations in Figs. 4.1 and 4.2, without tasking, of course). As such, the outlined function should contain all the logic needed to execute the loop iterations no matter if it is called from a team of threads, from a single thread, or from any other place in the program. Consequently, our proposed technique should not require *any* changes to the code produced by the compiler. It is the runtime library that must decide how to actually call the outlined function (i.e. through threads or through tasks). In fact, as we discuss in Section 4.3, the only change we had to implement in our compiler was adding a single parameter to the runtime call that creates the team of threads, so as to let the runtime system know that the construct is a combined parallel-for one.

According to the above discussion, almost the entire implementation of the proposed technique is concentrated within the runtime system, since it maintains all the needed information. The runtime has to be extended in such a way as to be able to encapsulate the functionality of implicit tasks within the context of explicit ones. In particular the runtime actions include:

* Decision whether threads or tasks should be used.
* Creation of special explicit tasks instead of implicit tasks (threads).
* Execution and synchronization of the special explicit tasks.

If the runtime decides to execute a nested parallel loop utilizing tasks instead of threads, the thread that encounters this particular construct must create a set of special explicit tasks, equal in number to the number of threads that would normally execute it. The code and the corresponding data of the special tasks should be identical to the implicit task code and data as generated by the compiler. These special tasks should be stored in a shared data structure, from where the sibling threads can steal and execute, as if they were normal explicit tasks.

In order to emulate the implicit barrier at the end of the parallel region[2], the thread that creates the special tasks must execute an `omp taskwait` directive afterwards. This

[2]Notice that no explicit barrier has to be emulated here, since there can be no barrier closely nested inside a parallel loop.

way, the runtime ensures that the thread which reached the parallel loop waits until the whole code block is executed. If any sibling threads are idle, they could help with the execution of the special tasks, otherwise all the special tasks will be executed by the thread that created them.

4.2.1 The Case of `ordered`

Our proposal substitutes a nested team of threads by an equivalent set of tasks, for any OpenMP loop schedule type. However, independently of the actual implementation, an initial concern is the possibility of something going wrong in the case where the number of threads available at the outer level is different (more precisely, *smaller*) than the one specified at the inner-level loop. As long as there are no dependencies among the loop iterations, the number of outer-level threads that execute them is actually irrelevant; even a single thread may execute all of the produced tasks.

There is, however, one particular case where dependencies are explicitly introduced by the programmer: `ordered` regions. The `ordered` directive forces ordering dependencies among the iteration executors. When the executors are threads there is no problem whatsoever but what about tasks? Because these tasks may be executed by a smaller number of outer-level threads than the requested for the nested parallel region, is there any possibility that particular task scheduling sequences lead to *deadlock*? What we show here is that this might only happen in one particular case.

The single problematic case is the `static` schedule with a user-specified chunk size. Although it is a matter of how the compiler and the runtime implement the `static` schedule, a straightforward way of executing it is by using a double loop; the outer loop iterates over the series of chunks while the inner loop goes over the actual iterations of a particular chunk, as illustrated in Fig. 4.4. As the loop bounds are *precalculated* (since for this particular schedule they are not subject to competition among the executors), imposing an `ordered` directive may lead to a deadlocked situation, depending on how tasks are implemented / scheduled.

To see this consider the case presented in Fig. 4.5 where we have 2 threads in the first level that have to execute 60 second-level loop iterations with a chunk size equal to 10. Because the original code employs 3 second-level threads, according to our methodology 3 explicit tasks are generated T_0, T_1, T_2. The compiler divides the 60 iterations into 6 chunks and assigns them statically to the 3 explicit tasks.

```
#pragma omp for schedule(static,c)
{
  for (i=lb; i<ub; i+=st) {
    <body>
  }
}
```

(a) Original

```
calc_chunks(lb, ub, st, &lch, &uch);
for (l=lch; l<uch; l++) {
  for (m=0; m<c; m++) {
    i = calc_index(l,m);
    <body>
  }
}
```

(b) Transformed code

Figure 4.4: Possible transformation of a loop with `schedule(static,c)`

According to the OpenMP specifications T_0 has to execute the first and fourth chunk, that is iterations 0-9 and 30-39, task T_1 executes the second and fifth chunk with iterations 10-19 and 40-49 and finally task T_2 executes the third and sixth chunk with iterations 20-29 and 50-59. The outer-level `parallel` region provides 2 active threads to execute the three tasks. Thread 1 executes T_1 and blocks at the first iteration of chunk 1 due to the `ordered` clause while thread 0 executes T_0 (Fig. 4.5(c)). When thread 0 finishes chunk 0, it proceeds to the beginning of the next assigned chunk (chunk 3) and blocks due to `ordered` clause. Thread 1 unblocks and executes chunk 1 (Fig 4.5(d)). When thread 1 finishes it proceeds to the beginning of the next assigned chunk (chunk 4) and blocks due to `ordered` clause. At this point both threads are blocked waiting for the execution of chunk 2 (Fig 4.5(e)). If tasks are executed on a run-to-completion basis, the remaining task (T_2) will never be given a chance to run and advance the iteration count, resulting in a deadlock. The solution to this problem depends on the implementation. We discuss a simple solution in the context of our compiler in Section 4.3.

The above problem can not occur in any other schedule type. This is because even if there is only one outer-level thread available to execute the generated tasks, there will always be at least one task active, advancing the iteration count and obtaining the next chunk of iterations (i.e. making progress). For example, consider the case of `dynamic` schedules; if there is a thread (executing a task) blocked at an `ordered` directive then there must exist at least one other thread that obtained the (sequentially) previous chunk; eventually the latter will be executed and the turn of the former will come.

```
#pragma omp parallel num_threads(2)
{
  …
  $pragma omp parallel for schedule(static, 10) num_threads(3) ordered
  for(i=0; i<60; i++)
  {
    <body>
  }
  …
}
```

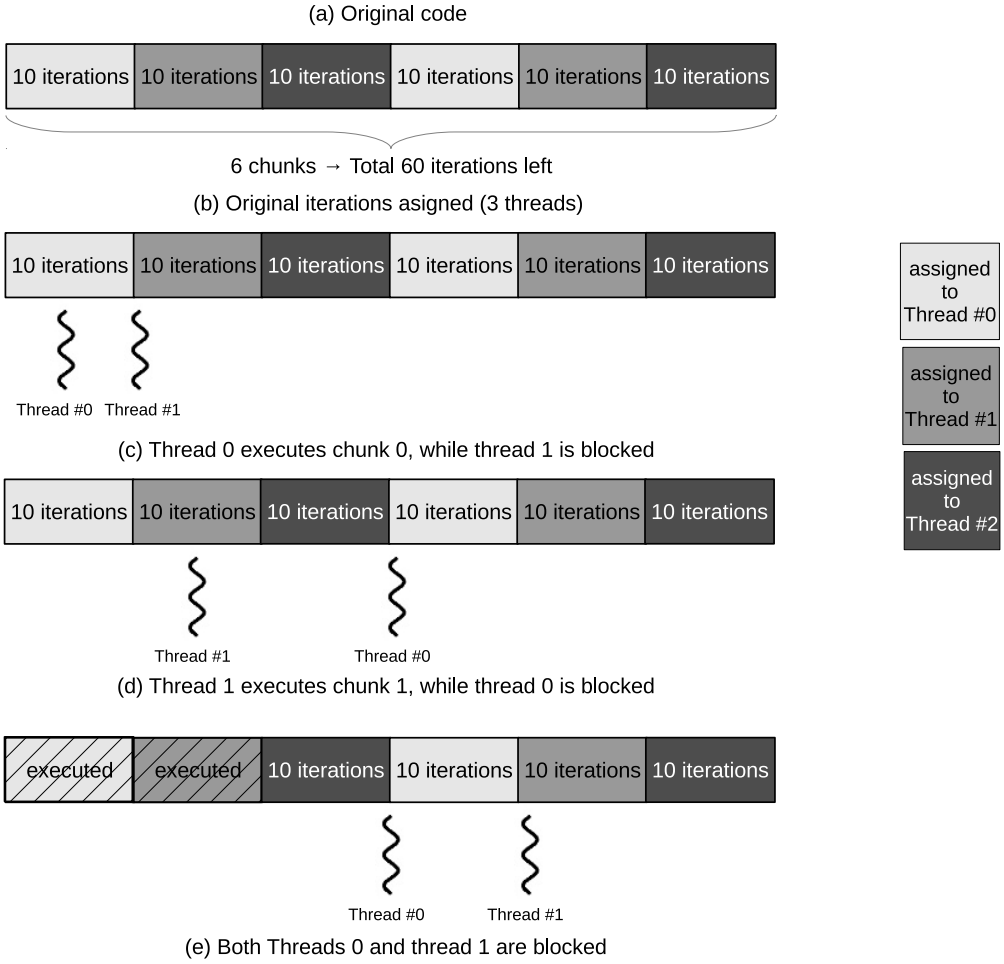


Figure 4.5: Two threads try to execute a nested `for` loop with a `static` schedule and an `ordered` clause that was initially meant for 3 threads. a) OpenMP code b) The iterations are statically assigned to the 3 second-level threads, which get substituted by 3 tasks. c, d) Threads 0 and 1 execute one chunks each and block on the `order` region. e) Deadlock occurs since iterations 20 to 30 will never be executed

4.2.2 Extension to `sections`

The technique we propose can be also applied to nested `parallel sections` work-shares. The idea is given in Fig. 4.6. The original code (Fig. 4.6(a)) can be replaced with the one in Fig. 4.6(b) which uses tasks instead of second-level threads. In the original code a first level team of M threads is created and each one of these threads

```
#pragma omp parallel num_threads(M)
{
  #pragma omp parallel sections
  {
    #pragma omp section
    {
      <code 1>
    }
    #pragma omp section
    {
      <code 2>
    }
    ...
  }
}
```

(a) Original

```
#pragma omp parallel num_threads(M)
{
  #pragma omp task
  {
    <code 1>
  }
  #pragma omp task
  {
    <code 2>
  }

  ...

  #pragma omp taskwait
}
```

(b) Transformed

Figure 4.6: Extending the technique to nested `parallel sections`

creates a new team in order to execute a `parallel sections` workshare. As in previous examples, here we may have possible system oversubscription. On the transformed code, each one of the first level threads creates a set of explicit tasks that represent the workload of the nested `parallel sections` region. At the end, the `taskwait` directive must be used to ensure the correctness of the execution.

The transformation procedure for `parallel sections` is much simpler than the one described for `parallel for`. Sections are basically independent code blocks that are assigned to a team of threads for instant execution. According to the OpenMP specification there is no restriction about which thread will execute which section block. Similarly to the `for` case, there is no barrier allowed within a `section` code block. As a result `sections` can be replaced by tasks.

The transformation procedure has similar limitations to the ones discussed in Section 4.1.5 for the case of `parallel for`. These limitations can be alleviated if the compiler and its runtime system undertake the automation of the required procedures. In the next section we present the actual implementation of our proposed techniques in the context of the ompi compiler.

4.3 Implementation in the OMPi Compiler

Our implementation is based on the infrastructure of the ompi compiler. An overview of ompi's architecture as well as a detailed description of the tasking runtime is given in Sections 1.5 and 3.3, respectively.

4.3.1 The Implementation of the Proposed Technique

In order to automate the generation of tasks, it was necessary to modify the code produced by the ompi compiler as well as add new functionality to the runtime system.

Regarding the compiler, we introduced rather minimal changes, which were limited to the case where a combined `parallel for` or `parallel sections` construct is encountered. An identical outlined function is still created which includes all the code needed for sharing the loop iterations among threads. However, the embedded call to create the required team of threads now includes a new parameter to let the runtime know that this is a combined construct. This extra parameter is basically the only change in the code produced by the compiler. Note that this covers both nested and orphaned construct cases alike; that is the compiler just marks that the construct is a combined one, without considering lexical nesting; the runtime is responsible for deciding whether this is actually a nested parallel region or not.

The changes in the runtime system (ort) were much more extensive. Whenever a team of threads has to be created (through the call produced by the compiler), then instead of threads an equal number of explicit tasks are created, if all the following are satisfied:

1. The team is going to operate in nesting level > 1.
2. The parallel region is a combined `parallel for` or `parallel sections` region.
3. The user allows it.

For the third condition above, we have added a new environmental variable (named `OMPI_PAR2TASK_POLICY`) which lets the user decide whether the proposed technique should be applied or not. However, as noted previously the above are not enough to cover the cases where the user code accesses thread-specific data.

In order to describe our solution we need to consider the way ompi handles OpenMP threads. Specifically, ompi associates a control block (eecb) with every execution entity

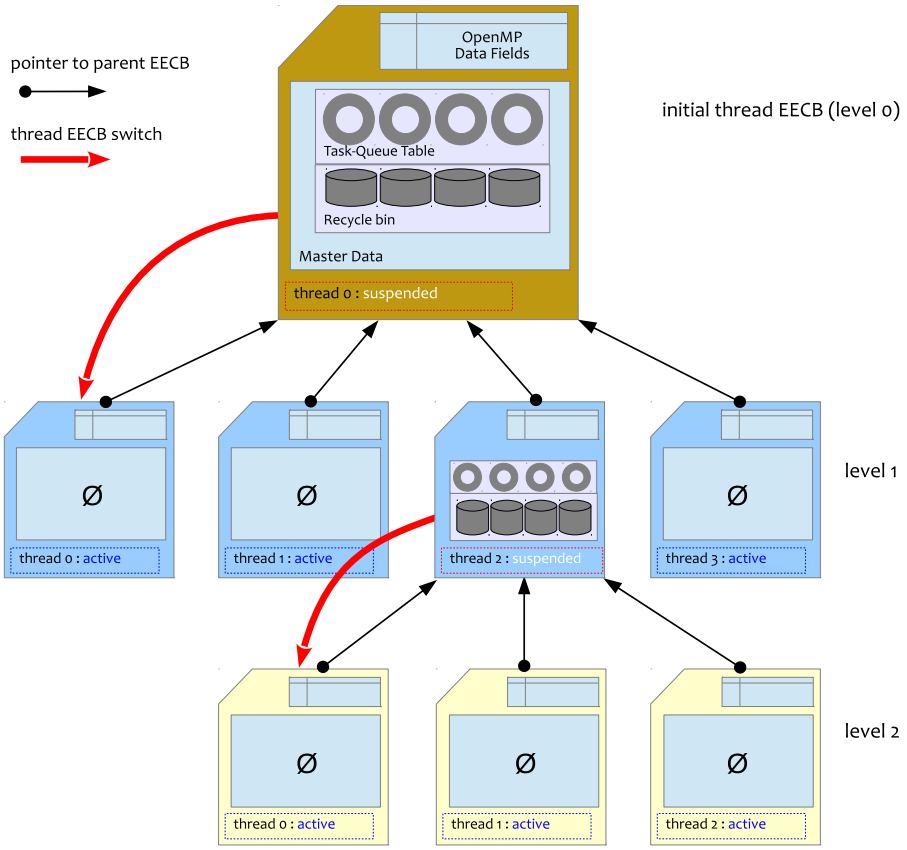


Figure 4.7: EECB handling under nested parallelism. A total of 6 different threads are employed. The initial thread and thread 2 of level 1 switch to new EECBs to act as team masters

it manages. The EECB contains everything ORT needs in order to schedule the thread, including the size of the team, the thread ID within the team, its nesting level, etc. The only thread-specific data not actually stored in a thread EECB are `threadprivate` variables. These are allocated at the team's parent control block and this because the runtime system has to guarantee persistence across parallel regions, as required by the OPENMP rules. The EECB makes `threadprivate` variables available through a pointer to the parent EECB (thus a tree of EECBs is formed at runtime). In addition, if an execution entity ever becomes a parent of a new team, the EECB stores the tasking structures (TASK_QUEUES, recyclers) on behalf of its children. This is depicted graphically in Fig. 4.7, where the initial thread (0th level) has created a team of 4 first-level threads, and child thread with id 2 has created its own team of 3 threads. Notice that because a parent thread participates in the team as a *master* thread (with id 0), the total number of distinct threads is actually equal to 6, while all 8 EECBs are actively participating in the tree. The initial thread temporarily assumes the EECB of

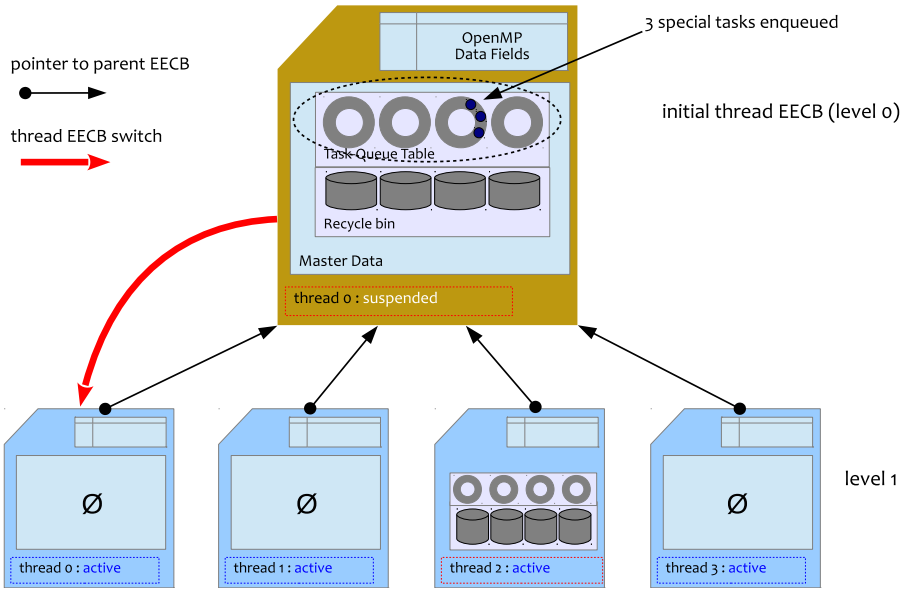


Figure 4.8: Runtime creates special tasks instead of second level threads

thread 0 in the first-level team (denoted by a thick arrow), and similarly thread 2 of the first level temporarily assumes the EECB of thread 0 in the second level. When the second-level threads finish their execution, the master thread will revert to its original first-level EECB, becoming again thread 2 of the first-level team.

Conclusively, everything a running thread requires is serviced through its control block. Whenever a thread starts the execution of a parallel region, ORT assigns a new EECB to it, which is later freed when the team is disbanded. Based on the above, the main idea behind our implementation is that the produced tasks try to mimic threads. Every task produced (instead of a thread) when a nested combined parallel-workshare region is encountered, carries a special flag (S) along with the ID number the corresponding thread would have. The tasks are inserted as usual in the TASK_QUEUE of the outer-level thread that encountered the nested construct. When such a task is scheduled for execution (either by the same thread or a thief), the S flag will induce the following actions:

* A new EECB is created, as would be done if a new nested thread was created in the first place, updating the tree of EECBs correspondingly.
* The outer-level thread that is about to execute the task assumes temporarily the new EECB and sets its thread ID equal to the ID stored within the task.
* The task becomes tied to this thread.

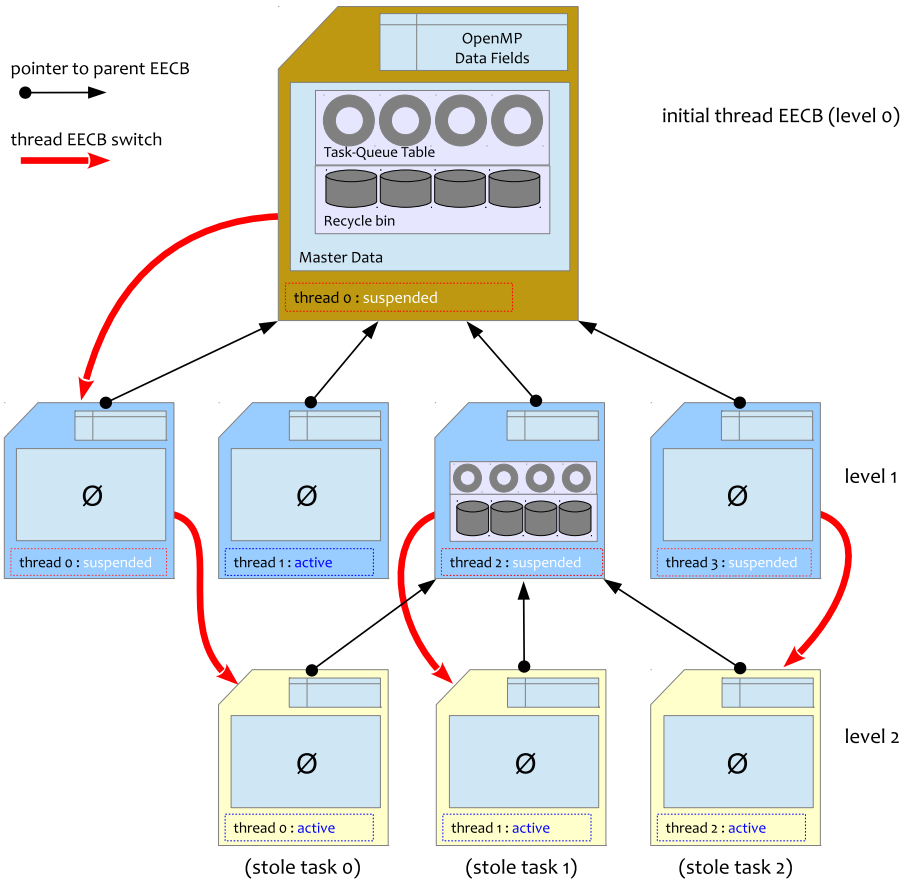


Figure 4.9: Runtime executes special tasks

An example is given in Figs. 4.8–4.9, which are based on the example discussed above and given in Fig. 4.7. In Fig. 4.8, thread 2 of the first level creates 3 special tasks instead of 3 second-level threads. The tasks descriptors are dully inserted in the thread's TASK_QUEUE which is stored in its parent. In addition, the thread will serve as a parent for the (virtual) second-level team, thus it initializes the tasking structures to be used by its "children". What happens next is that thread 2 and it siblings begin stealing and executing these special tasks. The situation is depicted in Fig. 4.9. In there, level-1 thread with id 0 stole and currently executes special task 0; a new EECB has been created and given temporarily to thread 0, letting it act as if it was thread 0 of the second-level team. Thread 2 of level 1 executes special task 1 and thus becomes thread 1 of the second-level team, with a new EECB. Finally, thread 3 executes special task 2. When a first-level thread finishes executing a special task (acting like a second-level thread), it restores its original EECB and resumes its role as a first-level thread.

In essence, while executing a special task, an outer level thread obtains all the characteristics that the inner level thread would have, if it was created normally. As such it is able to handle thread-specific data accesses, overcoming all the limitations mentioned in 4.1.5. Notice, for example, that because the old control block of the outer thread remains intact in the tree, all information needed to service runtime calls such as `omp_get_level()` or `omp_get_active_level()`, is readily available. When the task execution is finished, the temporary EECB is freed and the thread resumes its original control block, continuing with its normal operation.

4.3.2 Ordered

OMPi by default executes tasks to completion and is thus susceptible to this problem. The engineering solution we currently follow is to avoid the problem altogether: if the loop schedule is `static` and an explicit chunk size is given and an `ordered` clause is present, nested parallelism is generated as usual, instead of tasks. We are currently working to support the `taskyield` directive. Yielding upon an imminent `ordered` block should allow the possibility of other tasks to be executed and thus make progress.

4.4 Evaluation

We have performed several experiments in order to evaluate the performance gains of our proposal. We report here the results of a synthetic benchmark and a real world face detection application. These tests were run in two multicore machines. The first machine is a NUMA server which includes two 12-core AMD Opteron 6166 CPUs operating at 1.8GHz and a total 16GB of main memory. The operating system is Debian Wheezy based on the 3.2.0-4 Linux kernel. In this machine, apart from OMPi, we had the following compilers available: GNU GCC (version 4.7.2), Intel ICC (version 13.0.0) Oracle SUNCC (version 12.3). We used "-O3 -fopenmp" flags for GCC, "-O3 -openmp" flags for ICC and "-fast -xopenmp=parallel" flags for SUNCC. GCC with the "-O3" flag was used as a back-end compiler for OMPi. For all compilers, the default runtime settings were used. These settings also happened to produce the best results, after exhaustive experimentation.

The second machine is a NUMA SGI UV100 with four computational nodes. Each node includes an 8-core Intel Xeon E7-8837 CPU operating at 2.66GHz and 32GB of main memory. For our experiments we had access to one computational node, hence to only one 8-core multiprocessor. The operating system is 64bit SUSE Linux Enterprise Server 11 based on the 2.6.32.54 Linux kernel. The available compilers apart from OMPi were: GNU GCC (version 4.5.0) and Intel ICC (version 12.1.5). We used "-O3 -fopenmp" flags for GCC and "-O3 -openmp" flags for ICC. We used ICC with the "-O3" flag as a back-end compiler for OMPi. Again for all compilers, the default runtime settings were used.

