Implementation of OpenMP on Application-Specific Architectures

Cláudia Henriques
claudiafilipahenriques@tecnico.ulisboa.pt

Instituto Superior Técnico, Lisboa, Portugal

November 2014

Abstract

Currently, heterogeneous embedded systems are available in any place and their processing capacities are increasing. This processing capacity is result of the constant development of the computational world where we all live in. However, this capacities are not always used in the better way since, generally, the development of parallel programs for those systems is dificult and require too much time. On the other hand, for general purpose systems, like the computer we have at home, the are reliable tools, like OpenMP, that overcome these problems by automating the parallelization process of a given code. It is here purposed a model that supports OpenMP for heterogenous embedded architectures.
Keywords: OpenMP, Heterogeneous Embedded Systems, Parallel

1. Introduction

Open Multi-Processing (widely known as OpenMP) is a portable and scalable programing model whose main goal is to provide a simple and flexible interface for developing parallel applications. OpenMP allows the user to produce easy-to-read code by hiding all necessary communication details on OpenMP constructs (thus making parallelization of code an easier task). Even so, OpenMP offers both coarse and fine grained control over parallelism. This model has became a de-facto standard in the parallel programming scope and it is supported by many compilers and processor architectures.

This document presents a solution for developing parallel programs for heterogeneous embedded architectures. The solution is achieved by introducing support for executing OpenMP programs on heterogeneous embedded systems, without subjecting the OpenMP API to any change. The goals of this work relate to the high reliability, portability, scalability and performance characteristics of the developed model.

1.1. Motivation

Over the past few decades, the application areas of embedded systems grew not only in number but also in complexity. Nowadays, embedded systems can be seen in areas like automotive electronics, consumer electronics, medical electronics or building automation. Result, demanding requirements about the processing capacity of these systems are mandated.

Heterogeneous computing platforms refer to multi-core systems that gain performance not only by increasing the number of cores, but also by incorporating processing elements specialized to handle specific tasks. Heterogeneous computing platforms are composed by processors with different Instruction Set Architectures (ISAs), thus making parallel programming a complex task. Since the processing of CPU-intensive tasks on heterogeneous embedded systems is a current need, the parallelization of applications for these systems should not be complicated. Accordingly, this document provides support for the execution of OpenMP applications on heterogeneous embedded systems. This way, programmers will be able to develop parallel applications for their specific architectures with a model they probably already know and whose benefits have already been proven worldwide (specially the ones related to avoidance of errors and time waste, during the development phase).

2. Context

OpenMP is an API for user-directed parallelization that is composed by a set of compiler directives, library routines and environment variables. The API is written for C, C++ and Fortran and it provides the user with a portable and scalable parallel programming model. The portability across various shared memory platforms is result of the fact that OpenMP hides all communication needs and resource management behind OpenMP constructs. Throughout this chapter it is described the background knowledge the reader should have in order to fully understand the present work. It is focused on the description of the OpenMP programming model from user perspective, including the execution, memory and synchronization models of

OpenMP 4.0. The classical compilation process of OpenMP programs is also discussed at the end of the chapter.

2.1. Execution Model

The execution model of OpenMP is based on the creation and management of threads, who are required to execute a given parallel region. OpenMP constructs are divided into four categories: parallel constructs (responsible for creating teams of threads), work-sharing constructs (who distribute work by the threads available on the team), tasking constructs (which are used to explicitly define tasks to be executed in parallel with the code outside the task region) and SIMD constructs (who provide support for Single Instruction, Multiple Data operations). This work is focused the first two categories.

Every program is started by a single thread of execution, called initial thread. It is responsible for the sequential execution of the program, until a parallel construct is found. Whenever the initial thread encounters a parallel construct, it creates a team composed by one or more threads (that is, itself and zero or more additional threads) and it becomes the master of the new team. Every thread of the team, including the master, executes the parallel region (which was exported to a new function at compile time) and enters an implicit barrier where the execution of the team is synchronized. By the time the master thread reaches the barrier and it notices all threads have also reached the barrier, the team is joint by the master and it proceeds executing, sequentially.