4.4.1 Synthetic Benchmark

The first experiment aims at showing directly the performance gains possible with our methodology. A synthetic benchmark is used, measuring the time taken to execute the code shown in Fig. 4.10. This code is based on the EPCC microbenchmarks [92] which are used to estimate OpenMP construct overheads; we instead measure the total execution time. This synthetic benchmark was run on the 24-core Opteron server, so in the main function a team of 24 threads is created and each thread calls the `testpfor()` function once. In there a thread executes a combined `parallel for` directive `REPS` times, creating N second-level threads, each one performing work, the granularity of which is controlled by the `TASK_LOAD` parameter in the `delay()` function.

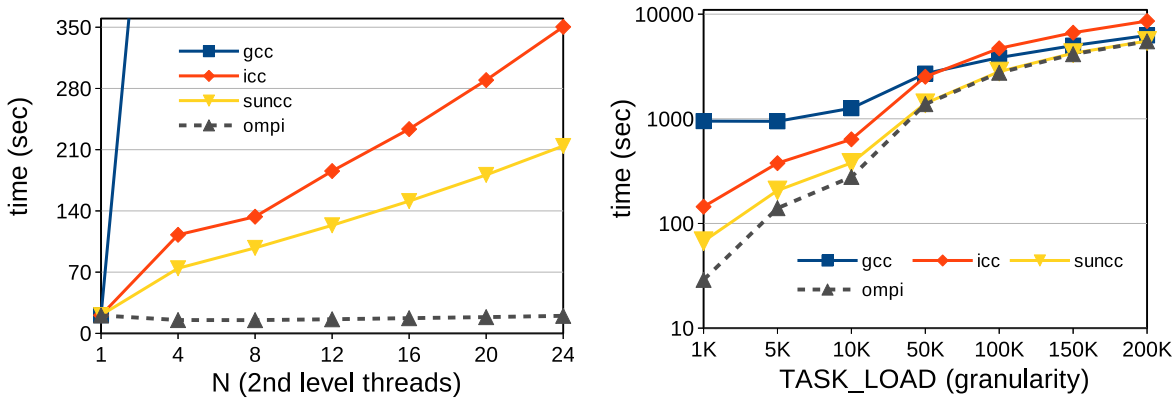
```
delay() {
  volatile i, a;
  for (i=0; i < TASK_LOAD; i++)
    a += i;
}

testpfor() {
  for(i=0; i < REPS; i++)
  #pragma omp parallel for num_threads(N)
  for (j=0; j < N; j++)
    delay();
}

main() {
  #pragma omp parallel for num_threads(24)
  for (i=0; i < 24; t++)
    testpfor();
}
```

Figure 4.10: Code for synthetic benchmark

We used REPS = 100000 and varied the TASK_LOAD value.

We present the results in Fig. 4.11. In Fig. 4.11a we consider fine grain work (TASK_LOAD = 500) and vary the number of second-level threads in order to stress the runtime system. The growing number of threads (up to $24 \times 24 = 576$ threads are produced) results in considerable overheads caused by the severe processor oversubscription. These overheads are clearly depicted in the total execution time. Because OMPi avoids spawning the extra level of nested threads, it exhibits remarkable stability in its performance, which is only very slightly affected by an increasing number of generated tasks.

In figure 4.11b we fixed the number of second-level threads to $N = 4$ and varied the work granularity, with TASK_LOAD values in the range of 1K to 200K. We use a logarithmic scale due to the wide range of timing results. As expected, for finer grain work our methodology results in significantly faster execution as compared to other compilers. As the work gets coarser, all compilers tend to exhibit similar performance since the task or thread manipulation stops being the performance bottleneck and execution time is dominated by the actual computation. For the coarsest load, most



(a) `TASK_LOAD` = 500

(b) $N = 4$ second-level threads

Figure 4.11: Synthetic benchmark execution times

compilers execute the benchmark in about 9000 sec.

4.4.2 Face Detection

As already mentioned in Section 4.1, we also experimented with a face detection application, which takes as input an image and discovers the number of faces depicted in it, along with their position in the image. The underlying algorithm is based on a special highly structured neural network topology that performs face detection with extremely high accuracy. The code has been parallelized with OpenMP, utilizing nested parallelism in order to obtain better performance than what is possible with only single-level loop parallelization. The system has been described in detail by Hadjidoukas et al [88].

To accomplish the task of face detection the application exploits a search strategy to detect an unknown number of faces that may exist in an any given image, at every possible location or scale. In order to detect faces at different scales, the input image is repeatedly sub-sampled by a factor of 1.2 resulting in a pyramid of scales. The number of scales depends on the image size and is usually less than 14.

Next, the main loop nest of the application follows; its structure is outlined in Fig. 4.12. In particular, for each scale (this is the first-level loop) a series of convolutional filters and non-linear subsamplings are applied through the three nested for-loops. In the first loop and for every image of this pyramid, the application performs a rough scanning for face candidates by applying the network with a step of 4 pixels in both directions. After the filtering of the images, the results from each

```
for each scale {   /* level 1 */
   for i=1 to 4 {      /* convolve && subsample */
      <body1>
   }
   for i=1 to 14 {     /* convolve && subsample */
      <body2>
   }
   for i=1 to 14 {     /* apply neuron */
      <body3>
   }
}
```

Figure 4.12: Structure of the main computational loop

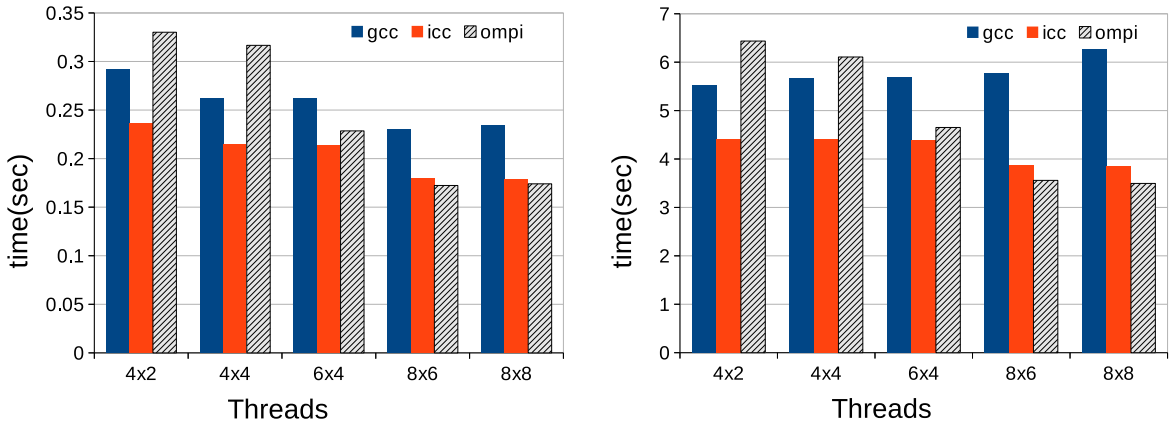
image scale are projected back to the original image. In the second loop, all these positive answers are grouped according to their proximity in image positions and scale, resulting in a list of candidate faces. In the last step of this procedure, every candidate face is processed by the network in a local and finer pyramid of scales and positions around it. This step gives us a more reliable estimate about the positive activation around it and the exact location and scale.

Because of the load imbalance between the different image scales, the level-1 loop is parallelized through a `parallel for` directive with `dynamic` schedule, while for the inner loops a `parallel for` directive with a `static` schedule is applied.

Results on the 8-core system

We run our first set of experiments in the 8-core Xeon machine described previously. We vary the number of participating threads per parallelism level; a configuration of $M \times N$ threads uses M (≤ 8) threads in the first level and N (≤ 8) threads for the second level. In Fig. 4.13 we show the performance obtained when each of the available compilers was used. We do not include results for single-level parallelization ($N = 1$) as they were inferior to what we obtained when $N > 1$. In Fig. 4.13a the application used as input a particularly demanding image which contains 57 faces (the 'class57' image from the CMU test set [93]). For obtaining the results in Fig. 4.13b we processed a series of 161 images with varying sizes and faces, one after the other.

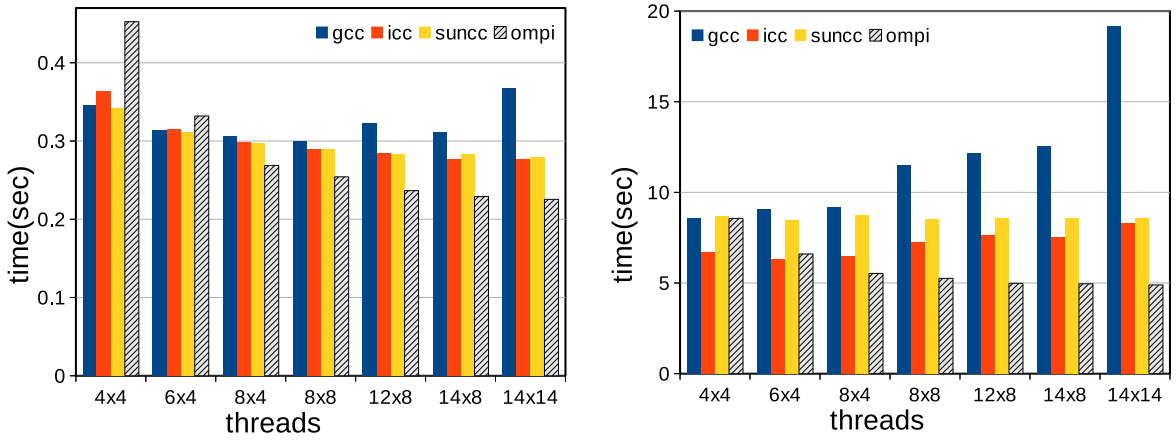
All compilers, except OMPi, are using nested parallelism, spawning $M \times N$ threads, while OMPi uses only M threads that execute $M \times N$ tasks in total. The results lead



(a) For the class57 image

(b) For processing all images

Figure 4.13: Face detection timing results on the 8-core Xeon



(a) For the class57 image

(b) For processing all images

Figure 4.14: Face detection timing results on the 24-core Opteron

to similar conclusions in both plots. When 4 threads are used in the fist level, OMPI exhibits higher execution times for both input images configurations. This is due to the few (4) available threads while all other compilers employ 8 threads in total, potentially exploiting all 8 cores of the system. For 6 first-level threads OMPI performs better than GCC and for $M = 8$ it has the best behaviour overall. ICC has the second best performance as it manages to handle quite well the oversubscription overheads. GCC exhibits the highest execution times for both image configurations.

Table 4.1: Best execution times and comparison with OMPi when processing all images (speedup is calculated in comparison to the best sequential time overall)

Compiler	Sequential time (sec)	Best configuration	Parallel time (sec)	Speedup	OMPi improvement
GCC	29.747	4x4	8.565	3.473	42.90%
ICC	40.173	6x4	6.334	4.696	22.80%
SUNCC	34.364	6x4	8.448	3.521	42.11%
OMPi	29.954	14x14	4.890	6.083	–

Results on the 24-core system

We run a similar set of experiments with the face detection application on the 24-core Opteron server. The number of participating threads in the first level is $M \leq 14$ and for the second level is $N \leq 14$. The limits were based on the observation that the pyramid of different scales contains usually ≤ 14 scales and that the second-level loops produce up to 14 iterations (see Fig. 4.12). In Fig. 4.14 we show the performance obtained for all available compilers. In Fig. 4.14a the application used 'class57' image while the Fig. 4.14b uses the all 161 images.

The figures lead to similar conclusions as previously described, only the differences among compilers are more pronounced. For the 4×4 configuration OMPi exhibits the worst execution times. On the other hand, when 8 or more threads are used in the first level, OMPi exhibits the lowest times for image 'class57'. When all images are used then OMPi performs better than GCC and SUNCC for all thread configurations, and for 8 or more first-level threads it outperforms ICC, too. ICC exhibits the second best performance overall and when processing image 'class57' it attains stable speedups for all thread configurations. For the set of all images ICC exhibits its best behaviour when 6×4 threads are used. GCC gets its best speedup for image 'class57' for 8×8 threads, whereas for the set of all images maximum speedup is shown for 4×4 threads. Finally SUNCC exhibits similar execution times compared to ICC for the image 'class57', while for the set of all images it is faster only when compared to GCC.

For completeness, in Table 4.1 we report the sequential times along with the best performance attained by each compiler. For each compiler, we include the time required for a sequential run, the best observed configuration and the parallel execution time for that configuration. Speedups are then calculated in relation to the

lowest sequential execution time, which is achieved using the GCC compiler. The last column demonstrates the performance improvement OMPi achieves in comparison to each compiler, based on the parallel execution times. It should be clear that our task-based technique outperforms the conventional implementations which utilize nested thread teams.

4.5 Dynamic Transformation

In the previous section we presented some experimental results that showed the benefits of automatically transforming the nested workshare regions into tasks. The great advantage of our technique is that it utilizes only the first-level threads. However this feature may become a disadvantage in the case where only a few threads comprise the first-level team. Furthermore, automatic task generation may utilize fewer than the available processors. Thus, the developer should ensure the presence of enough threads to fully exploit the system resources. On the other hand, nested thread teams lead to oversubscription overheads. To overcome these limitations, we propose a dynamic policy that chooses the best configuration between the number of nested threads and automatically created tasks at runtime. According to this dynamic policy the runtime system is allowed to dynamically choose how to parallelize the nested parallel-for or nested section, i.e. to decide whether to create tasks or threads. For example, whenever a nested loop region is encountered and all processing cores are occupied, then the runtime should create tasks so as to avoid the oversubscription overheads. In contrast, if there are enough available cores then a new team of threads should be created.

4.5.1 Implementation in OMPi

In order to let the user decide the way the runtime should treat the nested parallel-for or nested parallel-sections regions, we added a new environmental variable to OMPi, called `OMPI_PAR2TASK_POLICY`. In the case this variable is set to `TRUE`, all nested worksharing constructs are automatically transformed into tasks. A `FALSE` value forces the original behaviour i.e. creation of nested thread teams. To integrate the newly proposed dynamic policy we introduced a third value, namely `AUTO`. If the `AUTO` policy is selected then OMPi's runtime is allowed to choose the best configuration for a given

Algorithm 4.1 OMPi `AUTO` policy algorithm

```
N ← number of threads requested
K ← number of idling cores
if K == 0 then                                   ▷ No core is available
    ntsk ← N                                     ▷ Create N special tasks
    create_tasks(ntsk)
else if K ≥ N then                               ▷ There are enough available cores
    nthr ← N                                     ▷ Create N nested threads
    create_threads(nthr)
else                                             ▷ There are some available cores
    nthr ← K                                     ▷ Create K threads
    ntsk ← N − K                                 ▷ Special tasks emulate the remaining threads
    create_tasks(ntsk)
    create_threads(nthr)
end if
```

nested workshare region. In particular, the runtime library is allowed to decide to:

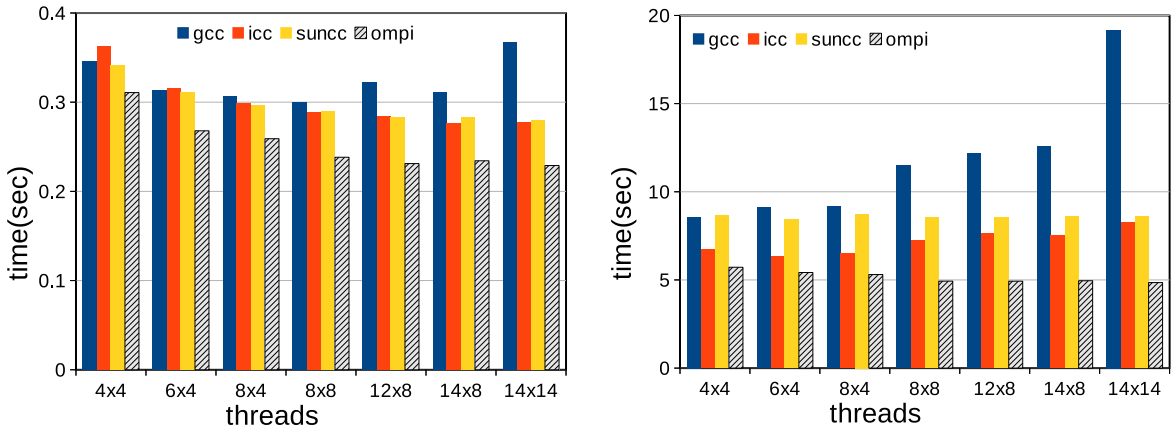
1. Create a normal nested team of threads, or
2. Create a set of special tasks that emulate implicit tasks, or
3. Create a combination of threads and special tasks.

The algorithm for `AUTO` policy that decides the number of nested-level threads and special tasks is given in Alg. 4.1. In essence, the choice is to create special tasks unless there exist idle processing cores. In that case enough threads are created so as to utilize them; the remaining requested threads are turned into special tasks.

By exploiting the `AUTO` policy, the runtime of OMPi can combine its two implicit task execution engines and may utilize all available cores of the system without causing oversubscription overheads. This is feasible because the extra need for nested threads is emulated through the manipulation of special tasks.

4.5.2 Performance of the AUTO Policy

We have conducted a set of experiments in order to assess the advantages of the `AUTO` policy. To do so, we used the face detection application and run it on both the



(a) For the class57 image

(b) For processing all images

Figure 4.15: Face detection timing results, OMPi utilizes its AUTO policy

24-core and the 8-core systems. Here we present only the results from the former but quite similar results were also obtained in the latter system. The results are shown in Fig. 4.15 and they use the same methodology with the ones shown in Section 4.4.2. In fact the only difference here is that OMPi utilizes the `AUTO` policy, so the timing results for ICC, GCC and SUNCC are the same. Again in the experiments we vary the number of participating threads per parallelism level $M \times N$ with $M \leq 14$ and $N \leq 14$. In Fig. 4.15a we show the results of 'class57' image and in Fig. 4.15b we show the results when a set of 161 images is used as input.

In contrast to the previous results, OMPi now has the chance to exploit the same number of cores as the other compilers. In all configurations and for both experiments, OMPi exhibits the lowest execution times. This is achieved due to the adaptive `AUTO` policy which retains the advantages of tasking in larger configurations but can additionally create nested teams of threads so as to utilize enough processing cores in smaller configurations.

Chapter 5

Runtime Support for Multicore Embedded Accelerators

A major part of this dissertation refers to the design of programming models for multicore embedded systems. These systems are computational devices that, contrary to general purpose computers, aim to perform special functions. Embedded systems are met in consumer electronics, auto-mobiles, trains, communication systems, etc. Designing the hardware and software for such systems is a challenging task; in comparison with general purpose computing systems, additional issues arise as for example power constraints, real-time execution constraints, reliability and security. The growing need for greater performance/watt ratio leads to multicore embedded systems which include multiple processors combined with specialized units for specific tasks.

Applications for multicore embedded systems include multimedia processing, real-time response and interaction with humans or certain events:

* Signal processing, e.g. software defined radio, radars or software defined GSM based stations, audio processors. These applications implement the signal processing blocks and execute them in parallel on the multicore SoC.

* Image processing; an example is object recognition as used in content based image indexing, object counting/monitoring, optical character recognition, etc.

* Video processing. Foreground recognition which is employed in traffic applications and self driving cars, constitutes a characteristic example.

Providing a programming model for these heterogeneous systems requires careful and novel designs. We believe that OpenMP can be utilized to offer a productive and efficient programming environment. In this chapter we present the design and implementation of OpenMP runtimes for two embedded system accelerators: the sthorm and the Epiphany.

5.1 STHORM Embedded Multicore Platform

Our group (parallel processing group of the University of Ioannina) along with 26 european research groups and industries participated in the SMECY (Smart Multicore Embedded Systems) project [94], that took place from January 2010 until the end of January 2013. SMECY envisioned that the multi-core technologies will rapidly develop to massively parallel computing environments which, due to improved performance, energy and cost properties, will extensively penetrate the embedded system industry. The mission of the project was to propose and develop parallel programming environments for embedded systems.

The concept of SMECY is based on the simple assumption that a programming tool should be designed in a way that takes into account both the application requirements and the hardware specification. Among the outcomes of the project, presented in [95], were the definition of intermediate representations between front-end and back-end tools defining three different tool chains. These programming tools are designed to support three application domains: 1) radar signal processing, 2) multimedia, mobile, and wireless transmission and 3) stream processing (video surveillance). For each application set they chose as target one of two platforms provided by the project partners; the sthorm provided by STMicroelectronics and the EdkDSP provided by the Institute of Information Theory and Automation in Czech Republic.

SMECY introduced two intermediate representations in order to provide generic parallel expressiveness and compatibility between the various programming models.

The first intermediate representation is a high level model called *SMEC-C* and is based on C with pragmas similar to OpenMP. The lower level representation is called *IR2*, it is also based on C with a set of APIs for communication, resource management and tasks management. The communication API is based on the MCAPI protocol proposed by the Multicore Association [96].

Our contribution as project partner was to design and set the basis for an OpenMP programming environment for the sthorm accelerator. To do so, we had access to a cycle accurate simulator for this platform. Because the actual hardware design and the simulator for the sthorm were developed during the project, we did not actually had a stable environment to work with and from one version to the next we had to alter our designs too.

From our point of view the sthorm is an accelerator attached to multicore host processor. Our novelty is the support of OpenMP functionalities both for the host and the accelerator through unified code written in a single file. To provide the host access and the ability to offload code and data to the accelerator we proposed an extension to the OpenMP specifications based on the *SMEC-C* intermediate representation. On the runtime side we redesigned ompi's libraries and ported them to the execution environment of the sthorm simulator. This way we were able to provide OpenMP v3.0 support for the embedded multicore platform. The result of our work was a functional prototype compiler that takes as input OpenMP v3.0 code with our offloading extensions for a system comprised of a multicore host and the multicore sthorm accelerator. Historically this work preceded the OpenMP device model, which followed with the introduction of v4.0 where new directives allowed the offloading parts of code to an accelerator device.

The rest of the section is organized as follows. In Section 5.1.1 we give details on the architecture of our target MPSoC. Section 5.1.2 describes the design and implementation of our OpenMP infrastructure. Finally, some initial experiments are reported in Section 5.1.4.

5.1.1 System Architecture

The system we target is sthorm [2], formerly known as the ST P2012 platform. The accelerator fabric consists of tiles, called clusters, connected through a globally asynchronous locally synchronous (GALS) network-on-chip where each cluster may

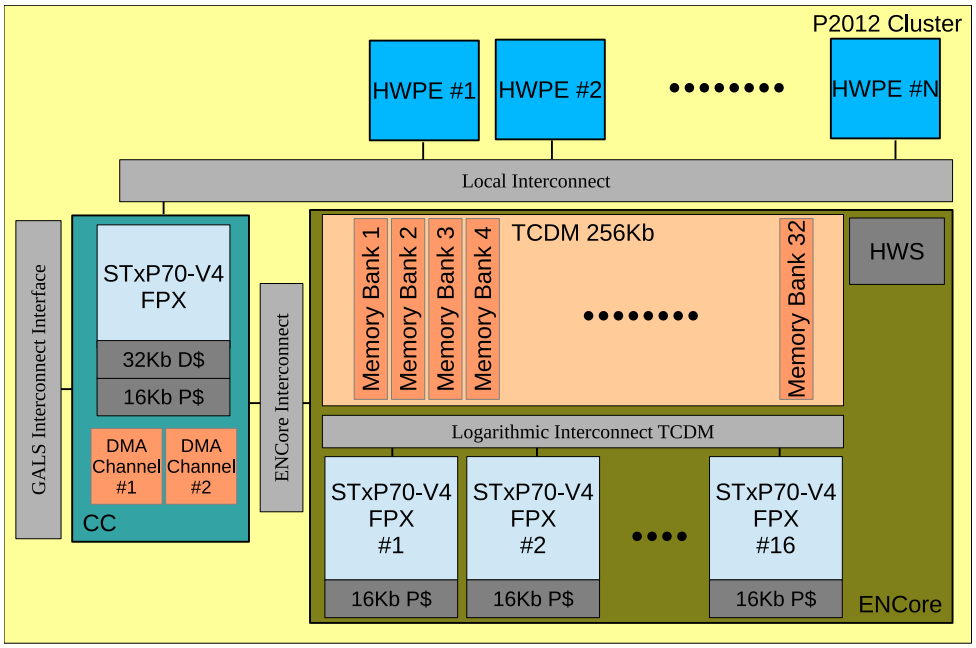


Figure 5.1: STHORM cluster architecture

operate in different clock speed. In our view, this will be a reference architecture for future multicore accelerators as it combines the ability to perform general-purpose computations alongside with specialized hardware, while also offering a scalable interconnection structure that will allow large core counts. The architecture of a single cluster is depicted in Fig. 5.1 and is composed of a multicore computing engine, called ENCore, and a cluster controller (CC). Each ENCore can host from 1 to 16 processing elements (PEs). Each PE is an STxP70-V4 processor, a cost effective and customizable 32-bit RISC core. It has a 32-bit load/store architecture with variable-length instruction set encoding, allowing manipulation of 32-bit, 16-bit or 8-bit data. This particular version (V4) of the processor includes a floating point unit and corresponding instruction set extensions (FPX). Thus the ENCore PEs are full-fledged processors and follow the MIMD execution model, i.e. each one operates independently of the others. This is an important feature that simplifies the design of embedded applications.