By default, the cardinality of a team is equivalent to the number of processing units of the CPU. This number is nevertheless changeable, depending on the environment variables, the clauses specified on the parallel construct and the library routines called before the parallel construct. Regarding the identification number assigned to each thread, the master is always assigned the id 0, while the children are incrementally assigned ids ranging from 1 to $NUM\_THREADS - 1$.

Regarding work-sharing constructs, the following constructs are possible: (i) The loop construct, which ensures that the iterations of one or more associated loops are executed in parallel; (ii) The sections construct, which ensures that each section block is executed once by one of the threads of the current team. (iii) The single construct, which ensures that the associated block of code is executed exactly once by one of the threads of the current team. A work-sharing loop has logical iterations numbered from 0 to N-1, where N is the number of loop iterations, and the logical numbering denotes the sequence in which the iterations would be executed if the associated loop(s) were executed by a single thread. In order to determine how the iteration space should be distributed among the team, the scheduling clause must be specified. In any scheduling kind, the iteration space is divided into chunks of iterations. In almost every scheduling kind, the size of each chunk can be specified by the user (or a default value is used). The available scheduling clauses are (i) the run time, in which the decision regarding scheduling is deferred until run time; (ii)the auto, in which the decision is delegated to the compiler or run time system; (iii) the static, in which the iteration space is divided into chunks that are assigned to the threads of the team in a round-robin fashion (ordered by thread identification number); (iv) the dynamic, in which chunks are dynamically assigned to threads who have already finished executing a previous chunk; (v) the guided, in which the iterations of the loop are assigned to threads as in the dynamic scheduling, but instead of being static, the size of each chunk is dependent of the number of unassigned iterations.

2.2. Memory Model

In addition to its private memory, in OpenMP each thread of a team owns access to an address space which is shared within the whole team. Whenever private variables are created, private copies of a given variable are created on the private memory of each thread of the current team. Private variables can be created by the private, firstprivate and lastprivate clauses. In the first and last cases, non-initialized variables are created. In the second case, the variable is initialized by the value of the variable outside the parallel region. The lastprivate clause targets OpenMP loops and ensures the last iteration sets the value of the variable owned by the thread who created the team. Regarding shared variables, each thread is given access to the variable created outside the parallel construct.

2.3. Synchronization Model

Atomicity is not guaranteed by OpenMP when accessing and/or modifying shared memory. Consequently, the avoidance of data race conditions is left to the user. For this purpose OpenMP implements simple locks, nested locks and synchronization constructs. Unlike simple locks, nested locks can be locked multiple times by its owner. In both cases the owner can only unlock the lock a single time. The user is able to manage locks through Runtime Library Routines. Regarding synchronization constructs, the following are the most used: (i) the master construct, used to specify a block of code which will only be executed by the master thread; (ii) the critical construct, which estricts the execution of the associated block of code to a single thread at a time; (iii) the barrier construct, which ensures that the execution of the team is suspended until all threads of the team reach the barrier; (iv) the atomic construct, which ensures that a specific location is accessed atomically.

2.4. Compilation Process

The heterogeneity between the Instruction Set Architectures available today forces compilers to understand machine-independent source programs. Moreover, with the advent of high-level languages, compilers are usually required to interpret multiple languages. In order to fill those requirements, compilers is typically composed by a sequence of three stages, each of them with a specific function:

Front-End Verifies syntax and semantics of the given input program. This stage is responsible for generating an Intermediate Representation (IR) of the source code, which is processed by the Middle-End. The IR is common to all supported programming languages.

Middle-End Performs optimizations like removal of useless (or unreachable) code and discovery and propagation of constant values. Another IR is generated for the Back-End.

Back-End Generates assembly code for the target machine. Before the emission of the assembly code, register allocation and optimizations regarding the utilization of the hardware are performed. Usually the output output is then assembled and linked, generating executable code.

When OpenMP is supported, only the FE is affected: it identifies the OpenMP primitives and ensures the expected behavior by modifying and injecting IR code. Generally, parallel regions are exported to new functions (called microtasks) whose arguments are implementation-specific. However, the set of arguments generally include an array of pointers to the original variables of the variables accessed inside the parallel region. Depending on its data-sharing clause, each variable is made private or shared inside the microtask.