Each PE has a 16KB private instruction cache and no data cache. Instead, the team shares a 256KB scratchpad memory called TCDM (tightly coupled data memory). The latter features single cycle accesses and is divided into 32 banks in order to reduce the probability of conflicts, when several PEs try to access it simultaneously. The cluster is coordinated by the cluster controller (CC) which is also an STxP70-V4 processor. The CC has a 16KB instruction cache, an additional 32KB of local data memory and also two DMA channels for memory copies.

Each ENCore cluster is provided with a Hardware Synchronizer (HWS) engine which provides hardware synchronization support for efficient implementation of semaphores, locks and barriers. It also includes an event and interrupt generator. The accelerator has also provisions for additional Hardware Processing Elements (HWPES) to efficiently support applications that require specialized hardware (for example audio/video decoders).

Fig. 5.2 shows the configuration of a STHORM SoC which consists of four clusters. The system includes a 1MB of L2 (or fabric) memory used for instruction and data, also shared among the clusters. In addition there exists a special unit called fabric controller (FC). The FC is similar to the CC processor and is responsible for interactions with the off-chip host and for instrumentation and coordination of the clusters. The overall memory organization follows the partitioned global address space (PGAS) model. This means that all processing elements can access all SoC memories but with varying delays depending on where the PE and the data are located. For example it is possible for a PE of one cluster to perform load and store instructions directly from/to the fabric memory or a remote TCDM memory of another cluster, albeit slower as compared to accessing its own local memory.

5.1.2 Implementing OpenMP

In order to provide an implementation of the OpenMP model for the STHORM architecture which however is general enough to be ported to similar future MPSoCs, we relied on source-to-source compilation. In particular we based our implementation on the OMPi compiler. An initial observation is that an accelerator is usually not a stand-alone device—it is rather a back-end system attached to a host, which could also be a multicore processor. There is a conceptual difficulty as to where and how the OpenMP programming model is to be applied: at the host side or at the accelerator side? That is, should OpenMP threads live in the host processor and generate simultaneous (sequential) computational requests towards the fabric, or should the parallelism (OpenMP threads) be created and executed within the accelerator? Our design decision was to be as general as possible, so as to have the greatest flexibility in supporting general MPSoC architectures, and as programmer-friendly as possible, so as to let the programmer express parallelism in any way convenient. Consequently the goal of our compilation chain was to support OpenMP at both the host and the

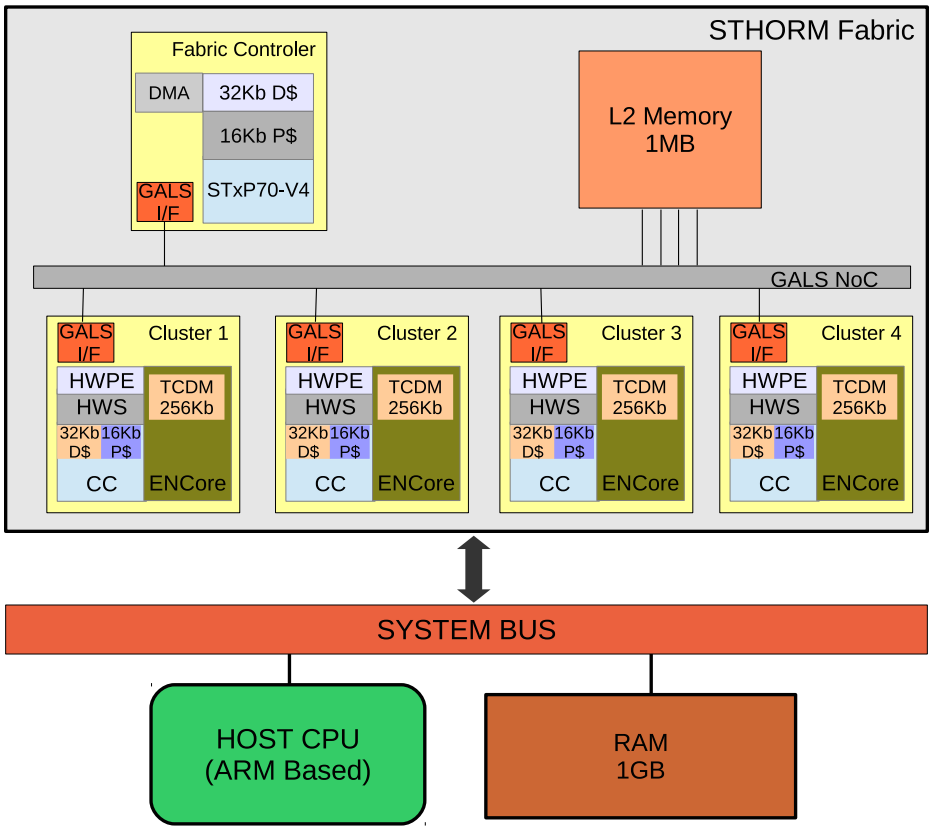


Figure 5.2: STHORM SoC Configuration

accelerator levels.

OMPi's compilation chain for MPSoC accelerated systems is shown in Fig. 5.3 and is composed of three phases. During the first phase and after preprocessing the source file, the compiler divides it into two new OpenMP files. The first file contains the code to be executed by the host processor and the second one has the kernel functions to be offloaded and executed on the MPSoC fabric. This target separation phase analyses the function call graph and packs all dependent functions into the fabric code. During the second phase the actual transformations take place; the separated files are transformed into intermediate pure C code. The intermediate file for the host embeds calls to the standard OMPi runtime designed for the support of shared memory systems, which has been extended to provide the necessary primitives for communication with the accelerator. The intermediate file for the fabric is augmented with calls to a new runtime that supports OpenMP execution within the accelerator. During the final phase, system back-end compilers are used to link the intermediate object files with the appropriate runtime libraries and to create the two executables. The executable for the fabric side is delivered as a shared library.

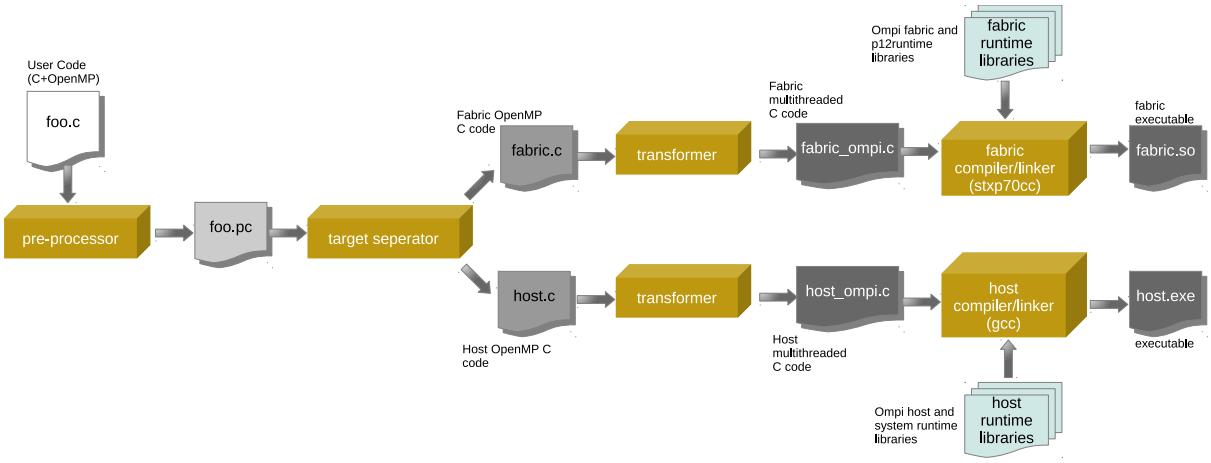


Figure 5.3: OMPi compilation chain

Programming Model

From a programmer's point of view the application consists of two parts to be executed, respectively, by the host and the accelerator, which however are presented in a unified code. The accelerator part is a collection of C functions that are appropriately annotated. Function annotation occurs at the call site. The annotation model we follow is based on the SME-C [97] representation that has been proposed by the SMECY project consortium. In particular, a function call preceded by the following pragma:

```
#pragma smecy map(HWunit) [arg[[,]arg]...]
 <function call>
```

causes the called function (or kernel in OpenCL terms) to be offloaded to the accelerator and executed (mapped) at a specific hardware unit. Valid hardware units are the PEs, the cluster controller and the fabric controller. Execution by a PE is described by the pair (PE, n) where n is the id of the PE that should execute the offloaded function. The optional 'arg' clauses describe the size and direction of function arguments (input/output/both). In the offloaded function code, OpenMP directives are allowed which dynamically spawn parallelism within the fabric. In addition OpenMP in the rest of the user code is translated as parallelism to be generated at the host. Multiple host threads may offload multiple functions for simultaneous execution on the fabric.

An example is shown in Fig. 5.4. The execution of this code begins at the host where a team of four threads is created (line 7), and is visualized in figure 5.5. The thread with id 3 that meets the offload directive, forces the accelerator to execute the kernel function *foo* and suspends its execution until the kernel is finished (lines

```
1  void foo(int A[256]) {
2    #pragma omp parallel
3    {...}
4  }
5
6  int main(void) {
7    #pragma omp parallel num_threads(4)
8    {
9      if (omp_get_thread_num()==3) {
10       #pragma smecy map(PE,0) arg(1,inout,[256])
11       foo(A);
12     }
13     else {
14       ...
15     }
16   }
17 }
```

Figure 5.4: Example of kernel (foo) offloading

10-11). After the accelerator is enabled, the PE with id 0 begins executing the kernel. When this PE encounters the parallel directive in line 2 it creates a team of 16 PEs to execute the code in line 3. After the kernel's execution, control goes back to the host thread with id 3 in order to resume its work. Notice that the programmer doesn't have to deal with special glue code for the accelerator management (i.e. discover devices, enable/disable units, load binary etc.) because this functionality is provided in the compiler generated code and its runtime library.

Data Management

From the programmer's perspective the accelerator memory hierarchy consists of three types of memory:

1. a *scratchpad* memory area (implemented by the TCDM in Fig. 5.1), which is fast and serves to store data used repeatedly by the PEs. For the STHORM platform its size is 256KB per cluster.

2. a slower *fabric* memory, outside of the clusters but shared among them, with a size of 1Mb and

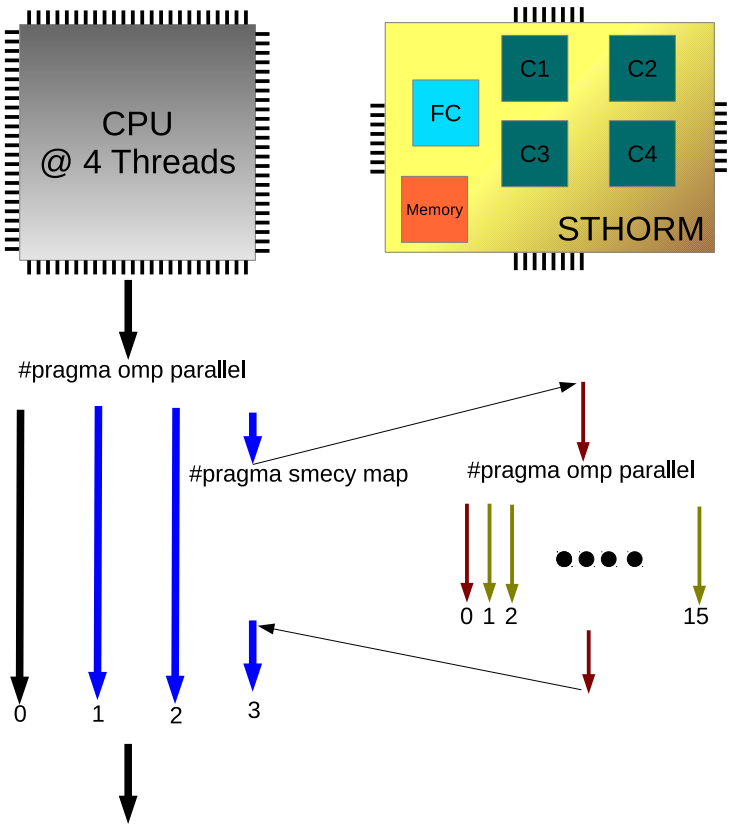


Figure 5.5: STHORM execution model

3. a *shared* memory area, accessible by both the host and the accelerator. This is part of system's RAM used for communication between the host and the fabric. Its size can be large, and so is its access time from any PE.

The arguments of the offloaded function are stored in the shared memory area. In the example of Fig. 5.4 the compiler transparently generates code to

(i) Allocate space in shared memory for a copy of the 256 elements of array A

(ii) Copy A to the allocated shared memory area

(iii) Execute the kernel

(iv) Copy the data back from shared memory to A (since it is both an input and an output of the function) and

(v) Free the shared memory space

Because of the need to utilize variables by multiple kernels, instead of transferring them back and forth multiple times, we provide a new directive that allocates such variables on the SoC and stores them there for the whole program execution:

```
#pragma device_global(var [,var [, ...]])
```

The enlisted variables will be stored in fabric memory. Finally, in order to allow programs exploit the full memory hierarchy, calls for allocating/freeing memory as well as calls for copying memory areas using the underlying DMA mechanisms are provided:

(1) `ompi_local_malloc()`

(2) `ompi_local_free()`

(3) `ompi_sthorm_dma_ext2loc_memcpy()`

(4) `ompi_sthorm_dma_loc2ext_memcpy()`

The first two functions are used to allocate/deallocate space within the scratchpad memory of a cluster, while the other two are used for DMA transfers between shared memory and scratchpad memory.

5.1.3 Runtime Support

Runtime support is provided on top of native libraries which provide basic operations such as feeding jobs to PEs, CCs and FC, memory management, DMA transfer primitives and synchronization facilities.

The two types of processing units, CC and PE have a discrete role in program execution. PEs execute code in a MIMD manner and have limited access to the cluster hardware. PEs can execute reads and writes in memory, request DMA transfers, increment/decrement atomic counters and send/receive signals. In contrast, the CC has full access to all hardware and can additionally allocate/deallocate space in the local, the fabric and the scratchpad memory, allocate/deallocate atomic counters, events, feed itself and PEs with computations, communicate through mailboxes with other CCs and with the FC. Therefore, from a programmer point of view a CC is a master unit that sends/receives requests to/from other CCs, prepares hardware, distributes work to the PEs, and supervises program execution. On the other hand the PEs are 'slave' units that receive job requests and execute them in a preallocated data environment.

The only way a PE can have an active role is by instructing the CC to perform a job. This way, when a PE wants to execute a privileged operation, it prepares a

request, sends it to CC and waits until the CC satisfies it. These privileged operations include: allocate/deallocate memory, give jobs to other PEs and allocate/deallocate DMA requests.

EECB Management

Throughout the execution of an OpenMP application ompi associates a block of special data called Execution Entity Control Block (EECB) with every OpenMP thread it manages. The EECB contains all the information needed by the runtime in order to schedule the thread, including the size of its team, the thread's ID, its nested parallel level, a pointer to its parent thread EECB (thus a tree of EECBs is formed at runtime) etc. Whenever a thread starts the execution of a parallel region, a new EECB is assigned to it, which is later freed when the team is disbanded.

An important design decision was the placement of EECBs. These structures are constantly accessed during program execution, so their placement in scratchpad memory was the only solution for guaranteed performance. On the other hand, EECBs may occupy significant space in some execution scenarios, for example in applications that perform nested parallelism in great depths or have a certain number of nested parallel teams which are constantly formed and deformed. To avoid the successive PE requests to the CC and possible overflow of scratchpad memory we designed an EECB placement strategy that uses the minimum memory possible in common OpenMP application cases.

Our strategy uses the following scheme: in the scratchpad memory we use a preallocated table of 16 EECBs and a dynamic list that will be used for all active EECBs. We also use a dynamic list placed in the fabric memory that will be used in case of nested parallelism. The preallocated table and the two lists mentioned are also used for recycling EECB data. The core idea is to always maintain the EECBs of active threads in scratchpad memory in every execution scenario. In non-nested parallelism cases the parent data is stored in the scratchpad list. In contrast, during the execution of nested teams, the parent data is placed in the fabric memory list.

This hybrid scheme is shown in Fig. 5.6. Here we show the EECB placements during the execution of a kernel in three cases: 1) when a single PE executes this kernel, 2) when this PE forms a team of 4 OpenMP threads and 3) when a PE of this team creates a new nested team of 3 OpenMP threads. In case 1), when a PE starts the execution of a kernel it is assigned an EECB from the preallocated table in scratchpad

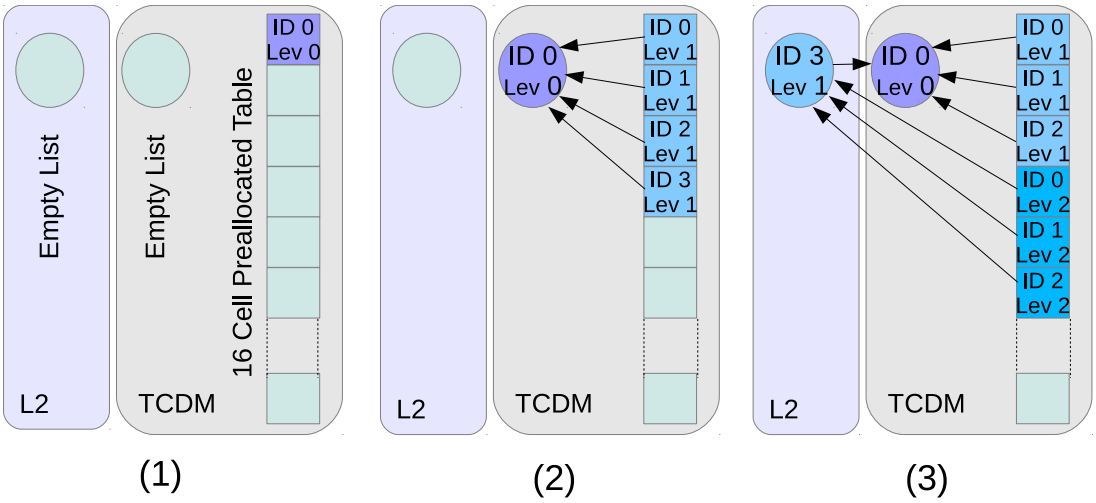


Figure 5.6: EECB placement. 1) When a single PE executes a kernel, 2) when this PE forms a team of 4 OpenMP threads and 3) when a PE of this team creates a new nested team of 3 OpenMP threads

memory. In case 2), this PE suspends the execution of its current job and participates in the new team of threads. To do that it becomes a parent, acquires a new EECB from the list in the scratchpad memory (to use as child), it copies the parent EECB data there and reuses its old EECB. The other threads in the team use EECBs taken from the table. All EECBs point to the parent in scratchpad memory list. Therefore in case of non-nested parallelism all data (parent and child) is placed in scratchpad memory. Finally in case 3), a thread of this team wants to form a nested parallel team. To achieve this, it acquires an EECB from the fabric memory, copies the parent EECB data there and reuses its old EECB. Other threads in the team use EECBs taken from the table in the scratchpad memory. Now the EECBs of child threads point to the EECB in the fabric memory list. The accesses in fabric memory are slower but this solution can efficiently support nested parallelism without wasting memory resources. During the deformation of parallel teams, the reverse procedure is followed and EECBs from the lists are copied back to the EECBs in the table. This way active EECBs are always placed in the scratchpad memory.

Parallel Regions

Our runtime treats an OpenMP parallel region as a group of implicit tasks that are executed in parallel by PEs. When a PE executing a kernel meets a parallel directive, it suspends the execution of its current job and sends a request to the CC in order to

supply other PEs with the appropriate implicit tasks. After that it allocates a new EECB with the procedure mentioned above and becomes the master PE of the newly created OpenMP team, executing its implicit task (job). Simultaneously, other PEs receive the request of the CC and start executing their jobs. When a PE finishes its job (i.e. exits the parallel region) it notifies the CC and then falls to sleep mode. When all PEs have finished their jobs, the CC informs the master PE so as to safely do its bookkeeping, return to its old EECB and resume its suspended job. We chose to utilize the CC in order to avoid the use of locks and to efficiently exploit CC's idle time.

The PEs need memory to use as stack in order to execute a job. OMPi allocates space for these stacks in the scratchpad memory and then uses a recycling mechanism to avoid unnecessary allocations and deallocations. The default stack size is 4KB for the master thread (PE 0) and 512 bytes for all other threads, occupying 11.5KB for a team of 16 concurrent threads. Depending on the application, larger stack sizes may be required; the actual amount of stack space is user-controllable by a standard OpenMP environmental variable (`OMP_STACKSIZE`).

In the current implementation, the total number of threads that can co-exist in a cluster is limited to 16. Therefore, we can have one team of up to 16 threads or two concurrent teams of up to 8 threads each etc. These parallel teams can result from the execution of one or multiple offloaded kernels and at different levels of parallelism. Our runtime can support the concurrent execution of up to 16 kernels where each kernel utilizes only one PE. In case all PEs are busy and a PE wants to create a new OpenMP team, then CC will deny its request and the PE will execute the code of the parallel region serially.

Tasking Infrastructure

Efficient tasking support for the OpenMP model requires a sophisticated runtime and a rather generous amount of memory. On the other hand it is uncommon for embedded applications to have deeply nested tasking behaviours or create large numbers of nested parallel teams with large memory requirements. Given the limited memory resources of an MPSoC, we designed a lightweight tasking subsystem. The original implementation of tasking in the OMPi infrastructure [8] is targeting general purpose SMPs and multicore systems with abundant resources, and is clearly unsuitable for an MPSoC like the one under consideration here.

In OMPi all task bookkeeping information, i.e. task status, task id, number of chil-

dren etc, are stored in structures called task nodes. For better utilization of the limited memory we use a preallocated table of task nodes that it is used as a recycling bin. This table is protected by a lock and is located in the scratchpad memory. At task creation the PEs use this table to allocate space for new tasks. After the task execution the corresponding node is recycled. The procedure of allocating and deallocating nodes includes only the altering of a task node's field.

Our tasking subsystem uses two important structures both located in a cluster scratchpad memory. The first one is a shared FIFO queue with a fixed size which is used to store pending tasks of all parallel teams that are executed within a cluster. This queue is protected by a single lock. The second structure is composed of private queues, one for each PE. These queues have a particularly small size and are also protected by a single lock.

We have extended the OpenMP task directive with an extra optional clause so one can request explicitly by what PE and on which cluster a task should be executed:

```
#pragma omp task on(cluster_id, pe_id)
```

This extension gives the user the ability to control task placement explicitly, since by default OpenMP tasks may be executed by any thread. This may also prove useful in increasing code locality. In case a PE meets this new task directive it enqueues the new job to the appropriate private queue.

There are three scheduling points in our tasking implementation. The first one is the aforementioned new clause. The other two are the `taskwait` and `barrier` clauses where PEs search for pending tasks in the global and their private queues and execute them. At `taskwait` a PE executes child tasks defined in the context of its current task, while at a `barrier` a PE will execute all tasks generated by its current OpenMP team.

Thread Synchronization & Locks

The HWS (hardware synchronizer) of the accelerator provides a small number of atomic counters (ACs) and an even smaller number of events. However, the tasking infrastructure is in great need for fine grain synchronization and a straightforward implementation of locks (using 1 AC per lock) is not feasible, because the atomic counters provided are not enough to cover the needs of typical task-based OpenMP applications. To solve this problem we present a novel locking scheme that provides an unlimited number of locks, using minimal hardware resources.

The basic idea is to use a small fixed number of ACs and map all program locks to

them. Locks are implemented by plain integers. Access to a block of those integers is then protected by a block lock implemented by an AC. All locks share a global event. Thus a PE first access the block AC and then set/unset the actual (integer) lock. The lock handling mechanism is shown in Fig. 5.7.

For locking, a thread first tries to get access to the lock by constantly increasing the value of the AC. When an AC value is 0 a thread can access the lock data, otherwise some other thread is making changes. After getting access it checks the value of the actual integer lock (`locked`). If `locked` has a value of 0, then it locks it by setting its value to 1 and releases the AC by setting its value to 0. In case `locked` is 1, the thread goes to sleep and waits for the event. For unlocking, a thread again gets access through the AC, unlocks the actual lock by setting `locked` to 0, releases the AC and signals the event to wake up any sleeping threads.

An interesting part is the initialization procedure of the lock data. The AC field is initialized through the assign_AC function. This function uses a preallocated table of ACs and returns the next available AC. If all ACs are being used then the same ACs are used again, forming thus blocks of locks protected by the same atomic counter. In order to equalize the load on the block ACs, we allocate the actual integer locks in a round robin manner among the blocks.

OpenMP barriers are also implemented using ACs. An atomic counter is used to count the number of threads that have reached the barrier. At the barrier all waiting threads keep looking for pending tasks to execute until the last one releases them.

Summary

The runtime infrastructure we presented has been highly optimized in order to provide full and efficient OpenMP support with a minimal footprint. The total memory requirements are approximately 9.5KB for a team of 16 threads; this includes everything (i.e. storage for EECBs, task pools, locks, etc) except the thread stacks which by default occupy 11.5KB as discussed above. Consequently, in a typical run our library consumes only about 21KB (≈8%) of the TCDM memory, leaving a large portion of it available for the application.

The current implementation can dedicate only one STHORM cluster per offloaded kernel. This means that all four clusters can be utilized albeit by four different kernels. A single kernel may thus only utilize up to 16 PEs of a single cluster.

```
void Initialize(lock l) {
  l.lockAc = assign_AC();
  l.lockEvt = globalEvt;
  l.loked = 0;
}

void Unlock(lock l) {
  /* Grant access to lock */
  old = 0;
  while (old != 1)
   old = increase(l.lockAc);

  /* Unlock, wake up PEs */
  l.locked = 0;
  set_value(l.lockAc, 0);
  raise_evt(lock.lockEvt);
}

void Lock(lock l) {
  while(1) {
    /* Grant access to lock */
    old = 0;
    while (old != 1)
      old = increase(l.lockAc);

    if (l.locked == 0 ) {  /* Try to lock */
      l.locked = 1;
      set_value(l.lockAc, 0);
      break;
    }
    else {                 /* Go to sleep */
      set_value(l.lockAc, 0);
      wait_event(l.lockEvt);
    }
  }
}
```

Figure 5.7: Code for lock initialization, locking and unlocking

Table 5.1: Overheads for various operations (16 threads)

Operation	# Cycles
kernel offload	6283
omp parallel	37750
omp for	8427
omp barrier	11941

5.1.4 Preliminary Experimental Results

We have conducted a number of experiments in order to test the efficiency of our OpenMP platform in the context of the sthorm accelerator. An actual implementation of the sthorm platform was not available. Consequently, for our experiments we utilized a cycle-accurate simulator provided by ST Microelectronics. The simulator is accompanied by a software development kit. Finally, the host processor consists of a dual core ARM Cortex cpu. After the simulation of an application a detailed trace file is produced for the execution steps within the MPSoC. We utilized a provided performance analyser tool to extract information from these trace files.

First we present the cost for some crucial operations of the runtime system. In particular, in Table 5.1 we provide the overheads (in cycles) for offloading a kernel to the fabric and for three important OpenMP constructs: creation of a parallel team, loop worksharing and team barrier. The offloading overhead is measured by repeatedly submitting empty kernels for execution through a `smecy map` directive and counting the average number of cycles. For the last three we considered a team of 16 threads and followed the method of the EPCC benchmarks [92]. The larger overhead of the parallel construct is justified because of the extensive communication required among the pe that encounters the parallel region, the cc and the rest of the pes.

Next, we present performance results for three applications: matrix multiplication, Laplace equation solver and calculation of the Mandelbrot set. We parallelized these kernels using OpenMP within the offloaded code and executed them in a cluster of the accelerator. The default stack sizes were used (512 bytes per worker and 4KB for the master thread) and the data sets were chosen so as to fit within the scratchpad memory.

In Fig. 5.8 we plot the speedup curves for the first two applications. Matrix multiplication uses the standard triple loop computation for matrices of 64×64 floats.

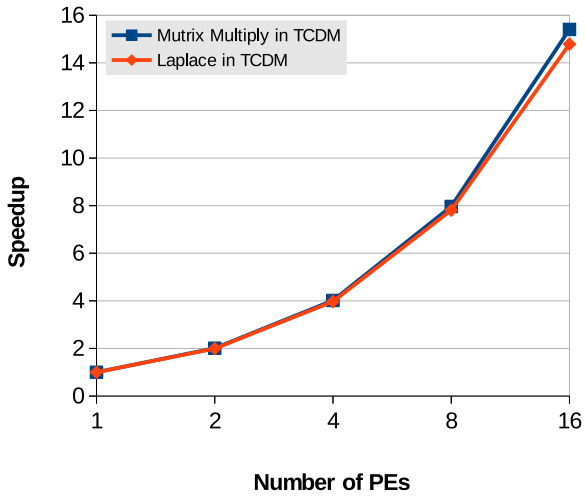


Figure 5.8: Performance of matrix multiplication and the Laplace equation solver

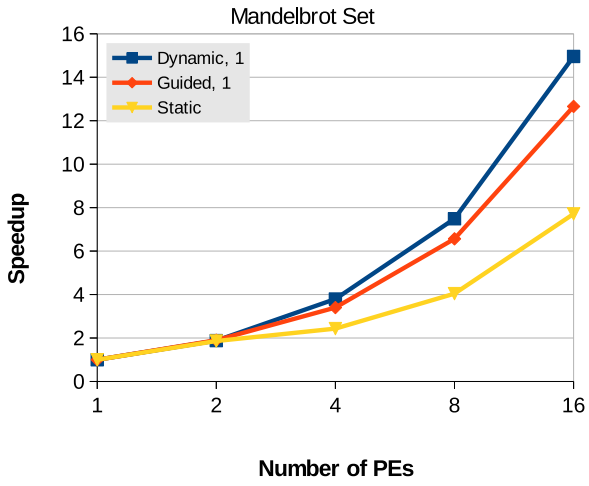


Figure 5.9: Speedup for the Mandelbrot set with a variety of loop schedules

All three matrices (including the result) fit in the scratchpad memory. It is worth mentioning that the only code modification during the parallelization process was the addition of one extra line of a `#pragma omp parallel for` directive. This greatly shows the simplicity and high level parallelization possible by OpenMP. The same figure includes the speedup curve of a simple Laplace equation solver with a 4-point stencil operation and 100 iterations. The parallelization technique utilizes two matrices of 162×162 floats, totaling 205KB which fits with the available TCDM space. In both cases, we observe a close to ideal behaviour.

Finally, Fig. 5.9 shows the performance of the Mandelbrot set calculation for an image of 362×208 pixels (occupying $\approx$220KB). We plot results for different scheduling

policies of the OpenMP `for` construct. Due to the highly unbalanced iteration load, the worst behaviour is exhibited by the `static` schedule while the `dynamic` schedule proves to be the best policy, achieving almost linear speedup. The above results demonstrate the efficiency of our runtime infrastructure.

5.2 Epiphany Accelerator

OpenMP, the de facto standard for shared-memory programming has been augmented with directives that target arbitrary accelerator devices. In the spirit of OpenACC [43], OpenMP 4.0 provides a higher level directive-based approach which allows the offloading of portions of the application code onto the processing elements of an attached accelerator, while the main part executes on the general-purpose host processor. What is important is that the application blends the host and the device code portions in a unified and seamless way, even if they refer to distinct address spaces.

In this section we present our design and implementation of an OpenMP runtime for the Epiphany accelerator of the Parallella board. It is the first OpenMP implementation for this particular system and also one of few OpenMP 4.0 implementations in general. Our implementation supports concurrent execution of multiple independent kernels. In addition it allows OpenMP directives within each offloaded kernel, supporting dynamic parallelism within the Epiphany. The rest of the section is organized as follows. In Section 5.2.1 we give a necessary brief description of the device directives, although a more comprehensive presentation follows in Chapter 6. Next in 5.2.2 we summarize the Parallella board architecture along with its native programming models. We describe our prototype implementation in detail in Section 5.2.3, while in Section 5.2.4 we present some performance measurements.

5.2.1 Introduction to OpenMP Device directives

The `target` directive is used to transfer control flow to a device. The code in the associated structured block (kernel) is offloaded and executed directly on the device side, while the host task waits until the kernel finishes its execution. Each `target` directive may contain its own data environment which is initialized when the kernel starts and freed when the kernel ends its execution. In order to avoid repetitive creation and deletion of data environments, the `target data` directive allows the definition of

a data environment which persists among successive kernel executions. Furthermore, the programmer can use the `target update` directive between successive kernel offloads to explicitly update the values of variables which are shared between the host and the device.

The memory for the data environment of a device is regarded as an autonomous extension of the OpenMP memory model. The data environment can be manipulated through `map` clauses within `target data` and `target` directives. These clauses determine how the specified variables are handled within the data environment. When an `alloc` map type is used an uninitialized variable is defined, whereas with a `to` map type the variable is additionally initialized from the value of the corresponding host variable. If variable is mapped as `from` then an uninitialized device variable is defined; when the specified directive region finishes, the value of the device variable is copied back to the original host variable. If no type is specified or the type is `tofrom`, the variable is considered mapped as both `to` and `from`. Finally, the variables declared within `declare target` directives are also allocated in the global scope of the target device, and their lifetime equals the program execution time.

5.2.2 Parallella Board Overview

The Parallella-16 board [5] is an 18-core credit card sized computer and comes with standard peripheral ports such as USB, Ethernet, HDMI, GPIO, etc. The computational power of the $99 board comes from its two processing modules. The main (host) processor is a dual-core ARM Cortex A9 with 32 KiB L1 cache per core and 512KiB shared L2 cache, built within a Zynq 7010 or 7020 SoC. The other is an Epiphany 16-core chip which is used as a co-processor. The board has 1 GiB of DDR3 RAM, addressable by both the ARM CPU and the Epiphany. The former runs Linux OS and uses virtual addresses while the latter runs no OS and has a flat, unprotected memory map.

The Epiphany co-processor offers an impressive power efficiency that can reach up to 70 GFLOP/Watt, depending on the chip version. Two configurations of the Epiphany co-processor are currently available: the Epiphany-16 (with 16 cores and a 4×4 mesh NoC) and the Epiphany-64 (with 64 cores and an 8×8 mesh NoC). Although our discussion here holds for both versions, we refer mostly to the first one since it is widely available and is what our board contains. This particular chip is clocked at

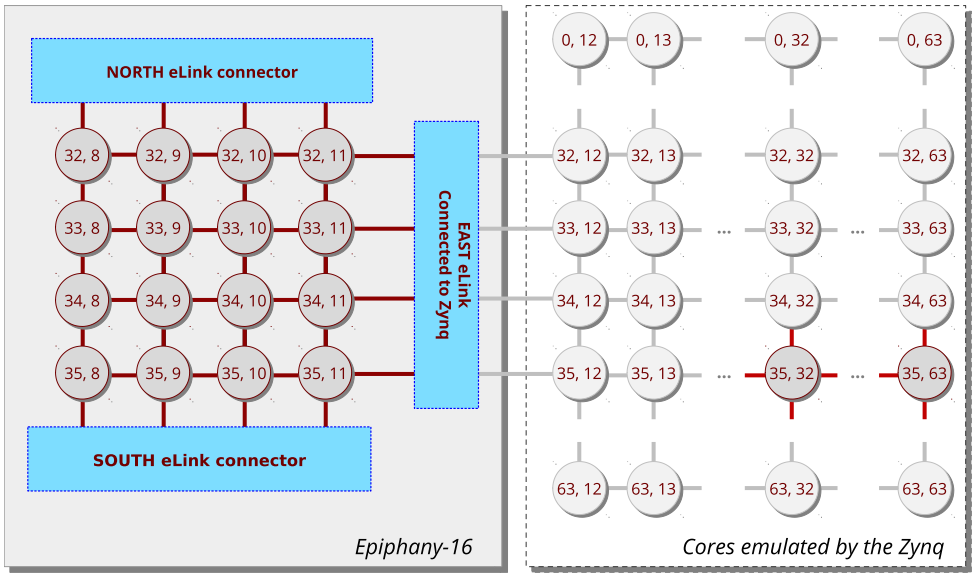


Figure 5.10: The Epiphany mesh in a Parallella-16 board

600MHz and has a peak performance of approximately 25 GFLOPS (single-precision) with a maximum power dissipation of less than 2 Watt.

The architecture of the Epiphany is designed around a 64×64 mesh interconnect, so (in theory) systems with up to 4096 Epiphany cores (ECORES) are possible, by combining 16- and 64-core chips. On the Parallella-16 board, the Epiphany chip is pinned on a 4x4 submesh of the virtual 64x64 mesh whose north-west coordinates are (32, 8), as shown in Fig. 5.10. The chip has four eLinks (west, east, north and south), that may be used to interconnect it with other chips. In the current version of the Epiphany-16 chip the west eLink is inactive and the east eLink is connected to the Zynq host. Notice that the mesh NoC actually contains three separate meshes: the fast *cMesh* for writing on-chip memory, the *xMesh* for off-chip writes and the slowest *rMesh* for reading remote memory.

Each ECORE is a 32-bit superscalar RISC processor, capable of performing single-precision floating point operations, equipped with 32 KiB local scratchpad memory and two DMA engines. All ECORES share a 32-bit address space with each one owning a 1MiB unique addressable slice; the scratchpad memory provides physically 32KiB of this slice. All memory is available through regular load/store instructions.

The Zynq, which is connected to the east eLink of the Epiphany, is perceived as the eastern part of the mesh. Based on the column-first routing scheme of the NoC, the Zynq can emulate the memory space of the cores in the 52 leftmost columns of the 64x64 virtual mesh, giving access to most of the board RAM to the Epiphany. A

32-MiB portion of the system RAM is left outside the Linux virtual memory manager area. From the Epiphany side it corresponds to the 32 cores located in coordinates from (35, 32) to (35, 63). This is designated as *shared memory* and is physically addressable by both the ARM and the Epiphany.

The Epiphany Software Development Kit (ESDK) is a programming tool for the Epiphany accelerator [61], which includes a C compiler and runtime libraries for both the host (EHAL) and the Epiphany (ELIB). A typical C program that utilizes the ESDK adheres to the following pattern: Initially the host executes some initialization and the sequential part of the application. Next, in order to offload code (kernel) to the co-processor it

(a) initializes the Epiphany

(b) prepares the shared memory with all the data needed for the computation

(c) forms a workgroup of ECORES and

(d) triggers the execution of the kernel

All host-ECORE communication occurs through the shared memory.

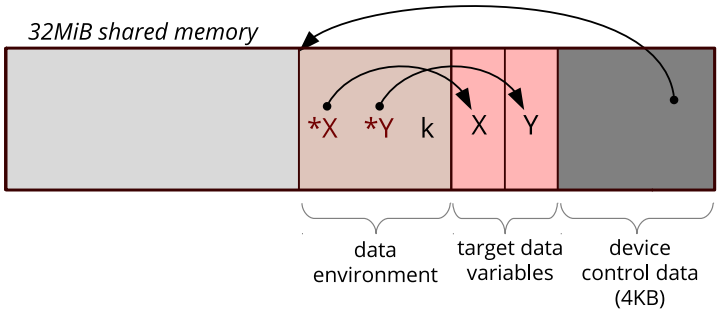
5.2.3 Implementing Runtime Support for the Epiphany

We based our implementation on the OMPI OPENMP compiler. At the host (Zynq) side the runtime system consists of two parts; the first is a full-fledged OPENMP runtime library, part of the regular OMPI infrastructure, necessary for supporting execution on the two ARM cores. The second part provides additional functionality, which is required for controlling and accessing the Epiphany device.

The communication between the Zynq and the ECORES occurs through the shared memory portion of the system RAM as described earlier. The shared memory is divided in two sections, see Fig. 5.11(b). The first section is called Device Control Data (DCD) area, and it has a fixed size of 4KiB; it is used transparently by OMPI for kernel coordination and manipulation of parallel teams created within the Epiphany. The second part is used for storing the kernel data environments and part of the tasking infrastructure of the Epiphany OPENMP runtime described later. More specifically, during the preparation for offloading a kernel, a region is allocated to store the data environment of the kernel. This contains variables or pointers to variables which

```
1 int X[10], Y[10];
2 int k;
3
4 #pragma omp target data map(X,Y)
5   #pragma omp target map(to:k)
6   {
7     /* Kernel code */
8   }
```

(a)



(b)

Figure 5.11: Shared memory organization

appeared in enclosing `target` or `target data` constructs and are not stored in the local memories of the eCORES. An example is shown in Fig. 5.11(a). Variables `X` and `Y` in line 4 are annotated as `tofrom`. This causes a copy of each one to be created in the shared memory. In line 5 the variable `k` is annotated as `to` and along with two pointers to `X` and `Y` form the data environment of the kernel. The beginning of the data environment is stored as a pointer in DCD, and is used by the kernel when starting its execution. All the above are stored at the higher end of the shared memory, leaving the lower end available for the programmer (e.g. for storing libraries which do not fit in the eCORE local memories).

In order to be able to control the eCORES independently through eLIB calls, the initialization phase creates 16 workgroups, one for each of the available Epiphany's cores and puts them to the idle state for energy and thermal efficiency. For offloading a kernel, the first idle core is chosen and the precompiled object file is loaded to it for immediate execution. Because the current version of eHAL does not provide a way for an eCORE to notify directly the host for kernel completion, a special region of the DCD is designated to store special flags set by the eCORES. The DCD infrastructure has a thread-safe design; this allows multiple host threads to offload multiple independent kernels concurrently onto the Epiphany.

OpenMP Within the Epiphany

The eCORES do not execute any operating system and there is no provision for creating and handling dynamic parallelism (e.g. threads) within the Epiphany chip. In addition, the 32KiB local memory of each eCORE is quite limited, unable to handle sophisticated OpenMP runtime structures in addition to application data. As such,

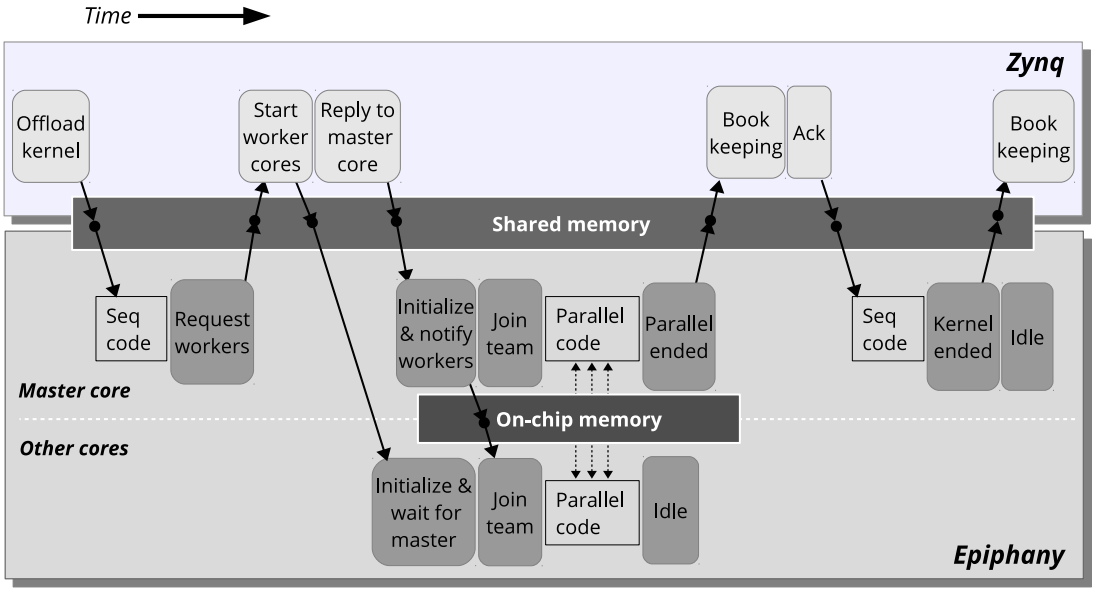


Figure 5.12: Offloading a kernel containing dynamic parallelism

supporting OpenMP within the device side of the board is non-trivial.

The creation of a parallel team within an offloaded kernel is depicted graphically in Fig. 5.12. When a kernel is offloaded to a specific eCore, the core executes its sequential part until a parallel region is encountered; the core will create a new team and become the master of the team. Because only the host can activate other Epiphany cores, the master core sends a request to the host through the device control data (DCD) section in shared memory, requesting the activation of a number of cores. The host-side thread which offloaded the kernel will activate as many-cores as possible to satisfy the master request. A copy of the same kernel is then offloaded to the newly activated cores. The activated cores begin their execution by fetching all the appropriate information regarding the parallel team and its master core from the DCD section in shared memory. Immediately after that they spin waiting for the master to signal the execution of the parallel code. Once all required cores have been activated, the master has access to the actual team size and the coordinates of the team cores. A local flag is then set to release the team cores and let them execute the parallel region. During the parallel code execution all synchronization between the cores occurs through their fast local memories. When the region completes, the cores return to the idle, power saving state, while the master core informs the host thread about the termination of the parallel team. The host marks the idling cores as available for future use, and sends an acknowledgment to the master. The latter

continues with the rest of kernel code.

We note that another, possibly faster, strategy for supporting dynamic parallelism would be to have all eCORES loaded with the kernel(s) in advance and spin, waiting for the master to signal them which kernel to execute. However, this would increase power consumption dramatically and thus we did not pursue it further.

To support the OPENMP worksharing constructs (`single`, `for`, `sections`), the infrastructure originally designed for the host was trimmed down to a minimum so as to minimize its memory footprint; this is linked and offloaded with each kernel. The corresponding coordination among the participating eCORES utilizes the structures stored in the local memory of the team's master core. This is possible because an eCORE can access any address in the Epiphany address space. In particular, while an eCORE may access its own scratchpad memory using *local* addresses (which range from 0_{16} to $7FFF_{16}$), its memory can also be globally accessed by all cores using its row and column coordinates: if r and c are the row and the column of a core, the start of its scratchpad memory is at address $r \times 4000000_{16} + c \times 100000_{16}$. The mesh coordinates of the master core are available to all team cores through the DCD area in shared memory.

The eSDK libraries for the Epiphany provide mechanisms for locks and barriers between the eCORES. Their implementation is highly optimized to exploit the fast cMesh subnetwork as much as possible. Because they assume that the synchronized cores belong to the same workgroup, we modified them in order to adhere with our multiple cooperating workgroup organization. Additionally the barrier was augmented with task execution extensions.

Our prototype tasking infrastructure is based on a blocking shared queue stored in the local memory of the master eCORE. The corresponding task data environments are stored in the shared memory.

5.2.4 Measurements

We have conducted a number of tests in order to measure the efficiency of our offloading mechanisms alongside the space and timing performance of the OPENMP runtime within the Epiphany accelerator. Our board is the Parallella-16 SKUA101020 and we use eSDK 5.13.9.10. The systems runs Ubuntu 14.04 with kernel 3.12.0 armv7l GNU/Linux. GCC and e-GCC v.4.8.2 were used as back-end compilers for OMPi.

Table 5.2: Size of empty kernel (bytes)

Scenario	ompi	esdk
1 kernel	7092	2232
16-core team	10560	3084

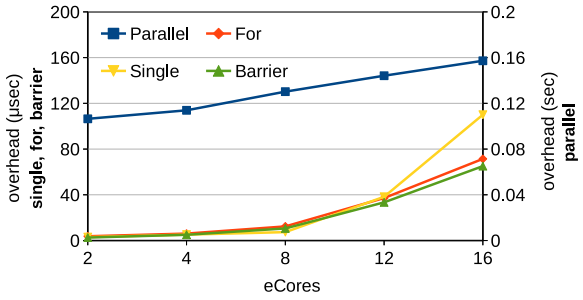


Figure 5.13: Overhead results of EPCC benchmark

Memory Footprint

To examine the memory overhead of our Epiphany runtime, which gets linked with each offloaded kernel, we created a set of simple OpenMP programs. The kernels were compiled with "-O3 -funroll-loops" flags and we used the *e-size* tool of the esdk to examine the produced elf object files. The results are shown in Table 5.2. In the first scenario, one effectively empty kernel is offloaded, containing only a single assignment. It can be seen that ompi incurs a 4.5KiB overhead as compared to an identical kernel created using the native elib. Examining the elf, it is seen that our runtime requires approximately 1KiB more for its internal data and another 3.5KiB for its runtime routines. In the second scenario we create a team of 16 cores running the previous trivial kernel; for ompi this is accomplished through a `parallel` directive while for the esdk program we create a workgroup of 16 cores which are synchronized using a barrier. While the data section remains constant, the additional offloaded runtime routines cause an increase in the text section; approximately 7KiB more than the corresponding native kernel are required. Additional functionality is offloaded if the kernel contains worksharing constructs and this accounts for another 3KiB approximately. All in all, ompi was found to require 4-10KiB more than a similarly structured esdk-based kernel. While this is certainly non-negligible, we note that a) our prototype has not been optimized yet, b) some portions could be moved to shared memory as a tradeoff between local memory space and speed and c) the programmability gains are rather significant.

Overheads

The EPCC micro-benchmarks suite [92] is widely used to measure OpenMP construct overheads for a particular implementation. In order to measure ompi overheads within the Epiphany, we created a modified version of the benchmarks. Their basic routines are offloaded through `target` directives and executed as kernels without further modifications. Measurements are taken from the host side, after subtracting any offloading costs. In Fig. 5.13 we present a sample of the results regarding the overheads of `parallel`, `for`, `single` and `barrier` constructs. The results are quite satisfactory, in all but the `parallel` construct. This is explained in part, because as described in Section 5.2.3, the formation of a dynamic team of cores incurs significant host-device communication, which includes additional kernel offloads. However, it should be stressed that offloading even an empty kernel has an overhead of at least 0.1 sec, needed for resetting the core(s) that will execute it. Eliminating this cost, would require keeping all eCORES active all the time, sacrificing power efficiency.

Mandelbrot Application

We tested ompi using a simple version of the Mandelbrot deep zoom application which calculates a Mandelbrot set and zooms in and out up to $10500\times$ at six predefined points. The whole frame by frame image is written directly to the frame buffer of the Parallella board (with a resolution of 1024×768), resulting in an impressive colourful video. The full traversal generates 204 frames per zoom point. The code for this application is one of the examples included with the eSDK in order to exhibit the real time performance possibilities of the Epiphany chip. Initially a host thread activates all 16 cores to execute the computation kernel. The kernel itself distributes the work statically among the cores; each core calculates the colors for a region of the image and writes the values to the frame buffer. At the end of each frame, all cores inform the host thread and wait to be synchronized. When all cores finish their calculations for the particular frame the host signals them to continue with the next one.

In order to utilize OpenMP, we unified the host and Epiphany code in a single file, moving the kernel code into a `target` region. Next, we removed all calls to eSDK and replaced them with OpenMP pragmas, and finally we removed the synchronization code, since this functionality is now carried out by a `barrier`. The generated kernel size was 11794 bytes; the original kernel was 4728 bytes, in comparison. The execution

Table 5.3: Frames per second for the Mandelbrot deep zoom application (1024x768)

#frames	esdk@Epiphany	ompi@Epiphany	ompi@Zynq
204	17.854	15.829	4.139
408	15.250	13.630	3.469
612	13.411	12.292	3.015
816	12.528	11.632	2.794
1020	13.330	12.304	2.997
1224	14.486	13.234	3.288

results are shown in Table 5.3. We give the total number of frames and the frame rate (i.e. the total number of frames divided by the execution time) for the original application and the OpenMP-based version. For comparison we also provide results of the application when the Zynq is used as the device that executes the kernels. In any given column, the differences between the frame rates is natural because of the variability of pixel calculations (darker pixels incur fewer computations).

As it can be easily seen, the original esdk application performs from 8% to 13% better than the OpenMP-based one. We consider this as a very small difference, given that our prototype is not yet highly optimized. Moreover, the OpenMP version, without any further modifications resulted in a total of 198 program lines, while the original required 301 lines of code. What is more important is that the programmability gains are huge. We achieved on average 90% of the performance of the original application with a mere 5 OpenMP pragmas. Finally, notice that the Epiphany achieves up to $4\times$ more frames per second as compared to the Zynq.

Chapter 6

OpenMP 4 & Support for Multiple Devices

The OpenMP device model specifies that an application can be executed by a set of *devices*. This set includes the *host device* and other *target devices* that may be supported by an implementation. The execution model is host-centric, thus the execution of a program begins on the host device and if one or more target devices are available, particular `target` regions can be offloaded to them. Each device has its own threads, distinct from others that execute on a different device.

The memory of a device data environment is regarded as an autonomous extension of the OpenMP memory model. When an application begins, each device has an initial device data environment. For the host device, this environment corresponds to the data environment of the initial implicit task. During the execution, a hierarchy of device data environments is created with corresponding variables among different devices. The initial values of the device data environments and the data movements are controlled by the `target`-related directives.

Adding device support to an existing OpenMP v3.1 implementation requires a

major reorganization/modification:

(a) Extending the input grammar of the compiler. New functionalities must be developed to parse the new directives, perform new transformations, and produce the intermediate code files. These files include the code executed by the host and code to be offloaded and executed by the target devices (kernels), augmented with calls to corresponding runtime libraries.

(b) Introducing a new subsystem in the runtime libraries of the host device, to handle the attached devices. This controls the discovery, the initialization and finalization of the target devices, furthermore this subsystem serves as an interface between the host and the target devices, by controlling the data transfers and the execution flow.

(c) Providing OpenMP support for each target device. The corresponding libraries should implement the desirable OpenMP functionalities within the device, and also handle communication with the host.

In this chapter we present the architecture of device support in the context of the ompi compiler. We aim at the device agnostic parts of ompi which correspond to items (a) and (b) above. In Section 6.1 we give a detailed description of the OpenMP device directives. In Section 6.2 we discuss the compiler transformations for the device directives and present an efficient solution to the important problem of data environment handling. Finally, in Section 6.3 we show the new modular runtime organization of ompi that supports multiple devices through the use of dynamic linked libraries which is utilized in the following chapters.

6.1 OpenMP 4 Device Support

The key new feature of version 4.0 of the OpenMP API [12] is the introduction of a state of the art, platform-agnostic model for heterogeneous parallel programming. Multiple devices, as for example co-processors, gpgpus or accelerators, can be utilized to reduce the execution time and improve the energy efficiency of an application by utilizing the new device directives. The programmer simply marks portions of the (unified) source code to be offloaded to a particular device; the details of data and code allocations, mappings and movements are orchestrated by the compiler. The

OpenMP device model requires that the accelerators (or target devices) are connected to a host processor which is also considered a device. The program execution follows the host-centric model; it starts executing at the host side until one of the newly introduced constructs is met, which may trigger the creation of data environment and the execution of a specified portion of code on a given device.

In order to transfer data and control flow to a device the `target` directive is used. This directive has an associated structured block representing the code (kernel) to be offloaded and executed directly on the device side. During the execution of the kernel the host task waits until the device finishes and returns back the control. Each `target` directive may contain its own data environment, that is a set of variables accessible in some way by both the host and the device, initialized when the kernel starts and freed when the kernel ends its execution. An `if` clause may appear in this directive, and if its condition evaluates to false then the target region will be executed by the host CPU instead of the chosen device.

Data movements between the host and the devices may be the cause for delays during the launch or the completions of the kernels. In order to avoid repetitive creation and deletion of data environments, the `target data` directive allows the definition of a data environment which persists among successive kernel executions. When an `if` clause is used, and the condition is evaluated to false then the data environment is initialized in the host memory space.

Furthermore, the programmer can use the `target update` directive between successive kernel offloads to selectively update data values that reside in the host and the device data environments.

The `declare target` directive specifies that the associated set of variables and functions are mapped to a device. In essence, the `declare`d variables are allocated in the global scope of the target device, and their lifetime equals the program execution time. The code of the `declare`d functions will be compiled to produce device binaries accessible from the code in the `target` regions.

The execution of an OpenMP program has a set of initial device data environments, that is, a set of variables associated with a given code region, one for each available device. The data environment can be manipulated through `map` clauses in `target data` and `target` directives. These clauses determine how the specified variables are handled within the data environment. When an `alloc` map type is used an uninitialized variable is defined, whereas with a `to` map type the variable is additionally initialized

from the value of the corresponding host variable. If a variable is mapped as `from` then an uninitialized device variable is defined; when the specified directive region finishes, the value of the device variable is copied back to the original host variable. Finally, if no type is specified or the type is `tofrom`, the variable has the characteristics of both the `to` and `from` types.

In order to better exploit the massive parallel architecture of GPGPUs some special directives were also introduced. The `teams` directive creates a given number of thread teams. Each team has a specified number of threads and the master thread of each team executes the code in the teams region. The new `distribute` worksharing construct distributes the iterations of the loops across the master threads of all teams that execute the `teams` region. The combination of `target`, `parallel`, `teams` and `distribute` directives offers an easy to use and powerful tool for accelerating SIMD-based applications.

According to the device model of OpenMP, any program that adheres to v3.1 of the specifications can theoretically form a kernel. This way, the constructs for creating a parallel team, nested parallelism, synchronization between OpenMP threads (locks, barriers), the worksharing constructs (single, for, sections), explicit tasking, etc are allowed within a `target` region. This flexibility transforms the OpenMP to a very powerful parallel programming tool for the accelerator and embedded systems era. Ideally any program, originally written in OpenMP for a shared memory system, can be easily altered to offload some compute intensive code parts to special hardware. Currently, the Intel ICC compiler [44, 45] and GNU C Compiler, GCC (as of the latest version [46]) support offloading directives, with both of them only targeting Intel Xeon Phi as a device. GCC offers a general infrastructure to be tailored and supplemented by device manufacturers. Preliminary support for the OpenMP `target` construct is also available in the ROSE compiler [47]. A discussion about an implementation of OpenMP 4.0 for the LLVM compiler is given by Bertolli et al [48].

6.2 Compiling & Data Environments

6.2.1 Code Transformations

The main transformation step of OMPi for `parallel` and `task` directives is *outlining*. A new function is created, containing the transformed body, and the construct is replaced by a runtime call with the new function and a struct as parameters. The struct contains any variables declared before the construct but used in the body of the construct, and is initialized according to the data-sharing attribute clauses. The same technique is used to transform target constructs. When outlining a `target` construct, we store a copy of the outlined function along with any other outlined functions that may occur during the transformation of its body (e.g. when having a `parallel` directive inside the `target`), in a global list of syntax trees. After the main transformation phase of the code, we use the trees stored in that list to produce kernel files, one for each `target` construct.

The code in Fig. 6.1 is an example that illustrates the transformation details. In the first line of the original code we have the definition of a data environment (DE1) for the device with id 2 that includes the variables `x` and `y` mapped as `to/from`. The second line defines a nested data environment (DE2) for the default device including the variable `y` with a `from` mapping. The last line denotes the code that is to be offloaded to the default device. This code accesses variables `x`, `y` and `k`, forcing an implicit `to/from` mapping for the first and last variable.

When transforming `target data` and `target` directives, a "start" and an "end" call are injected before and after the body of the directive, in order to create the corresponding data environments. In the example of Fig. 6.1 we inject calls to the runtime for the `target data` and `target` directives in lines 2, 3 and 8, 9 respectively. The code in lines 35 and 30 marks the destruction of DE1 and DE2 respectively. For each variable, depending on the map type we inject suitable runtime calls. In particular, an *alloc* or an *init* call is inserted at the start of the body for `alloc/from` or for `to/tofrom` mappings respectively (lines 4, 5, 11, 12). If the variable is `from/tofrom` a finalize call is inserted at the end of the body (lines 27, 28, 33, 34).

The data structure defined in lines 14-18 stores pointers to all variables that form the data environment of the newly created kernel. This struct will be passed as argument to the outlined function (created during the kernel transformation). The pointers are initialized using runtime calls to get the address of the variables on the

Original code:

```
#pragma omp target data map(x,y) device(2)
    #pragma omp target map(from: y)
        x=y=k=1;
```

Transformed code:

```
1 {                             // start target data
2    _devid = 2;                // requested device
3    _ddenv = _start_ddenv(_devid, ..);
4    _initvar(&x, sizeof(x));
5    _initvar(&y, ..);
6
7    {                          // start target
8       _devid = -1;            // default device
9       _ddenv = _start_ddenv(_devid, ..);
10
11      _allocvar(&y, ..);      // ignored if default device is 2
12      _initvar(&x, ..);       // ditto
13
14      struct __dd__ {
15         int (* x);
16         int (* y);
17         int k;
18      } * _devdata = _devdata_alloc(_devid, sizeof(struct __dd__));
19      _devdata->x = get_vaddress(&x, ..); // request address @device
20      _devdata->y = get_vaddress(&y, ..);
21      _devdata->k = k;        // optimized
22
23      ort_offload_kernel(_kernelFunc0_, _devdata, ..); // kernel code
24
25      k = _devdata->k;
26
27      _finvar(&x);
28      _finvar(&y);
29
30      _end_ddenv(_ddenv);
31   }                          // end target
32
33   _finvar(&x);
34   _finvar(&y);
35   _end_ddenv(_ddenv);
36 }                             // end target data
```

Figure 6.1: Compiler transformation example

device space (lines 19, 20). As an optimization, if a variable does not appear in any enclosing `target data` directive, space for the variable is created directly within the

struct, instead of using a pointer (line 21). In our example, the kernel body ($x = y = k = 1$;) has been moved to an outlined function `_kernelFunc0_` the actual execution of the kernel occurs in line 23, where the runtime call is given the function name and the above struct.

6.2.2 Data Environment Handling

The host device needs a bookkeeping mechanism in order to store and retrieve information regarding the variables that constitute the data environments. This information is used when the host accesses these variables for reading or writing, for example during the mapping of a variable (lines 4, 5, 11, 12 of Fig. 6.1), or before/after the execution of a kernel when the `target update` directive is used. This information is also used during the initialization phase of a kernel (lines 19, 20 of Fig. 6.1). Because of the arbitrary nesting of data environments, and the possibility of multiple active devices, the bookkeeping cannot be statically handled at compile time. Of the few compilers that support the OpenMP device model, some handle the data environments incorrectly, or do not face such difficulties since they support only one device. In what follows, we present a general solution to this non-trivial problem.

To allow the host to have fast access to information regarding the variables involved in a data environment, we utilize a special mechanism based on typical separately-chained hash tables (HT). It works approximately as a functional-style compiler symbol table [98], albeit operated by the runtime. A sequence of nested data environments in the source program produces a dynamic sequence of HTs with entries for the mapped variables. The information stored on each entry includes the id of the device which the mapping refers to, and a pointer to the actual storage space of the variable. The hash function takes as input the original variable address combined with the device number and returns the corresponding bucket. Collisions are handled through separate chaining.

More details about this mechanism can be found in our work [14]. Here we present an illustrative example of our mechanism. In Fig. 6.2 we show a code snippet where in line 3 a data environment (DE1) is created with the mapping of variable a. Then a team of two threads is created on the host device and each thread defines a separate data environment (DE2$^{(1)}$ and DE2$^{(2)}$) which includes the variables a and b. Finally, each thread offloads a code block, while at the same time creates a new data environment

```
1  int a, b;
2
3  #pragma omp target data map(a)          // DE1
4    #pragma omp parallel num_threads(2)
5    {
6      int c;
7      #pragma omp target data map(b)      // DE2^(1), DE2^(2)
8        #pragma omp target map(a, c)      // DE3^(1), DE3^(2)
9          ....
10   }
```

Figure 6.2: Nested device data environments created by a team of threads

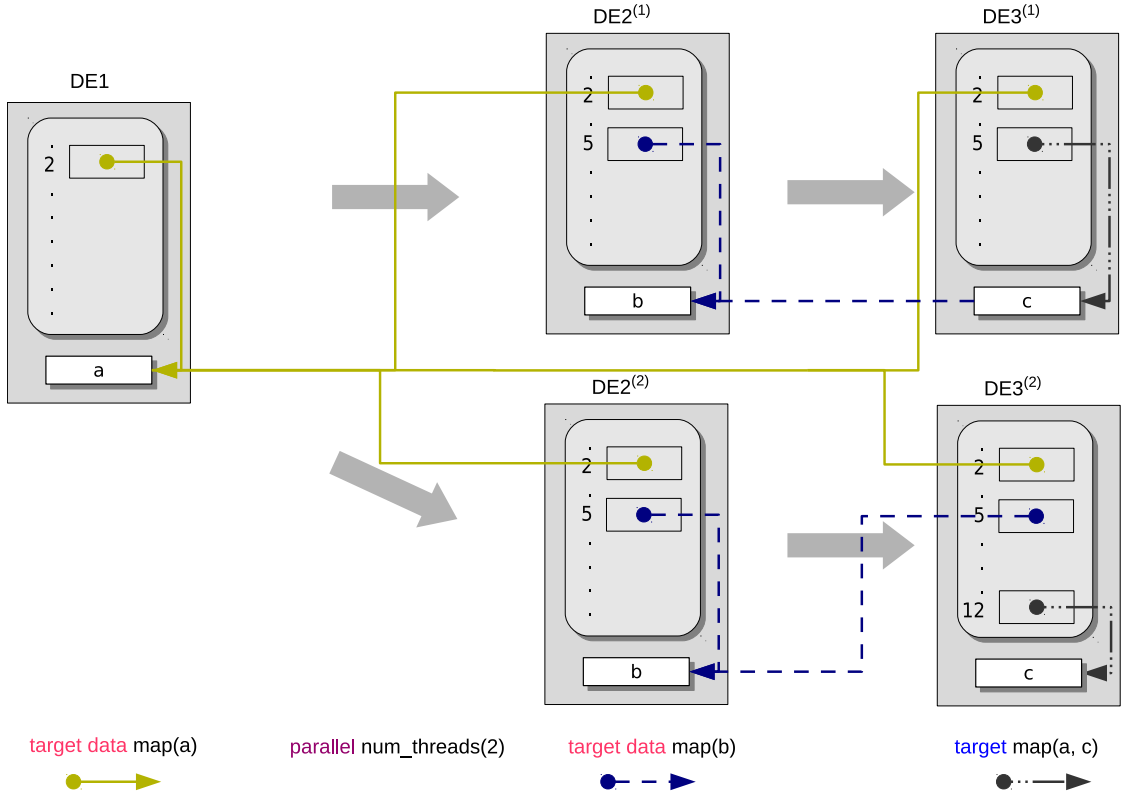


Figure 6.3: Hash table sequence for code in Fig.6.2. Solid yellow, dashed blue and mixed black arrows denote allocations and definitions made at lines 3, 7 and 8, respectively

(DE3$^{(1)}$ and DE3$^{(2)}$) which includes the variables a and c. All the mappings in the example refer to the default device.

The sequence of the HTS for the above example is given in Fig. 6.3. The result of line 3 is the creation of DE1 which stores only one entry, that for variable a. The

creation of the thread team does not affect the bookkeeping mechanism, and both threads can access the data of DE1 without the need of locks. Line 7 defines $\text{DE2}^{(1)}$ and $\text{DE2}^{(2)}$ (one for each thread); this causes the creation of new HTs, both of which are created as copies of the previous HT. As a result, each thread retains access to the enclosing data environment. The mapping of variable b adds a new entry for $\text{DE2}^{(1)}$ and $\text{DE2}^{(2)}$ and the addition of a pointer from the HT to this entry. Similarly, in line 8 we have the creation of new HTs that handle $\text{DE3}^{(1)}$ and $\text{DE3}^{(2)}$, respectively, as copies of the previous two HTs. In the case of the second thread, variable c is hashed onto bucket 12, which is a free bucket. In contrast, variable c of the first thread caused a collision (hashed onto bucket 5), resulting in a chain between the variables c and b.

The data handling mechanism is operated by the host, resides in the host address space, and is independent from any attached devices. The HTs allow for efficient variable insertion and look-up operations. Each time a nested data environment is created, a new HT is initialized as a copy of the HT used by the enclosing data environment, as in functional-style symbol tables; destruction of the data environment requires a single memory deallocation for the corresponding HT.

6.3 A Modular Runtime Architecture for Multi-Device Support

The runtime architecture we envisage is able to drive multiple and disparate devices. To enable this flexibility, the existing runtime system must be equipped with a generic device management capabilities so as to be able to interface with any device. Supporting a particular device would then require a device-specific *module* to interface with the device management subsystem (used by the host) and an autonomous runtime library (*devrt*) to implement OpenMP functionalities for offloaded kernels (living at the device). The general organization is shown in Fig. 6.4.

The host runtime system (see Fig. 1.2) has been extended with a device management subsystem whose role is to:

* Collect information about the available target devices.
* Bind the devices to the host runtime.
* Make all needed initialization steps to prepare the devices for kernel execution.

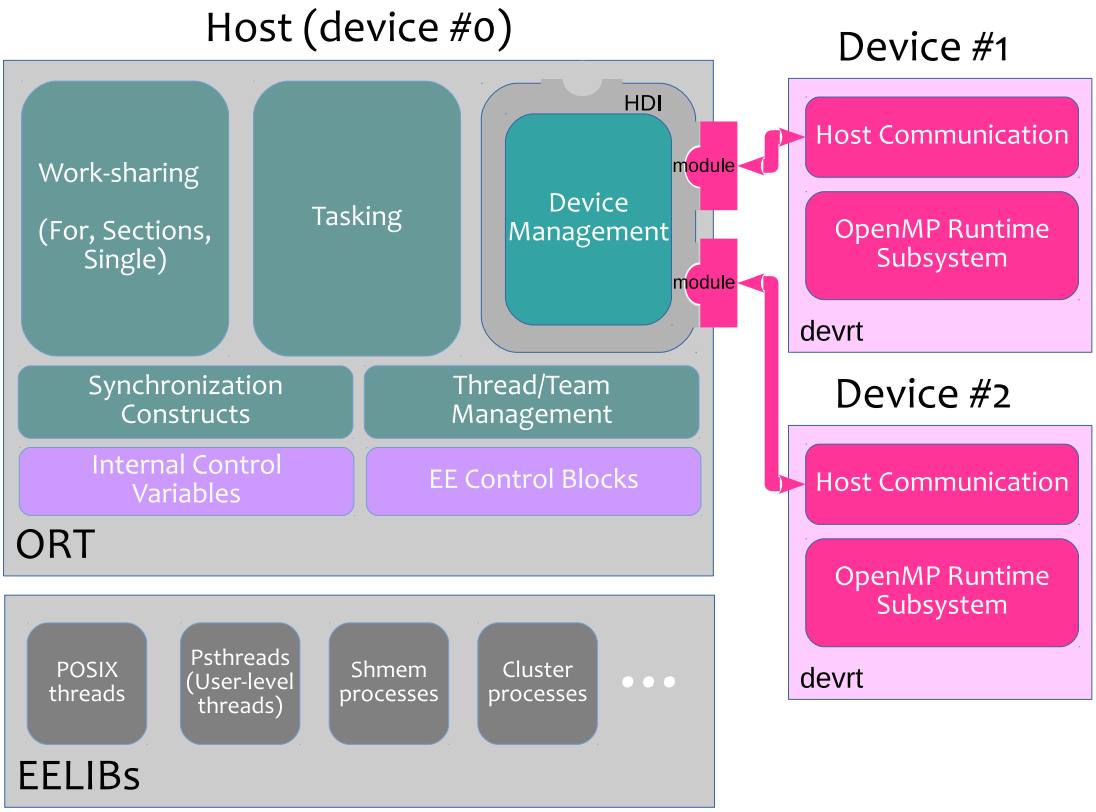


Figure 6.4: OMPi's architecture for device support

* Support the creation and destruction of device data environments and transfer of data from/to the devices.

* Transfer execution flow to the target devices.

Since multiple threads may exist on the host side (simultaneously accessing the target devices), the functionalities described above must be designed to be thread-safe wherever this is needed. Furthermore, the device management subsystem is *device-agnostic*, meaning that it is designed to support any possible accelerator attached to the host. To achieve that, we propose a general *host-device interface* (HDI) which acts as a bridge between the host and target devices. The full HDI calls can be found in Appendix A and they include calls for:

* controlling the device (initialization and finalization)

* getting information about the device e.g. capabilities

* transferring data between the host and device

* triggering the execution of a kernel

Supporting a particular device involves the implementation of two distinct components: one that is executed by the host and one that is executed by the device. The first component is the module (see Fig. 6.4) and snaps in the device management subsystem in order to allow the host to establish communication with the device, and implements the HDI. The HDI serves as two-side provider:

a) It provides the module access to the synchronization primitives (the host's EELIB layer). This is needed because the functions of the module need to be thread-safe.

b) It provides the device management module access to the target device (memory & thread manipulation).

Because devices may be removed or attached to a system over time, interfacing them with the OMPi runtime system cannot be static. As a result, all modules are implemented as dynamic libraries. They are loaded at execution startup and provide the application access to the available devices.

The second component, is the devrt (see Fig. 6.4) and it is the device-side one. It largely implements two functionalities:

a) communication with the module and interacting with the host

b) support of the kernel execution on the device. According to the OpenMP specifications, a kernel can contain any valid OpenMP code. Consequently, the devrt could actually provide full OpenMP facilities within the device. Thus it represents a device specific OpenMP implementation. This is either resident on the device, or linked and offloaded with every kernel.

We have included the above functionality in the runtime system of the OMPi compiler. We have implemented a number of virtual devices mostly for testing and debugging purposes. Furthermore, we have a full implementation that supports the OpenMP device model for the Parallella board, which includes the Epiphany accelerator (as described in Section 5.2). Finally, we have preliminary support for OpenCL-capable devices. In particular we have developed a device runtime on top of OpenCL v2.0 libraries, that can be used to offload execution to a compatible GPGPU.

Chapter 7

A Compiler-Assisted Runtime

7.1 Supporting OpenMP on the Device Side

The OpenMP device model offers a great deal of flexibility regarding the constructs allowed within kernel codes. This in effect requires that a complete OpenMP rts be present to support kernel execution. However, OpenMP was originally designed for shared memory multiprocessors. These machines include a large amount of shared memory supported by sophisticated cache coherent protocols, high bandwidth interconnections and usually offer a large set of hardware-assisted synchronization primitives (e.g. compare-and-swap, fetch-and-add, memory barriers). Moreover, these systems are equipped with an operating system accompanied with optimized low-level software libraries such as posix threads, for manipulating the execution units of the system.

On the other hand, embedded or attached accelerators have different architectures and are designed to serve different purposes. For example, the organization of some accelerators is targeted to the efficient execution of streaming applications;

GPGPUs are better suited to speed up matrix-based computations; e.g. co-processors are synonymous to hardware diversity, since each manufacturer equips a product with specialized hardware modules and target a specific class of applications.

With some notable exceptions such as the Xeon Phi accelerator[45], a common characteristic of the various types of co-processors is that they offer a limited amount of resources. Hence, the challenges posed when implementing an OpenMP RTS for such devices depend on these resource limitations. The absence of a POSIX-like interface for manipulating threads may add design difficulties or considerable offloading costs regarding dynamic or nested parallelism. For example, most of the current native development tools for GPGPUs do not support nested parallelism. Lack of hardware synchronization primitives would add overheads in the cooperation of the accelerator cores, since software implementations of locks or barriers result to considerable delays. Arguably, one of the most important limitations is the size of the available memory; small private or shared memories at the co-processor cores impose restrictions regarding the kernel executable size and/or the actual application data. This is particularly pronounced in the absence of a fast global memory; the kernel code has to include the OpenMP RTS, further limiting the available memory space. The Epiphany accelerator used in the Parallella [5] is an example of an embedded accelerator with severely limited memory resources; each core is equipped with just 32KiB of fast local memory. While it can also access a larger 32MiB memory shared with the host processor its access times are almost an order of magnitude larger.

There are two approaches for supporting OpenMP on a device with limited resources:

Partial support Partial support of the constructs is a pragmatic solution that works in real world applications [49, 47, 58]. For example, there is no point in trying to implement an optimized tasking infrastructure for a GPGPU with limited local memory which lacks fine grain synchronization primitives. Instead, a careful implementation of a combined construct such as `target teams distribute parallel for` is a desirable feature that exposes the computational power of this kind of hardware. Of course, partial support minimizes the expressiveness of the programming environment. The application code may have to be redesigned to match the availability of OpenMP constructs, a fact that also reduces code portability and re-usability.

Full support Some works in the bibliography [44, 46] choose to support OpenMP fully on the device side. This strategy provides a powerful tool for developing parallel applications based on a high level hardware abstraction. Nevertheless, the design of a complete OpenMP RTS is not a trivial task. Furthermore, the hardware limitations may lead to poor performance for some of the OpenMP constructs [13, 48, 47].

In this work we are the first to propose a general methodology which can be utilized to offer flexible and adaptive OpenMP RTS. The goal is the development of an RTS architecture which implements only the OpenMP features required by each particular application. That is, it results in an application-specific RTS configuration. This is possible because of a key observation: all kernel code must lie within a single source file. This enables a compiler to analyse the behaviour of the kernel with respect to OpenMP constructs, through detailed interprocedular analysis. Thus, it can decide exactly what constructs are used, their nesting levels, the types of employed loop schedules, etc.

The proposed system is shown in Fig. 7.1. The compiler is responsible for analyzing and transforming the code. It takes as input an OpenMP program with `target`-related constructs. The output is a set of files; the main one is to be executed on the host and the other files represent the kernels to be executed on the devices. Along with each kernel, a set of *metrics* which are gathered during its analysis are output. The metrics are passed to the *mapper*. The latter is responsible for choosing the most efficient runtime configuration for the given metrics. In order to do this, the mapper either selects one from a precompiled set of libraries or parametrizes appropriately one of them and builds it on the fly.

7.2 Analyzing a Kernel

The motivation behind our proposal stems from the observation that providing comprehensive OpenMP support for an attached device can be quite demanding both in terms of memory requirements and execution overheads, especially in devices with limited resources (e.g. embedded MPSoCs). Each kernel should be accompanied by a rather sizable runtime library in order to enjoy OpenMP support. However, most applications (typical kernels included) rarely need all of the OpenMP facilities. What

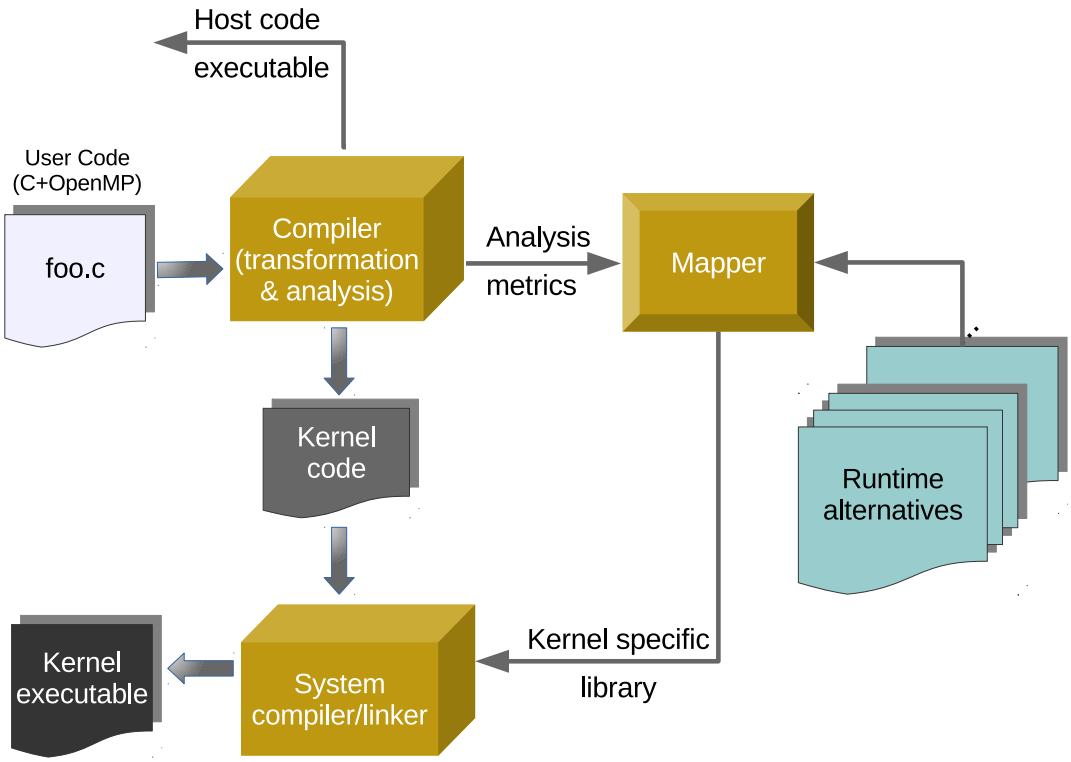


Figure 7.1: Overview of compiler-assisted RTS

if the compiler can decide on the subset of OpenMP functionality that is necessary and just utilize a custom, kernel-specific runtime library to accompany it? The potential savings could be quite significant.

An OpenMP kernel is defined by a block of code enclosed lexically within a `target` construct. The actual kernel *region* includes any code in called routines. Such routines are defined within `declare target` constructs and are in fact offloaded with the kernel. The compiler has thus access to the whole kernel region and can employ inter-procedural analysis in order to analyse the entire dynamic extend of the kernel.

The compiler can build the call graph of each kernel and visit each of the called routines. Our thesis it that the compiler can then extract information about the employed OpenMP constructs (if any), and thus determine the actual OpenMP functionality that is necessary for the execution of each particular kernel. More often than not, a given kernel will not require the entire OpenMP functionality but a rather small portion of it. Given this information, the offloaded kernel can be accompanied by a suitable subset of the OpenMP runtime library, potentially decreasing the total

offloaded footprint.

Here is a number of important conclusions, among others, that can be derived from the above code analysis:

* *OpenMP in kernel*: Decide whether OpenMP functionality is required at all. If no OpenMP directives are utilized and no OpenMP runtime functions are called, then there is no need to include OpenMP support.

* *Dynamic parallelism*: Determine whether the kernel spawns parallelism, through `parallel` directives, and if possible, their nesting levels and/or the total number of threads employed. The absence of parallelism can make the required runtime support rather minimal. Knowledge about the number of threads and the nesting levels can also tailor the corresponding runtime data structures to exact sizes.

* *Worksharing regions*: A major portion of an OpenMP runtime library is devoted to the handling of worksharing regions. Knowledge about the exact types of worksharing constructs utilized by a kernel (`for`, `sections`, `single`) can slim down the necessary runtime support.

* *Explicit tasking*: Discover the presence of user-defined `tasks`. If none is observed, the tasking subsystem of the runtime library is not needed at all. Supporting tasks is one of the most sophisticated assets of an OpenMP RTS, with significant overheads and memory requirements.

* *Internal control variables*: Decide whether the code makes use of OpenMP Internal Control Variables (ICVs), either by setting them or getting their values. Furthermore, it can be determined exactly which ICVs are being utilized. Storing and maintaining ICV values represents a major issue in an OpenMP support library. If, for example, ICV values are used only for retrieving information then a single copy of them (instead of repeating them in every task structure) is adequate for the execution of the whole kernel.

The above proposal can be extended to the general case of host OpenMP programs, not just kernels. The only obvious requirement, is that the application code must not refer to external routines, so that the compiler is in a position to perform full interprocedural analysis and derive the above conclusions. In case where the program

depends on external routines, the analysis response will declare inability to provide valid conclusions or metrics. Otherwise, the conclusions and a set of related metrics will be output to optimize the RTS used for the specific application.

7.3 Mapper: Utilizing Compiler Metrics

The set of metrics generated by the compiler are passed to the mapper module which is responsible for choosing the most appropriate runtime "flavor". In the case where the kernel does not make use of OpenMP directives, the RTS should only include basic features for enabling a single co-processor core to execute the serial code of the kernel. Thus, this RTS should include mainly host-specific functionalities, and will result to a minimal footprint library regarding the device side. The reduced capabilities of the RTS include the co-processor initialization and finalization phases, as well as the code and data offloading. This minimal RTS may prove quite useful in systems where the parallelization capabilities can not be abstracted as a team of independent threads. Furthermore, there might be cases where dynamic creation of a parallel team on the device side is hard or sometimes impossible to implement. A workaround for this scenario is the utilization of a team of threads executing on the host side, that concurrently offload kernels (containing serial code) to a multicore co-processor.

The usual case, nevertheless, is where the kernel includes directives for creating a parallel team of threads that cooperatively execute a code block. The RTS library that is to be linked with the kernel code consists of some RTS-specific data along with the code implementing the required functionalities. In more detail the internal data are related to:

* the execution entities, represented mainly by some kind of thread abstractions and

* the implicit (or explicit) tasks executed by the threads

The smaller total footprint for the RTS library the more beneficial would be in the cases where the cores of a co-processor are equipped with small amount of local memory.

Additionally, the presence of extra functionalities within the library not needed by the kernel code, may lead to non-optimized execution times. A representative example of such functionality are the tasking extensions of the OpenMP barrier. Upon

encountering a barrier construct, a thread has to wait until all its siblings reach the barrier and until all pending tasks of its team are executed. The implementation of former condition results to far less overheads when compared with the task executing part. Typically threads can be notified about the entrance of their siblings in the barrier through the use of a counter or a matrix of flags. Ensuring that all pending tasks are executed though, involves repeated snooping to shared queues and/or counters, adding substantial amount of overheads. Thus, avoiding the unnecessary functionalities may result to execution speed-up.

Implementing a general, full fledged RTS which is capable of offering OpenMP support is a typical approach in the bibliography. What we propose here is to utilize custom, application-driven RTSs, that only supply the functionality required by the particular application. In Fig. 7.1 the mapper is the module responsible for this: based on the application characteristics as depicted by the compiler-generated metrics, it optimizes the RTS by tuning its internal data and functionalities to best fit for the particular application.

Possible realizations of specialized runtime libraries include:

* A fixed set *pre-compiled* libraries. The set of libraries is selected to target specific classes of applications, as derived from typical use-case scenarios. For example, there can exist a library that does not provide tasking support. Another possibility would be a trimmed down library that only supports a selected worksharing construct (e.g. `for` loops). The mapper then undertakes the task of mapping the provided kernel metrics to the set of available libraries; the most appropriate one should be selected so as to minimize the offered OpenMP functionality while at same time covering all kernel requirements.

* A set of on-the-fly *parameterizable* libraries. Because not all applications can benefit from the default values of the runtime parameters, the mapper can choose to tune some parameters according to kernel characteristics and build different library flavors. For example, if the team sizes are known, the barrier data structures can be tuned to service the specific number of threads. Of course, parametrization requires recompiling and thus the custom libraries are built at the compile-time of the application.

The mapper combines the metrics with all possible library configurations to provide the optimized library. The larger the number of the pre-compiled libraries/-

parametrized RTSs, the better the decisions taken by the mapper for the result libraries that will support the application kernels.

7.4 Implementation in the OMPi Compiler

7.4.1 Kernel Analysis

The analysis of the kernels is done at a high level. The whole program is represented by an abstract syntax tree. Upon encountering an OpenMP `target` node, the compiler analyses its body and follows the chain of routine calls (if any) in order to discover the OpenMP functionality required by this particular kernel. To avoid visiting a routine multiple times (since it may be called by multiple kernels), all routines defined within `declare target` regions are analysed before any other program transformations. The compiler constructs the call graph and traverses it; for each visited function f, the following are some of the metrics currently gathered:

* The total number of OpenMP constructs
* The number of first-level (non-nested) `parallel` constructs ($N_p^{(f)}$).
* The number of `for` loop ($N_l^{(f)}$), `sections` ($N_s^{(f)}$) and `single` ($N_i^{(f)}$) constructs; a counter for the number of constructs with `nowait` clauses is also maintained ($N_{nw}^{(f)}$).
* The number of `task` constructs ($N_t^{(f)}$).
* The number of explicit `barrier` directives ($N_b^{(f)}$).
* The maximum level of parallelism ($L_p^{(f)}$).

All the metrics except the last one count the constructs encountered in the function itself. The parallelism nesting level is determined from the function and all the functions called by it as follows: If a function g is called by f at nesting level $l_{f\to g}$, then the nested parallelism level for this particular call is given by $l_{f\to g} + L_p^{(g)}$. The maximum parallelism level observed for function f is given by:

$$L_p^{(f)} = \max_{g \text{ called by } f} \left\{ l_{f\to g} + L_p^{(g)} \right\}.$$

Consequently, if for example $L_p^{(f)} = 1$, there may be no need to add support for nested parallelism to a kernel that calls function f. If the compiler detects recursion, this particular metric is disabled.

To maximize performance, ompi allows overlapping worksharing regions whereby each thread of a team may proceed independently to a following worksharing region, as long as the previous one contains a `nowait` clause. The mechanism is quite complex [99, 100] and requires handling of sizable data structures. If $N_{nw}^{(f)} = 0$, there is no need to implement it; a simple blocking barrier would be enough to support all worksharing regions. On the other hand, if for all functions f called by a kernel, $N_{nw}^{(f)} = N_l^{(f)} + N_s^{(f)} + N_i^{(f)}$, and $N_b^{(f)} = 0$, there may not be a need to implement a barrier mechanism at all.

The gathered metrics are used at every encounter of a `target` tree node during code transformations. Before actually transforming the construct, its body is analysed in a similar way as above, and the metrics are combined with the precomputed ones for every function called from the kernel. The final set of metrics are stored in a table and the compiler proceeds to the transformation of the kernel body. During code generation, the computed metrics for each `target` construct are embedded into the corresponding kernel file as C language comments, for communicating them to the mapper.

7.4.2 A Concrete Target: The Epiphany Accelerator

This original rts (Fig. 7.2) was used as a basis for the design of a set of adjustable rtss, each one specialized for a certain type of kernels. For the rest of the text we will refer to the original rts as the *Full* rts. The architecture of this rts is similar to the one used to provide OpenMP support for the host device. The upper layer includes all functionalities regarding the execution of worksharing constructs, the tasking infrastructure and all the internal bookkeeping data for the execution of an application. The lower layer is responsible for manipulating the threads (dynamic parallelism is possible through communication with the host) along with some synchronization primitives and asynchronous memory transfers.

The rts is built as a Linux static library, and is linked with each offloaded kernel. The rts is organized as a collection of a largely independent routines so that the system linker can attach only the necessary routines with each kernel. However, the

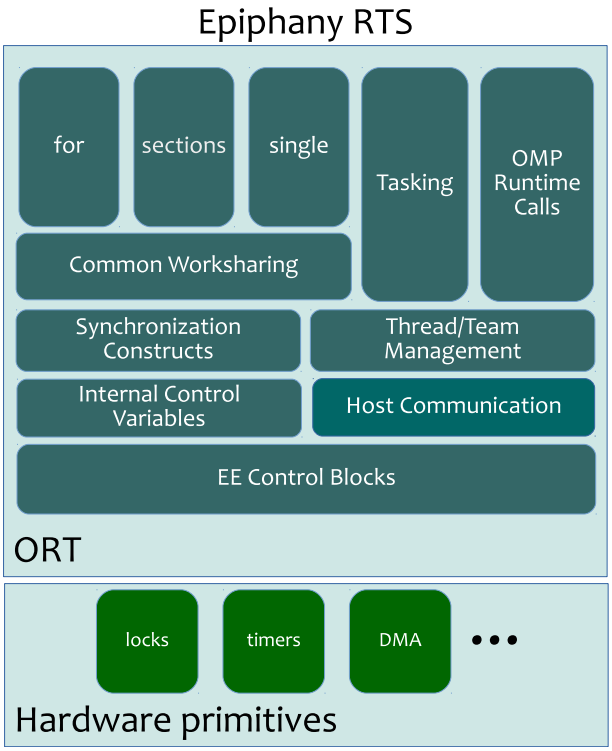


Figure 7.2: The runtime system architecture for OpenMP support in the Epiphany

complex relations between the internal data structures and the runtime routines force the linker to include sizeable portions of the library. As a result, the *Full* RTS has a relatively large footprint, even when it accompanies an effectively empty kernel [13]. Furthermore, because dynamic memory allocation is not supported at the eCORE level, the RTS must reserve enough local memory space to cover the worst case. As a result, the actual local memory left for pure application data is well below the 32 KiB available.

Our strategy for implementing the proposed mechanism was to create different library flavors, aiming to minimize the library footprint. In particular, based on detailed analysis of the runtime organization, we identified three parts that contribute the most both because of the size of the involved routines and the size of the required data structures:

1. Dynamic parallelism: A substantial amount of data structures and routines are needed in order to support dynamic parallelism within a kernel. In particular:

   (a) The RTS stores bookkeeping information regarding the status of the execution entities (run-to-completion threads executed by the eCORES).

   (b) The RTS provides the mechanisms for an eCORE to form and deform a paral-

lel team by communicating with the host. This mechanism occupies space in the local memory of the eCORES and also in the shared memory between the host and the Epiphany. The support of nested parallelism results to larger footprint for the runtime; memory consumption is analogous to the supported levels.

(c) In order to coordinate the execution of an OpenMP team of threads the RTS utilizes internal locks. Furthermore, to synchronize the threads it provides a barrier mechanism. For performance reasons, the data and routines of these functionalities are located in the local memories of eCORES, thus reducing the memory availability for computation data.

All this infrastructure can be discarded if the kernel does not contain a `parallel` directive. Hence we developed an RTS variant which does not support dynamic parallelism creation. Although the original design can support arbitrary levels of nested parallelism, there is no practical use for more than two levels of parallelism on 16 cores. Consequently, we designed two versions of the RTS, one supporting exactly one level of parallelism and another supporting two levels.

2. Worksharing: The OpenMP worksharing constructs (`single`, `for`, `sections`) may have different combinations of `reduction`, `schedule` (with the various schedule types), `collapse`, `ordered` and `nowait` clauses. Supporting all of them requires data structures with large memory footprint. In practice, typical applications do not utilize all possible variations. As a result, we developed a set of RTSs that support specific combinations of the above constructs and clauses.

3. Tasking: The tasking infrastructure for the Epiphany is the module with the largest memory requirements. The required functionalities include fine grain synchronization so most of the runtime data must be stored in local memories; in particular they are stored in the local memory of the team's master eCORE. This means that the local memory of one eCORE stores the tasking data of all eCORES. Because all eCORES are candidates for team masters, preallocated tasking structures must be present in the local memories of all eCORES. Furthermore, barrier synchronization is charged with task execution duties which impact overall performance. To optimize the support for applications which do not utilize tasks, we developed RTS flavors with no tasking subsystem. In addition,

these flavors implement lighter/faster versions of the barrier mechanism.

7.5 Evaluation

7.5.1 Experimental Environment

To evaluate our proposed method we used the Parallella-16 SKUA101020 board. The board is equipped with two processing modules; the main (host) dual-core ARM processor and the Epiphany-16 co-processor. For the results presented here we used ESDK 5.13.9.10.

7.5.2 A Detailed Breakdown of the RTS of OMPi

The original RTS support for the Epiphany [13] was designed to provide full OpenMP support, under the constraint of the limited memory resources. The first step towards designing a set of adjustable RTSs was to analyse the original runtime and understand the impact each component has. The purpose of this procedure was to discover in detail byte sizes of all different data structures and the corresponding functionalities they support. The results of this analysis guided the design of distinct RTSs, specialized to different kernel scenarios.

In Table 7.1 we present the sizes of the most important runtime data structures. Notice that these represent only the eCORE-resident parts; additional data structures are kept in the (slower) shared memory and are of no interest here. The RTS of OMPi utilizes two fundamental descriptors: the thread and the task descriptor. The former is named EECB (execution entity control block) and holds all the information needed by an OpenMP thread to execute a code region and to coordinate with sibling or child threads. The later holds the data required for the execution of a specified task, either implicit or explicit. As seen in Table 7.1, the sizes of these entities have the biggest impact on the total footprint of the RTS.

Not all bookkeeping data are actually needed for the execution of every kernel. The idea is to trim down these structures to save local memory, while at the same time satisfy the real needs of a kernel. The essential data require 48 bytes per EECB while the data for worksharing constructs are 80 bytes. A closer look at each worksharing construct reveals that the loop construct occupies almost half of the space.

Table 7.1: Data sizes in the original RTS

Data structures	**Size in bytes**
EECB data	1440 (1 active region)
Essential	48
Worksharing	≥ 80
No-wait regions	$8 + 64 \times$ (# active regions)
Loops	32
Static loops	4
Ordered	28
Sections	4
Tasking data	1312
Task descriptor	72
ICV data	32 per task
Reductions	16 per ECORE
Critical	16 per ECORE
User defined locks	176 per ECORE
Nested parallelism	88(+1088 in SM) per level

A performance-oriented but memory-consuming feature of the original runtime, is the ability to allow multiple active worksharing regions, whereby each thread of a team may proceed independently to a following worksharing region, as long as the previous one contains a `nowait` clause. If up to n overlapping regions are supported, an additional $n \times 64$ bytes per EECB are necessary.

The space needed for a task descriptor is 72 bytes. In the current RTS all the task-related structures of all ECORES must be stored in the EECB data structure of the master ECORE. Because all ECORES are eligible as team masters, the same space must be provided in all ECORES. This results in a total 1312 bytes per EECB for the whole tasking mechanism. This demonstrates the potential for reducing the memory footprint in the case where the an application does not use explicit tasking. The size of the necessary ICVS is measured to be 32 bytes per task. If the kernel does not modify any of their values (e.g. there are no calls to `omp_set_xxx()` routines), then it could be possible for the RTS to use only one copy of the ICVS for all tasks. In such a case, up to 12256 bytes could be saved in total (in the local memories of all 16 ECORES).

Synchronization between the eCORES is relatively cheap, since 16 bytes per core are needed for the `reduction` and `critical` constructs. Due to lack of dynamic memory allocation, the original runtime pre-allocates space for 8 user-defined locks. This mechanism requires 176 bytes in the local memory of each eCORE, even if a kernel uses no locks at all. Finally, considering parallelism levels, the data needed for each supported nesting level occupies a significant amount of memory. Each additional level needs 88 extra bytes in the local memory (plus more than 1KiB on the shared memory).

We should make two important observations at this point. First, except for the data structures, there is the corresponding code that handles them, so removing unnecessary data structures has the beneficial side-effect of decreasing the size of the library code. Second, slimmer code usually means faster code. Although in this section we concentrated on minimizing the library sizes, we also expect to have some performance gains for free. Additional performance gains are possible by redesigning the employed algorithms. For example, if the tasking subsystem is removed from the equation, a significantly faster barrier implementation is possible, which avoids polling for tasks to execute.

7.5.3 Implementation of Runtime Flavours

Based on the above analysis of the original RTS, we redesigned and implemented 12 different runtime flavors. Each flavor is a modified version of the original, trimmed to support a limited number of constructs. For each flavor we removed the unnecessary internal data structures and modified all routines respectively. The set of the different RTSs are as follows:

(1) *NoOMP*. This RTS does not support any OPENMP directives within the kernel. Each eCORE can execute only sequential code.

(2) *ParallelOnly*. This RTS provides the mechanism for an eCORE to form and deform a parallel team. No other OPENMP functionality is supported.

(3) *ParReduction*. This RTS is an extension of the previous one, which implements the `reduction` clause on a list of variables.

(4) *ParCritical*. This RTS extends (2) and allows only the `critical` synchronization construct between the eCORES of a parallel team.

(5) *ForStatic*. This is the *ParallelOnly* RTS where the team members can also utilize the `for` worksharing construct. Only the `static` schedule is supported. No other worksharing constructs are offered.

(6) *ForOrdered*. This alternative extends the previous one by adding the ability to utilize the `ordered` clause of the `for` directive.

(7) *SingleOnly*. Here we extend the *ParallelOnly* flavor by supporting only the `single` worksharing construct.

(8) *NoTasks*. We developed this RTS to optimize the support of kernels with no explicit tasks. The rest of the OPENMP functionality (e.g. workharing, synchronization, etc) is present.

(9) *BlockingOnly*. This is an almost complete OPENMP RTS but the support for `nowait` worksharing regions has been disabled in order to reduce the footprint of the related RTS structures.

(10) *NoTasksBO*. We added a variation of the *BlockingOnly* flavor where the tasking support has been removed.

(11) *SingleTasks*. This RTS provides support for creating teams of ECORES that can create explicit tasks and can workshare only through the `single` construct. This is based on a common pattern where one thread generates tasks and the others consume them.

(12) *Full*. This is the original RTS.

The above set does not cover all possible use cases, i.e. does not include all possible combinations of OPENMP constructs. Instead it was guided by common sense for supporting usual application scenarios. Anyway, our goal is to prove the potential of the proposed mechanism, and not to derive all possible runtime flavors targeted for all possible kernels.

Furthermore, all RTS routines were carefully trimmed to implement only the required functionality. Barriers constitute an important example. In a complete OPENMP runtime system a barrier has to synchronize team threads and also act as a task scheduling point. In all flavors but *BlockingOnly*, *SingleTasks* and *Full* there is no tasking support and consequently barriers were simplified to handle only thread synchronization.

The RTSs are parametrized so as to offer better adaption to specific kernels. This adjustment occurs during the library compilation time, based on the metrics produced by the compiler. Specifically, in all flavors but *NoOMP* we parametrize the number of parallel nesting levels that the library supports. For the *NoTasks* and *Full* RTSs there is an additional parameter that sets the number of maximum active `nowait` worksharing regions. Furthermore for the RTSs (8), (9), (10) and (12) we can limit the number of user-defined locks that the library can support.

Choosing the Runtime Module

The mapper imports the set of metrics provided by the compiler and uses them in order to choose the most appropriate RTS module to be linked with a kernel. In our implementation the mapper is designed to work with the specific device (Epiphany). This means that the mapper is aware of the characteristics of the 12 RTS flavors described in the previous section and maps the compiler metrics onto them. A different approach would be to utilize a configurable, device-agnostic mapper. Such a mapper can adapt its decision process according to the capabilities of each device. This is achievable through additional information taken from a meta-data file that describes the features of the available RTSs. The implementation of such a universal mapper would offer the advantage of providing a single mechanism that can deal with different types of devices and runtime flavors, but it is beyond the scope of this work.

The operation of our mapper can be summarized in two steps:

* During the first step, it reads the metrics generated by the compiler and decides the actual RTS flavor to be used.

* In the second step, it parametrizes (if needed) the chosen RTS and compiles its sources to provide the final binary of the library.

An overview of the decision making mechanism of the first step is presented in Fig 7.3. RTS (1) is chosen to accompany kernels which do not include OpenMP constructs. Based on the tasking metrics, RTSs (9), (11) or (12) are used when tasks are present; the actual choice depends on the type of worksharing regions observed. If no explicit tasks are used RTSs (2)-(8) and (10) are candidates. The decision is driven by the presence of `parallel`, `reduction` and other worksharing constructs and clauses.

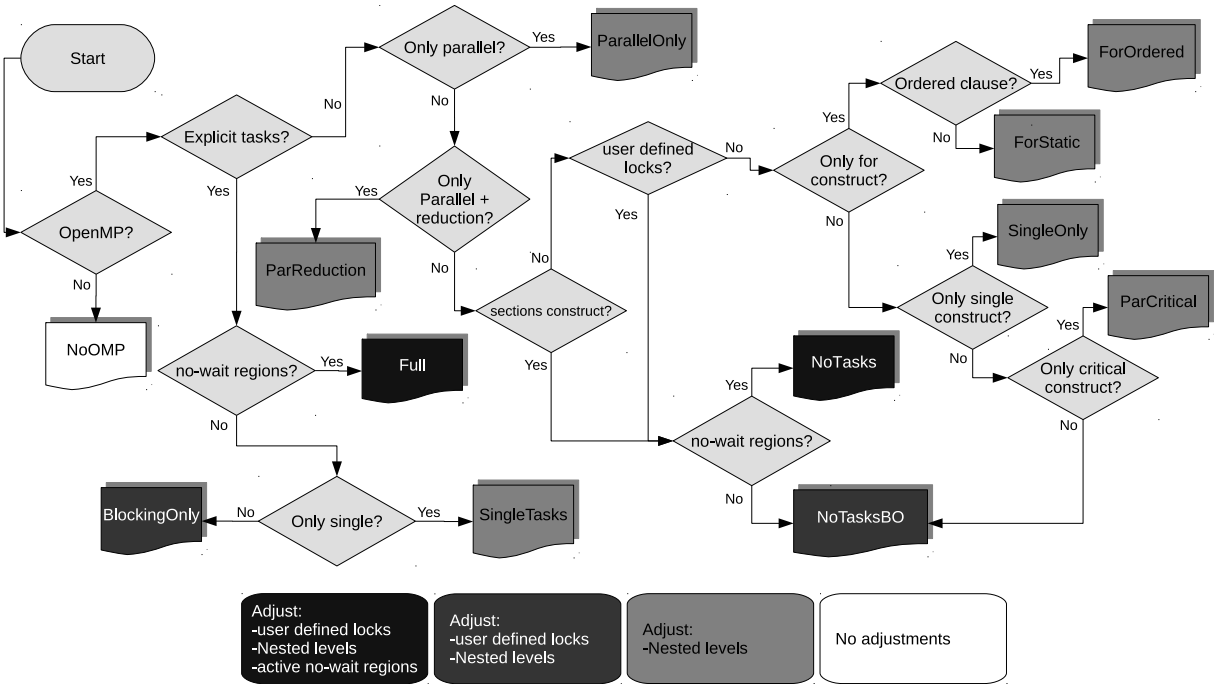


Figure 7.3: Mapper decision flow

7.5.4 Results

For our experiments we use as a reference the *Full* RTS (12) with the default parameters, and compare it with the optimized RTSs resulting from the combination of the kernel analysis and the mapper selection. The kernels were compiled with "-O3 -funroll-loops" flags and we used the `e-size` tool of the ESDK to examine the produced ELF object files.

The first set of tests included a modified version of the EPCC microbenchmark suite [92] where their basic routines are offloaded through `target` directives. These benchmarks are intended for measuring the overheads of specific constructs; in addition we utilized them to exhibit possible size benefits for the produced kernels. From the whole set, we selected the benchmarks related to `for` with `static` schedule, `critical`, `single` and `for` with the `ordered` clause.

Next, we implemented three simple applications. The first one is the scenario of a kernel which does not include any OpenMP functionality at all. In practice, this is an empty kernel containing only one assignment instruction. The second one is the iterative computation of $\pi = 3.14159$, based on the trapezoid rule with $2,000,000$ intervals, and using an OpenMP kernel which spawns a parallel team of 16 threads. The third application is a modified version of the NQueens task benchmark, taken

Table 7.2: Elf sizes (bytes)

Application	Full RTS	Optimized RTS	Reduction
Empty kernel	8228	2252	72.63%
Mandelbrot	13156	9620	26.88%
Pi calculation	11972	8864	25.96%
NQueens (tasks)	20908	19704	5.76%
EPCC-for-static	14096	10992	22.02%
EPCC-critical	12532	9420	24.83%
EPCC-single	12116	8944	26.18%
EPCC-ordered	14048	10992	21.75%

from the Barcelona OpenMP Tasks Suite [80]. This application computes all solutions of the N-queens placement problem on an $N \times N$ chessboard, so that none of the queens threatens any other. Due to the severe memory limitations of the Epiphany, we considered the manual cut-off version of the benchmark, where the nested production of tasks stops at a given depth. We present the results for $N = 12$ queens, and a cut-off value of 2, where a total of 144 tasks are produced. Our last experiment was the Mandelbrot deep zoom application which was discussed in Section 5.2.4.

Size Results

In Table 7.2 we present the sizes in bytes of the resulting object files when our mechanism is employed. Each application is linked with an appropriate optimized RTS as selected by the mapper. For comparison we show the corresponding sizes without applying our mechanism (i.e. the *Full* RTS is linked with the kernels). The last column of table represents the reduction percentage with respect to *Full* RTS case. A quick glance reveals significant improvements in all cases.

For the special case of a kernel with no OpenMP directives the mapper clearly utilized the *NoOMP* RTS, listed as (1) in Section 7.5.3 and the savings were almost 6 KiB, freeing precious space in local memories for the eCORES to fit more application data. For the case of the Mandelbrot application the chosen RTS was the *ParallelOnly* one, which provides only functionalities for creating and synchronizing a parallel team. This resulted in object file smaller by 3 KiB.

The kernel for the calculation of π creates a team of eCORES that share evenly

Table 7.3: EPCC overheads (μsec)

Kernel	Full RTS	Optimized RTS	Reduction
EPCC-for-static	59.13	6.59	88.86%
EPCC-critical	2.06	1.48	28.16%
EPCC-single	44.95	2.03	95.48%
EPCC-ordered	2.95	2.94	0.34%

the workload. The code utilizes the `reduction` clause to combine the partial results. Therefore, the mapper selected the *ParReduction* RTS, which resulted in a savings of 3 KiB. The NQueens application utilizes the `parallel`, `single` and `task` directives. Consequently, the *SingleTasks* runtime library was linked with the kernel. In the cases of the EPCC-based kernels the mapper employed the RTSs (4) through (7) according to kernel directives; the final result exhibits memory savings in excess of 3 KiB.

For completeness, we note that the ESDK versions of the Empty kernel and the Mandelbrot application gave object files with sizes 2248 and 4728 bytes, respectively. Obviously, one cannot compare these with what an OPENMP compiler produces, since the lower-level ESDK API lacks most of the functionality provided by OPENMP. However, we consider important the fact that when OPENMP is not utilized in a kernel of the application, OMPi does not introduce any bloat to the executable (just 6 bytes). Furthermore, the productivity benefits should be clear. For example, while the ESDK version of the Mandelbrot application required separate host and Epiphany programs with a total of 301 lines of code, the OPENMP program was written in a single file with 198 lines.

Timing Results

Following the size results, we compare the execution performance of the optimized kernels with that obtained when the *Full* RTS is employed. Starting with the OPENMP overheads, in Table 7.3 we present timing results for the EPCC microbenchmarks. As mentioned previously, we modified the original suite by having their basic routines offloaded through `target` directives. Time measurements were taken from the host side, after carefully subtracting any offloading costs. These timings, shown in microseconds, corroborate our intuition on the performance benefits of the specialized RTSs. Improvements up to 82% are observed. The noticeable cases are those of `single`

Table 7.4: Application kernels execution times (sec)

Kernel	Full RTS	Optimized RTS	Reduction
Empty Kernel	0.10	0.10	0%
Mandelbrot	30.05	30.00	0.16%
Pi calculation	0.28	0.26	7.14%
NQueens (tasks)	1.81	1.81	0%

and `for` with `static` schedule. The reason is mostly the optimized barrier; in contrast to the *Full* RTS, the runtimes chosen by the mapper contain barriers with no tasking extensions. We get borderline improvements in the case of `for` with an `ordered` clause, because in both scenarios the loop iterations are executed in a serial manner and the eCORES perform their synchronization through a shared variable, stored in the (slow) shared memory.

The execution times (in sec) regarding the other applications are given in Table 7.4. The 0.1 sec of the empty kernel is due to the way the Parallella handles execution on the Epiphany and is a performance burden that any offloaded kernels must bare (even eSDK-based ones). Regarding the Mandelbrot application, most of the execution time is spent on actual calculations, and the OpenMP overheads constitute a rather negligible quantity. Nevertheless, the optimized RTS results offers some minimal speed gains. The same holds for the NQueens kernel. In addition, accesses to the shared memory area which stores the tasks data environments have impact to the total execution time. Finally, a significant improvement of 7% is observed in the kernel that calculates π. The reason behind this is that the optimized runtime does not support tasks. Therefore, it utilizes the lighter barrier which has no tasking extensions. In fact, the barrier flavors are the only algorithmic optimization we implemented in the various RTSs. We expect to get even better performance if other portions of the OpenMP infrastructure are written from scratch, specialized for each different RTS.

We also report that the eSDK version of the Mandelbrot application runs in 26.76 sec; it is approximately 11% faster than our version. We consider this very encouraging, considering that the original is a hand-optimized, bare-metal code, while we only have a general-purpose OpenMP infrastructure prototype which still has room for optimizations.

Chapter 8

Conclusion and future work

8.1 Possible Future Work

Contemporary computing systems integrate multicore chips in order to answer both the never-ending demand for more processing power, as well as the non-functional problem of energy consumption. Personal computers are equipped with multicore CPUs, some of which include two or more groups of cores, thus exhibiting NUMA-like characteristics. The embedded systems market has also turned to multicore CPUs which integrate general and special-purpose cores to increase the performance per watt ratio. The majority of recent systems, follow a heterogeneous paradigm to speedup the execution of specific application parts.

The real challenge in this era of multicore computing proliferation is to provide programming models that enable extracting satisfactory performance while also keeping programmer productivity at high levels. OpenMP is a very intuitive parallel programming model which can help in dealing with these issues. OpenMP is the de facto standard for programming shared-memory multiprocessors. Version 3.0 introduced support for task-based parallelism. With tasking, the expressiveness of OpenMP goes beyond loop-level parallelism and is enriched with constructs that allow one to easily express irregular and dynamic parallelism. In addition, version 4.0 introduced a high-level directive-based approach which allows offloading portions of the application code onto the processing elements of an attached accelerator, while the main part executes on the general-purpose host processor.

In this dissertation we presented the design and implementation of productive and performance-oriented infrastructures for the OpenMP parallel programming model. In particular, in the first part of this thesis we present the design of a general tasking subsystem, and its implementation in the context of the ompi compiler. We present the necessary code transformations and optimizations performed by the compiler, the architecture of the runtime system that supports them and the utilization of *fast* execution paths which optimize task execution performance in certain scenarios. We then re-engineer of parts of the runtime system to match the architecture of modern scalable systems, which exhibit numa characteristics. To exploit these platforms, our runtime system is reorganized in such a way as to maximize local operations and minimize remote accesses which may have detrimental performance effects. This is combined with a very efficient, numa-aware work-stealing queue algorithm. Next, we show how the tasking subsystem can be used to efficiently handle typical cases of nested parallelism. We provide a novel technique, whereby nested parallel loops or sections can be transparently executed by a single level of threads through the existing tasking subsystem, thus avoiding the overheads of nested thread creation and manipulation.

In the second part of the dissertation we present our work related to the design and implementations of efficient OpenMP infrastructure for embedded and heterogeneous multicore systems. We present the first implementation to provide OpenMP support in the sthorm platform. In our view, this will be a reference architecture for future multicore accelerators as it combines the ability to perform general-purpose computations alongside with specialized hardware, while also offering a scalable interconnection structure that will allow large core counts. Next, we talk about the design of the OpenMP device model for the popular Parallella-16 board. It is the first OpenMP implementation for this particular system and also one of few OpenMP 4.0 implementations in general. We discuss the necessary compiler transformations, the general runtime organization, and we deal with the important problem of data environment handling.

Finally, we conclude with the concept of a compiler-assisted adaptive runtime system: Instead of having a single monolithic runtime for a given device, we propose an adaptive, compiler-driven runtime architecture which implements only the OpenMP features required by a particular application. More specifically, the compiler is required to analyse the kernels that are to be offloaded to the device, and to provide

metrics which are later used to select a particular runtime configuration tailored to the needs of the application. The choice of an appropriately optimized runtime may result in dramatically reduced executable sizes and/or lower execution times.

8.1 Possible Future Work

In this section we list some thoughts and ideas for possible future work.

Hierarchical workstealing NUMA architectures are characterized by the hierarchical multilevel memory organization and by varying delays for data accesses. A NUMA-aware OpenMP runtime is lacking. For the tasking subsystem, promising extension for a workstealing mechanism is to schedule the way that a thread visits the TASK_QUEUES of victim threads according to the cache hierarchy. A thread would first try to steal a task from threads that share L1 memory; if the operation does not succeed, then it will attempt to steal a task from threads that share L2 memory etc. Although there has been some work in this area ([101]) there are still open problems.

Device runtime optimizations Our OpenMP 4.0 infrastructure for the Parallella board is not optimized yet and has a number of limitations, as for example the lack of sophisticated management for the shared memory area. Implementing an improved allocator, is a necessary future plan. Towards that direction, we will work on minimizing both the memory footprint of the device runtime as well as its overheads for the OpenMP constructs. Furthermore, we plan to design a new tasking subsystem that takes into account more details of the hardware organization of the Epiphany accelerator. Finally, one of our next targets is the support of the new `teams` and `distribute` directives, which create a given number of thread teams within the accelerator, and divide loop iterations among them.

CARS extensions Regarding our compiler-assisted runtime proposal, we have set goals on three directions: First to optimize the runtime library flavors even more so as to produce even smaller and faster kernels. Second, to work on implementing similar functionality for more platforms. Third, we examine whether new metrics can be added to our code analysis and the potential impact they

may have on further code optimization.

HDI for special devices Our host-device interface (HDI) presented in Section 6.3 was designed as general as possible, in order to support basic and necessary functionalities on a broad range of devices. Nevertheless, this interface could be extended to better exploit devices with special characteristics and features. For example, when executing in specific hardware, these extensions could cover special memory transfers or faster code offloading.

Support for different OpenMP devices Currently, our support for the device model of OpenMP includes a prototype working library for the Epiphany accelerator and a limited initial support for AMD GPGPUs compatible with OpenCL 2.0. We plan to continue our work with OpenCL GPGPUs, since exploiting their massive processing power is a desirable and challenging task. We have identified that some combined target constructs fit very well to the organization of GPGPUs (e.g. the `target teams distribute parallel for` combined construct), so an optimized implementation for them could greatly enhance performance.

OpenMP extensions During our experimentation with the Parallella board and the OpenMP device model, we had difficulties when trying restructure the code of some applications in order to take advantage of the Epiphany coprocessor. Here we present some possible OpenMP extensions that may help to better exploit the processing power of such an attached accelerator.

1. **Data block transfer.** MPSoCs usually come with fast but quite limited local memories which cannot fit large program data structures. A common technique to alleviate this limitation, is the overlapping of computations and data transfers (between slow main memory and fast local memories) through dedicated DMA hardware. However the OpenMP device model does not include any directives that can be used towards this end. We believe that new data mapping clauses should be added to provide hints to the OpenMP compiler about the usage of the respective variables. For example a new parameter, that accompanies the `map` clause for a 2D array M, could specify M is to be processed row by row (i.e. in a `for`-loop). Having this information, the compiler could transparently generate DMA code to bring in the appropriate row at each iteration while at the same time working on

another row.

2. **Resident kernels.** We observed that a major overhead is the time needed to offload a kernel. If the computation is structured in a way where kernels are repeatedly offloaded, to operate on different data each time, then it is questionable whether the application will experience significant performance gains. For such scenarios, we believe that support of *resident kernels* may bring considerable improvements. A kernel would be offloaded only once, while at specific points the host would communicate new data (through target updates) for the kernel to operate on.

Bibliography

[1] D. Evans, "The Internet of Things: How the Next Evolution of the Internet Is Changing Everything," *White paper*, *Cisco Internet Business Solutions Group (IBSG)*, 2011.

[2] L. Benini, E. Flamand, D. Fuin, and D. Melpignano, "P2012: Building an Ecosystem for a Scalable, Modular and high-efficiency Embedded Computing Accelerator," in *Proc. of DATE'12*, *Design Automation and Test in Europe*, (Dresden, Germany), pp. 983–987, Mar. 2012.

[3] Intel® Xeon Phi™ Product Family, "http://www.intel.com/content/www/us/en/processors/xeon/xeon-phi-detail.html."

[4] C. Research, "Guide to Automatic Vectorization with Intel AVX-512 Instructions in Knights Landing Processors, Online publication," tech. rep., May. 2016.

[5] Adapteva, "Parallella Reference Manual," Sept. 2014.

[6] OpenMP ARB, "OpenMP Application Program Interface V3.1," July 2011.

[7] V. V. Dimakopoulos, E. Leontiadis, and G. Tzoumas, "A portable C compiler for OpenMP V.2.0," in *Proc. EWOMP 2003*, *the 5th European Workshop on OpenMP*, (Aachen, Germany), pp. 5–11, Sept. 2003.

[8] S. N. Agathos, P. E. Hadjidoukas, and V. V. Dimakopoulos, "Design and Implementation of OpenMP Tasks in the OMPi Compiler," in *Proc. PCI '11*, *the 15th Panhellenic Conference on Informatics*, (Kastoria, Greece), pp. 265–269, Sept. 2011.

[9] S. N. Agathos, N. D. Kallimanis, and V. V. Dimakopoulos, "Speeding Up OpenMP Tasking," in *Proc. Euro-Par 2012*, *the 18th International European Con-*

ference on Parallel and Distributed Computing, (Rhodes Island, Greece), pp. 650–661, Aug. 2012.

[10] S. N. Agathos, P. E. Hadjidoukas, and V. V. Dimakopoulos, "Task-based Execution of Nested OpenMP Loops," in *Proc IWOMP 2012, the 8th International Workshop on OpenMP, Heterogenous Execution and Data Movements*, (Rome, Italy), pp. 210–222, June 2012.

[11] S. N. Agathos, V. V. Dimakopoulos, A. Mourelis, and A. Papadogiannakis, "Deploying OpenMP on an embedded multicore accelerator," in *Proc. of SAMOS '13, the 13th International Conference on Embedded Computer Systems: Architectures, MOdeling and Simulation*, (Samos, Greece), pp. 180–187, IEEE, Jul. 2013.

[12] OpenMP ARB, "OpenMP Application Program Interface V4.0," Jul. 2013.

[13] S. N. Agathos, A. Papadogiannakis, and V. V. Dimakopoulos, "Targeting the Parallella," in *Proc. Euro-Par 2015, the 21st International European Conference on Parallel and Distributed Computing*, (Vienna, Austria), pp. 662–674, Aug. 2015.

[14] A. Papadogiannakis, S. N. Agathos, and V. V. Dimakopoulos, "OpenMP 4.0 Device Support in the OMPi Compiler," in *Proc IWOMP 2015, the 11th International Workshop on OpenMP, Heterogenous Execution and Data Movements*, (Aachen, Germany), pp. 202–216, Oct. 2015.

[15] S. N. Agathos and V. V. Dimakopoulos, "Compiler-Assisted OpenMP Runtime Organization for Embedded Multicores," Tech. Rep. 2016-01, University of Ioannina, Department of Computer Science & Engineering, Apr. 2016.

[16] OpenMP ARB, "OpenMP Application Program Interface V3.1," July 2011.

[17] P. E. Hadjidoukas and V. V. Dimakopoulos, "Nested Parallelism in the OMPi OpenMP/C Compiler," in *Proc. Euro-Par '07, the 13th International Euro-Par Conference on Parallel Processing*, (Rennes, France), pp. 662–671, Aug. 2007.

[18] V. V. Dimakopoulos, "Parallel Programming Models," in *Smart Multicore Embedded Systems* (Torquati, Massimo and Bertels, Koen and Karlsson, Sven and Pacull, Francois, ed.), ch. 1, pp. 3–20, Springer Publishing Company, Incorporated, 2013.

[19] M. Frigo, C. E. Leiserson, and K. H. Randall, "The Implementation of the Cilk-5 Multithreaded Language," in *Proc. PLDI '98, the Conference on Programming Language Design and Implementation*, (Montreal, Quebec, Canada), pp. 212–223, June 1998.

[20] T. Gautier, X. Besseron, and L. Pigeon, "KAAPI: A thread scheduling runtime system for data flow computations on cluster of multi-processors," in *Proc. PASCO '07, the international workshop on Parallel symbolic computation*, (London, Ontario, Canada), pp. 15–23, July 2007.

[21] K. Arvind and R. S. Nikhil, "Executing a Program on the MIT Tagged-Token Dataflow Architecture," *IEEE Transactions on Computers*, vol. 39, pp. 300 – 318, Mar. 1990.

[22] Dally, William J. and Chao, Linda and Chien, Andrew A. and Hassoun, Soha and Horwat, Waldemar and Kaplan, Jon and Song, Paul and Totty, Brian and Wills, D. Scott, "Architecture of a Message-Driven Processor," in *Proc. ISCA '87, the 14th Annual International Symposium on Computer Architecture*, (Pittsburgh, PA, USA), pp. 189 – 196, June 1987.

[23] J. B. Dennis and D. P. Misunas, "A Preliminary Architecture for a Basic Data-flow Processor," *SIGARCH Computer Architure News*, vol. 3, pp. 126 – 132, Dec. 1974.

[24] G. M. Papadopoulos and D. E. Culler, "Monsoon: An Explicit Token-store Architecture," in *Proc. ISCA '90, the 17th Annual International Symposium on Computer Architecture*, (Seattle, Washington, USA), pp. 82 – 91, May 1990.

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

[26] R. V. Van Nieuwpoort, G. Wrzesińska, C. J. H. Jacobs, and H. E. Bal, "Satin: A high-level and efficient grid programming model," *ACM Transactions on Programming Languages and Systems*, vol. 32, pp. 9:1–9:39, Mar. 2010.

[27] M. C. Rinard and M. S. Lam, "The Design, Implementation, and Evaluation of Jade," *ACM Transactions on Programming Languages and Systems*, vol. 20, pp. 483–545, May 1998.

[28] L. Hendren, X. Tang, Y. Zhu, G. Gao, X. Xue, H. Cai, and P. Ouellet, "Compiling C for the EARTH Multithreaded Architecture," in *International Journal of Parallel Programming*, vol. 25, pp. 305–338, IEEE Computer Society Press, Aug 1997.

[29] D. E. Culler, A. Dusseau, S. C. Goldstein, A. Krishnamurthy, S. Lumetta, T. von Eicken, and K. Yelick, "Parallel programming in Split-C," in *Proc. SC '93, the International Conference for High Performance Computing, Networking, Storage and Analysis*, (Oregon, Portland), pp. 262–273, ACM, Nov 1993.

[30] T. El-Ghazawi and L. Smith, "UPC: Unified Parallel C," in *Proc. of SC '06, the 2006 ACM/IEEE Conference on Supercomputing*, (Tampa, Florida), Nov. 2006.

[31] M. Sato, Y. Kodama, S. Sakai, and Y. Yamaguchi, "EM-C: Programming with Explicit Parallelism and Locality for EM-4 Multiprocessor," in *Proc. IFIP WG10.3, the Working Conference on Parallel Architectures and Compilation Techniques (PACT '94)*, (Montreal, Canada), pp. 3–14, Aug. 1994.

[32] OpenMP ARB, "OpenMP Application Program Interface V4.5," Nov. 2015.

[33] E. Ayguadé, N. Copty, A. Duran, J. Hoeflinger, Y. Lin, F. Massaioli, E. Su, P. Unnikrishnan, and G. Zhang, "A Proposal for Task Parallelism in OpenMP," in *Proc. IWOMP '07, the 3rd International Workshop on OpenMP, A Practical Programming Model for the Multi-Core Era*, (Beijing, China), pp. 1–12, June 2007.

[34] E. Ayguadé, A. Duran, J. Hoeflinger, F. Massaioli, and X. Teruel, "An Experimental Evaluation of the New OpenMP Tasking Model," in *Proc. LCPC '07, the 20th International Workshop on Languages and Compilers for Parallel Computing*, (Urbana, Illinois, USA), pp. 63–77, Oct. 2007.

[35] Free Software Foundation, "GCC, the GNU compiler collection."

[36] C. Liao, O. Hernandez, B. Chapman, W. Chen, and W. Zheng, "OpenUH: An Optimizing, Portable OpenMP Compiler: Research Articles," *Concurrency and Computation : Practical and Experience*, vol. 19, pp. 2317–2332, Dec. 2007.

[37] D. B. Kirk and W. mei W. Hwu, *Programming Massively Parallel Processors, Second Edition: A Hands-on Approach*. MA 01803, USA: Morgan Kaufmann, Dec. 2012.

[38] R. Dolbeau, S. Bihan, and F. Bodin, "HMPP: A Hybrid Multi-core Parallel Programming Environment," in *Proc. GPGPU 2007, Workshop on General Purpose Processing Using GPUs*, (Boston, MA USA), Oct. 2007.

[39] Khronos Group, "http://www.khronos.org/opencl/."

[40] J. Labarta, "StarSs: A Programming Model for the Multicore Era," in *PRACE Workshop 2010, New Languages & Future Technology Prototypes*, (Leibniz, Germany), Mar. 2010.

[41] A. Fernández, V. Beltran, X. Martorell, R. M. Badia, E. Ayguadé, and J. Labarta, "Task-based programming with ompss and its application," in *Euro-Par 2014 International Workshop, Revised Selected Papers, Part II*, (Porto, Portugal), pp. 602 – 613, Aug.

[42] E. Ayguadé, R. M. Badia, F. Igual, J. Labarta, R. Mayo, and E. S. Quintana-Ortí, "An Extension of the StarSs Programming Model for Platforms with Multiple GPUs," in *Proc. of Euro-Par 2009, the 15th International Euro-Par Conference on Parallel Processing*, (Delft, The Netherlands), pp. 851 – 862, Aug. 2009.

[43] OpenACC, "The OpenACC™ Application Programming Interface Vesion 2.5," Nov. 2015.

[44] Intel Corporation, "User and Reference Guide for the Intel® C++ Compiler 15.0, OpenMP* Support."

[45] C. J. Newburn, R. Deodhar, S. Dmitriev, R. Murty, R. Narayanaswamy, J. Wiegert, F. Chinchilla, and R. McGuire, "Offload Compiler Runtime for the Intel® Xeon Phi™Coprocessor," in *Proc. IPDPS Workshops, the 27th IEEE International Parallel and Distributed Processing Symposium*, (Boston, USA), pp. 1213–1225, May 2013.

[46] GCC 5 Release Series, "https://gcc.gnu.org/gcc-5/changes.html."

[47] L. Chunhua, Y. Yonghong, B. R. de Supinski, D. J. Quinlan, and B. M. Chapman, "Early Experiences with the OpenMP Accelerator Model," in *Proc. IWOMP 2013, the 9th International Workshop on OpenMP, OpenMP in the Era of Low Power Devices and Accelerators*, (Canberra, Australia), pp. 84–98, Sept. 2013.

[48] C. Bertolli, S. F. Antao, A. E. Eichenberger, K. O'Brien, Z. Sura, A. C. Jacob, T. Chen, and O. Sallenave, "Coordinating GPU Threads for OpenMP 4.0 in LLVM," in *Proc. LLVM-HPC '14, LLVM Compiler Infrastructure in HPC*, (New Orleans, Louisiana), pp. 12–21, Nov. 2014.

[49] G. Mitra, E. Stotzer, A. Jayaraj, and A. P. Rendell, "Implementation and Optimization of the OpenMP Accelerator Model for the TI Keystone II Architecture," in *Proc. IWOMP 2014, the 10th International Workshop on OpenMP, Using and Improving OpenMP for Devices, Tasks, and More*, (Salvador, Brazil), pp. 202–214, Sept. 2014.

[50] D. Cabrera, X. Martorell, G. Gaydadjiev, E. Ayguadé, and D. Jiménez-González, "OpenMP extensions for FPGA accelerators," in *Proc. SAMOS '09, the 9th International Conference on Embedded Computer Systems: Architectures, MOdeling and Simulation*, (Samos, Greece), pp. 17–24, Jul. 2009.

[51] M. Sato, Y. Nakajima, Y. Ojima, and Y. Hotta, "OpenMP Implementation and Performance on Embedded Renesas M32R Chip Multiprocessor," in *Proc. EWOMP '04, the 6th European Workshop on OpenMP*, (Stockholm, Sweden), pp. 37–42, Oct. 2004.

[52] T. Hanawa, M. Sato, J. Lee, T. Imada, H. Kimura, and T. Boku, "Evaluation of Multicore Processors for Embedded Systems by Parallel Benchmark Program Using OpenMP," in *Proc. IWOMP '09, the 5th International Workshop on OpenMP: Evolving OpenMP in an Age of Extreme Parallelism*, (Dresden, Germany), pp. 15–27, June 2009.

[53] F. Liu and V. Chaudhary, "A Practical OpenMP Compiler for System on Chips," in *Proc. WOMPAT 2003, the Workshop on OpenMP Applications and Tools*, (Torondo, Canada), pp. 54–68, June 2003.

[54] Feng Liu and Vipin Chaudhary, "Extending OpenMP for Heterogeneous Chip Multiprocessors," in *Proc. ICPP 2003, the 32nd International Conference on Parallel Processing*, (Kaohsiung, Taiwan), pp. 161–, Oct. 2003.

[55] W.-C. Jeun and S. Ha, "Effective OpenMP Implementation and Translation For Multiprocessor System-On-Chip without Using OS," in *Proc. ASP-DAC '07, the*

12th Asia and South Pacific Design Automation Conference, (Yokohama, Japan), pp. 44–49, Jan. 2007.

[56] M. Gonzàlez, E. Ayguadé, X. Martorell, and J. Labarta, "Exploiting pipelined executions in OpenMP," in *Proc. ICPP 2003, the 32nd International Conference on Parallel Processing*, (Kaohsiung, Taiwan), pp. 153–160, Oct. 2003.

[57] P. Carpenter, D. Rodenas, X. Martorell, A. Ramirez, and E. Ayguadé, "A streaming machine description and programming model," in *Proc. SAMOS '07, the 7th International Conference on Embedded Computer Systems: Architectures, MOdeling and Simulation*, (Samos, Greece), pp. 107–116, Jul. 2007.

[58] B. Chapman, L. Huang, E. Biscondi, E. Stotzer, A. Shrivastava, and A. Gatherer, "Implementing OpenMP on a high performance embedded multicore MPSoC," in *Proc. IPDPS '09, the IEEE International Symposium on Parallel& Distributed Processing*, (Rome, Italy), pp. 1–8, May 2009.

[59] P. Burgio, G. Tagliavini, A. Marongiu, and L. Benini, "Enabling Fine-Grained OpenMP Tasking on Tightly-Coupled Shared Memory Clusters," in *Proc. DATE 13, Design Automation and Test in Europe*, (Grenoble, France), pp. 1504–1509, Mar. 2013.

[60] A. Basu, S. Bensalem, M. Bozga, J. Mottin, F. Pacull, A. Poulakidas, and A. Aggelis, "System-Level Modeling, Analysis and Code Generation: Object Recognition Case Study," in *Proc. of Embedded World Conference 2012*, (Nuremberg, Germany), Feb. 2012.

[61] Adapteva, "Epiphany SDK reference Manual," Sept. 2013.

[62] Brown deer Technology COPRTHR, "http://www.browndeertechnology.com/coprthr.htm," 2016.

[63] X. Teruel, P. Unnikrishnan, X. Martorell, E. Ayguade, R. Silvera, G. Zhang, and E. Tiotto, "OpenMP tasks in IBM XL compilers," in *Proc. CASCON '08, the 2008 conference of the centre for advanced studies on collaborative research*, (Ontario, Canada), pp. 207–221, Oct. 2008.

[64] X. Teruel, X. Martorell, A. Duran, R. Ferrer, and E. Ayguadé, "Support for OpenMP tasks in Nanos v4," in *Proc. CASCON '07, the 2007 conference of the*

centre for advanced studies on Collaborative research, (Ontario, Canada), pp. 256–259, Oct 2007.

[65] S. Shah, G. Haab, P. Petersen, and J. Throop, "Flexible control structures for parallelism in OpenMP," *Concurrency and Computation: Practice and Experience*, vol. 12, no. 12, pp. 1219–1239, 2000.

[66] J. Balart, A. Duran, M. Gonzàlez, X. Martorell, E. Ayguadé, and J. Labarta, "Nanos Mercurium: a Research Compiler for OpenMP," in *Proc. EWOMP '04, the 6th European Workshop on OpenMP*, (Stockholm, Sweden), pp. 103–109, Oct. 2004.

[67] Intel threading Building Blocks, "https://www.threadingbuildingblocks.org/," 2016.

[68] M. Korch and T. Rauber, "A comparison of task pools for dynamic load balancing of irregular algorithms: Research Articles," *Concurrency and Computation: Practice and Experience*, vol. 16, pp. 1–47, Dec. 2003.

[69] P. Brucker, *Scheduling Algorithms*. Secaucus, NJ, USA: Springer-Verlag New York, Inc., 3rd ed., 2001.

[70] J. Clet-Ortega, P. Carribault, and M. Pérache, "Evaluation of OpenMP Task Scheduling Algorithms for Large NUMA Architectures," in *Proc. Euro-Par 2014, the 20th International Euro-Par Conference on Parallel Processing*, (Porto, Portugal), pp. 596–607, Aug. 2014.

[71] A. Duran, J. Corbalán, and E. Ayguadé, "Evaluation of OpenMP task scheduling strategies," in *Proc. IWOMP'08, the 4th international conference on OpenMP in a new era of parallelism*, (West Lafayette, IN, USA), pp. 100–110, May 2008.

[72] S. L. Olivier, A. K. Porterfield, K. B. Wheeler, and J. F. Prins, "Scheduling task parallelism on multi-socket multicore systems," in *Proc. ROSS '11, the 1st International Workshop on Runtime and Operating Systems for Supercomputers*, (Tucson, Arizona), pp. 49–56, May 2011.

[73] F. Broquedis, J. Clet-Ortega, S. Moreaud, N. Furmento, B. Goglin, G. Mercier, S. Thibault, and R. Namyst, "hwloc: a Generic Framework for Managing Hardware Affinities in HPC Applications," in *Proc. PDP 2010, the 18th Euromi-*

cro International Conference on Parallel, Distributed and Network-Based Computing, (Pisa, Italy), pp. 180–186, Feb. 2010.

[74] A. Duran, J. Corbalán, and E. Ayguadé, "An adaptive cut-off for task parallelism," in *Proc. SC '08, the 2008 ACM/IEEE conference on Supercomputing*, (Austin, Texas), pp. 1–11, Nov. 2008.

[75] D. Libenzi, "http://www.xmailserver.org/libpcl.html."

[76] C. D. Polychronopoulos, "Nano-Threads: Compiler Driven Multithreading," in *Proc. CPC'93, the 4th International Workshop on Compilers for Parallel Computing*, (Delft, The Netherlands), Dec. 1993.

[77] J.-H. Chow, L. E. Lyon, and V. Sarkar, "Automatic Parallelization for Symmetric Shared-memory Multiprocessors," in *Proc. CASCON '96, the Conference of the Centre for Advanced Studies on Collaborative Research*, pp. 76–79, Nov. 1996.

[78] R. D. Blumofe, C. F. Joerg, B. C. Kuszmaul, C. E. Leiserson, K. H. Randall, and Y. Zhou, "Cilk: An Efficient Multithreaded Runtime System," *Journal of Parallel and Distributed Computing*, vol. 37, no. 1, pp. 55–69, 1996.

[79] D. Chase and Y. Lev, "Dynamic circular work-stealing deque," in *Proc. SPAA '05, the 17th annual ACM symposium on Parallelism in algorithms and architectures*, (Las Vegas, Nevada, USA), pp. 21–28, July 2005.

[80] A. Duran, X. Teruel, R. Ferrer, X. Martorell, and E. Ayguadé, "Barcelona OpenMP Tasks Suite: A Set of Benchmarks Targeting the Exploitation of Task Parallelism in OpenMP," in *Proc. ICPP '09, the 38th International Conference on Parallel Processing*, (Vienna, Austria), pp. 124–131, Sept. 2009.

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

[82] A. Tzannes, G. C. Caragea, R. Barua, and U. Vishkin, "Lazy binary-splitting: a run-time adaptive work-stealing scheduler," in *Proc. PPoPP '10, the 15th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, (Bangalore, India), pp. 179–190, Jan. 2010.

[83] P. Fatourou and N. D. Kallimanis, "Revisiting the combining synchronization technique," in *Proc. PPoPP '12, the 17th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, (New Orleans, Louisiana, USA), pp. 257–266, Feb. 2012.

[84] M. M. Michael and M. L. Scott, "Simple, fast, and practical non-blocking and blocking concurrent queue algorithms," in *Proc. PODC '96, the 15th annual ACM symposium on Principles of distributed computing*, (Philadelphia, Pennsylvania, USA), pp. 267–275, May 1996.

[85] P. Fatourou and N. D. Kallimanis, "A highly-efficient wait-free universal construction," in *Proc. SPAA '11, the 23rd ACM symposium on Parallelism in algorithms and architectures*, (San Jose, California, USA), pp. 325–334, Jun. 2011.

[86] C. E. Leiserson, "The Cilk++ concurrency platform," *Journal of Supercomputing*, vol. 51, pp. 244–257, 2010.

[87] OpenMP ARB, "OpenMP Application Program Interface V4.1," July 2015.

[88] P. E. Hadjidoukas, V. V. Dimakopoulos, M. Delakis, and C. Garcia, "A high-performance face detection system using OpenMP," *Concurrency and Computation: Practice and Experience*, vol. 21, pp. 1819–1837, Oct. 2009.

[89] M. Sato, M. S. Shigehisa, K. Kusano, and Y. Tanaka, "Design of OpenMP Compiler for an SMP Cluster," in *Proc. EWOMP '99, the 1st European Workshop on OpenMP*, (Lund, Sweden), pp. 32–39, Sep. 1999.

[90] C. Brunschen and M. Brorsson, "OdinMP/CCp - a portable implementation of OpenMP for C," *Concurrency - Practice and Experience*, vol. 12, pp. 1193–1203, 2000.

[91] LLVM Compiler Infrastructure, "http://llvm.org/," 2015.

[92] J. M. Bull, "Measuring Synchronisation and Scheduling Overheads in OpenMP," in *Proc. EWOMP '99, the 1st European Workshop on OpenMP*, (Lund, Sweden), pp. 99–105, Sept. 1999.

[93] H. Rowley, S. Baluja, and T. Kanade, "Neural network-based face detection," *IEEE Transactions on Pattern Analysis and Machine Intelligence*, vol. 20, pp. 23–28, 1998.

[94] Artemis Industry Association, Project 100230 SMECY, "https://artemis-ia.eu/project/25-smecy.html," 2013.

[95] Torquati, Massimo and Bertels, Koen and Karlsson, Sven and Pacull, Francois, *Smart Multicore Embedded Systems*. Springer Publishing Company, Incorporated, 2013.

[96] The Multicore Association, "http://www.multicore-association.org/," 2016.

[97] M. Torquati, M. Vanneschi, M. Amini, and G. et. al, "An innovative compilation tool-chain for embedded multi-core architectures," in *Proc. of Embedded World Conference 2012*, (Nuremberg, Germany), Feb. 2012.

[98] A. W. Appel, *Modern Compiler Implementation in C*. Cambridge, UK: Cambridge University Press, 1999.

[99] G. Philos, V. Dimakopoulos, and P. Hadjidoukas, "A Runtime Architecture for Ubiquitous Support of OpenMP," in *Proc. ISPDC 2008*, *the 7th International Symposium on Parallel and Distributed Computing*, (Krakow, Poland), pp. 189–196, June 2008.

[100] G. Zhang, R. Silvera, and R. Archambault, "Structure and algorithm for implementing OpenMP workshares," in *Proc. WOMPAT '04*, *the 5th Workshop on OpenMP Applications and Tools*, (Houston, TX, USA), May 2004.

[101] Thibault, Samuel and Namyst, Raymond and Wacrenier, Pierre-André in *Proc. Euro-Par 2007*, *the 13th International Euro-Par Conference on Parallel Processing*, (Rennes ,France), pp. 42–51, Aug. 2007.

Appendix A

Host Device Interface

The calls of HDI:

```
/**
 * Host passes the name of the module
 *
 * @param modname the name of the module
 *
 */
void  hm_set_module_name(char *modname){}

/**
 * Calculates the number of available devices supported by this module
 *
 * @return number of devices
 */
int   hm_get_num_devices(void){}

/**
 * Prints information for this module and the available devices
 *
 * @param device_offset the id of the first device available from this module
 */
void  hm_print_information(int device_offset) {}
```

```
/**
 * Registers host runtime functions (currently it registers functions for locks)
 *
 * @param init_lock_in pointer to the function used for initializing a lock.
 *                     It's parameters are the address of a "void *" variable
 *                     and one denoting the type of the lock
 * @param lock_in      pointer to the function used for acquiring a lock
 * @param unlock_in    pointer to the function used for releasing a lock
 * @param hyield_in    pointer to the function used for thread yield
 */
void  hm_register_ee_calls(void (*init_lock_in)(void **lock, int type),
                           void (*lock_in)(void **lock),
                           void (*unlock_in)(void **lock),
                           int  (*hyield_in)(void)){}
/**
 * Initializes a device
 *
 * @param dev_num the id of the device to initialize
 *                (0 <= dev_num < hm_get_num_devices())
 * @param ort_icv Pointer to struct with initial values for the device ICVs.
 *
 * @return device_info pointer that will be used in further calls.
 *                     Return null only if it failed to initialize
 */
void *hm_initialize(int dev_num, ort_icvs_t *ort_icv) {}

/**
 * Finalizes a device
 *
 * @param device_info the device to finalize
 */
void  hm_finalize(void *device_info) {}

/**
 * Offloads and executes a kernel file.
 *
 * @param device_info         the device
 * @param host_func pointer   to offload function on host address space
 * @param dev_data pointer    to a struct containing kernel variables
 * @param decl_data pointer   to a struct containing globally declared
 *                            variables
 * @param kernel_filename_prefix filename of the kernel (without the suffix)
 * @param num_teams           num_teams clause from "teams" construct
 * @param thread_limit        thread_limit clause from "teams" construct
 * @param argptr              The addresses of all target data and target
 *                            declare variables (only used in OpenCL devices)
 */
void  hm_offload(void *device_info, void *(*host_func)(void *),
                 void *dev_data, void *decl_data,
                 char *kernel_filename_prefix, int num_teams,
                 int thread_limit, va_list argptr){}
```

```
/**
 * Allocates memory on the device
 *
 * @param device_info the device
 * @param size        the number of bytes to allocate
 * @param map_memory  used in OpenCL, when set to 1 additionaly to the memory
 *                    allocation in shared virtual address space, the memory
 *                    is mapped with read/write permissions so the host cpu
 *                    can utilize it.
 * @return hostaddr a pointer to the allocated space
 */
void *hm_dev_alloc(void *device_info, size_t size, int map_memory) {}

/**
 * Frees data allocated with hm_dev_alloc
 *
 * @param device_info  the device
 * @param hostaddr     pointer to the memory that will be released
 * @param unmap_memory used in OpenCL, when set to 1 prior to the memory
 *                     deallocation, the memory is unmapped.
 */
void  hm_dev_free(void *device_info, void *devaddr, int unmap_memory) {}

/**
 * Transfers data from the host to a device
 *
 * @param device_info the device
 * @param hostaddr    the source memory
 * @param devaddr     the target memory
 * @param size        the size of the memory block
 */
void  hm_todev(void *device_info, void *hostaddr, void *devaddr, size_t
               size) {}

/**
 * Transfers data from a device to the host
 *
 * @param device_info the source device
 * @param hostaddr    the target memory
 * @param devaddr     the source memory
 * @param size        the size of the memory block
 */
void  hm_fromdev(void *device_info, void *hostaddr, void *devaddr,
                 size_t size) {}
```

```
/**
 *  Returns a pointer in the device address space
 *
 * @param device_info the device
 * @param devaddr     allocated memory from hm_dev_alloc
 *
 * @return pointer containing the address on which code running on the device
 *         can access hostaddr
 */
void *hm_get_dev_address(void *device_info, void *devaddr) {}
```

Author's Publications

Conferences and Journals

1. Alexandros Papadogiannakis, Spiros N. Agathos, Vassilios V. Dimakopoulos, "OpenMP 4.0 Device Support in the Ompi Compiler", IWOMP 12, International Workshop on OpenMP, Heterogenous Execution and Data Movements, Aachen, Germany, October 2015, pages 202 - 216.

2. Spiros N. Agathos, Alexandros Papadogiannakis, Vassilios V. Dimakopoulos, "Targeting the Parallella", Europar 2015, International European Conference on Parallel and Distributed Computing, Vienna, Austria, August 2015, pages 662 - 674.

3. Spiros Agathos, Evangelos Papapetrou, "On the Set Cover problem for Broadcasting in Wireless Ad Hoc Networks", IEEE Communications Letters, 26 September 2013. Volume 17, Issue 11, pages 2192 - 2195.

4. Spiros N. Agathos, Vassilios V. Dimakopoulos, Aggelos Mourelis, Alexandros Papadogiannakis, "Deploying OpenMP on an Embedded Multicore Accelarator", SAMOS XIII, International Conference on Embedded Computer Systems: Architectures, MOdeling and Simulation, Samos, Greece, July 2013, pages 180 - 187.

5. Spiros N. Agathos, Nikolaos D. Kallimanis, Vassilios V. Dimakopoulos, "Speeding Up OpenMP Tasking", Europar 2012, International European Conference on Parallel and Distributed Computing, Rhodes, Greece, August 2012, pages 650 - 661.

6. Spiros N. Agathos, Panagiotis E. Hadjidoukas, Vassilios V. Dimakopoulos, "Task-based Execution of Nested OpenMP Loops", IWOMP 12, International Workshop on OpenMP, OpenMP in a Heterogenous World, Rome, Italy, June 2012, pages 210 - 222.

7. Spiros Agathos, Eyangelos Papapetrou, "Efficient Broadcasting using Packet History in Mobile Ad-Hoc Networks", Communications IET, 14 October 2011, Volume :5 Issue :15, Pages 2196-2205.

8. Spiros N. Agathos, Panagiotis E. Hadjidoukas, Vassilios V. Dimakopoulos, "Design and Implementation of OpenMP Tasks in the OMPi Compiler", PCI 2011, 15th Panhellenic Conference on Informatics, Kastoria, Greece, September 2011, pages 265 - 269.

Technical Reports

1. Spiros N. Agathos, Vassilios V. Dimakopoulos, "Compiler-Assisted OpenMP Runtime Organization for Embedded Multicores", Technical Report, Number 2016-01, University of Ioannina, Department of Computer Science & Engineering, April 2016.

Submitted for Publication

1. Spiros N. Agathos, Vassilios V. Dimakopoulos "Compiler-Assisted OpenMP Runtime Organization for Embedded Multicores", *submitted for Journal publication*.

2. Spiros N. Agathos, Panagiotis E. Hadjidoukas, Vassilios V. Dimakopoulos, "Reducing OpenMP Nested Loop Parallelism Overheads Through Tasking", *submitted for journal publication*

Short Biography

Spiros N. Agathos was born in Corfu island, Greece in 1982. He received his BSc (Ptychion) from the Department of Computer Science, University of Ioannina in 2005. He received his MSc with specialization in Computer Systems, from the same department in 2008. In 2000 he received an award for entering with the 2nd best grade in the Department of Physics after participating in the greek national exams. In 2010 he earned a scholarship for pursuing a PhD degree from the Greek State Scholarship Foundation (IKY) after participating in written exams.

His research interests include parallel programming models, multicore embedded systems and performance analysis with emphasis in the design and development of runtime systems. He has published papers in international conferences in the area of parallel programming models and embedded systems, such as Europar, IWOMP and SAMOS.