The work presented in the following sections works of top of Clang, an open-source compiler that already supports OpenMP in a branch of Clang 3.4. Clang is a FE for the C, C++, Objective C and Objective C++ programming languages which uses LLVM for the middle and back end stages.

3. Related Work

Most OpenMP solutions for heterogeneous embedded systems are architecture-specific (the solution presented on article [5], for instance) and/or propose extensions to the OpenMP API (as presented by the articles [3] and [1]). Although hiding implementation details, [6] emerges with an eye on generality. This chapter reports the state of the art in the scope of OpenMP solutions for heterogenous many-core embedded systems.

3.1. libEOMP: A Portable OpenMP Runtime Library Based on MCA APIs for Embedded Systems

In [6], OpenMP was extended to heterogeneous multicore embedded systems. To address the architectural challenges of heterogeneous systems, it was proposed a lightweight unified OpenMP runtime library, libEOMP, with MCA APIs as the target of OpenMP translation. MCA APIs was introduced by the Multicore Association (MCA). It supports device-level communication and resource management for multicore embedded systems. The MCA has built a set of APIs to standardize communication (MCAPI), resource management (MRAPI) and virtualization spanning cores on different chips. So, the OpenUH, a source-to-source OpenMP compiler, was used and its translation process works as follows: (i) The source code is parsed and translated into WHIRL IR, an intermediate representation, with OpenMP directives; (ii) An Inter Procedural analyzer step is performed; (iii) A Loop nest optimizer step is performed; (iv) Transformation of OpenMP, i.e., OpenMP directives are translated into WHIRL representing multithreaded code with OpenMP runtime library calls; (v) A Global scalar optimizer is performed. The generated code is then translated into C code (with the hand of a specific tool). The generated files are linked into an executable file.

Although presenting a solution for the problem they propose to solve, this work does not explore the resources available on most heterogeneous embedded systems for the benefit of OpenMP. In the case of barriers, the authors exploit resources with a centralized barrier algorithm (whose poor scalabilty is widely known).

3.2. Cyclops-64

Cyclops-64 (C64) is a supercomputer developed by IBM T.J. Watson Research Center, ETI Inc. in association with the University of Delaware. It is a distributed shared-memory many-core system composed by tens of thousands of C64 computing nodes arranged in a 3D-mesh Network, each C64 computing node comprising a C64 chip, an external DRAM and a small amount of external interface logic. A C64 chip is a set of eighty processors, each of them containing two Thread Units, one Floating Point Unit and two SRAM memory banks of 32KB each. All processors are connected to a crossbar network, so each of them is able to access to other processors on-chip memory as well as off-chip DRAM. An A-switch connects each C64 chip to its nearest neighbors in the 3D-mesh. Each five processors share an instruction cache of 32Kb. Since there are no data caches, a portion of the 32KB SRAM bank is configured as a scratchpad memory. The global shared-memory is built from the remaining space on the 32KB SRAM memory, uniformly addressable from each Thread Unit. With a scratchpad memory, the Thread Unit is provided with fast temporary storage.

The threading model, specially designed for C64 (called Tiny-Threads or TNT), is described in [2]. The model relies on non-preemptive tasks (i.e., after a software thread is assigned to an hardware thread unit, it will run there until completion) and a memory hierarchy that is fully visible by the programmer.

In TNT, a thread is activated for execution by binding an hardware thread unit to a thread activation pointer. Thread activation pointers are defined by a program pointer (address specified by the program counter associated with the corresponding hardware thread unit) and a state pointer (pointer to a TNT descriptor, where the thread specific information, including thread identifier and stack pointer, is stored). At boot time every hardware thread unit is given access to a scratch-pad memory. When a software thread is initialized and assigned to an hardware thread unit, it is given control over the corresponding scratch-pad memory (where its TNT descriptor is stored).

The TNT descriptor carries a status field indicator of the activity of the thread: inactive (period between the initialization of the thread and the assignment of a function to be run), available (or running) or idle. Idle threads are queued in a list so that they can be asked for generating a TNT thread. Each TNT descriptor is guarded by a lock, avoiding concurrent requests for services.

The efficiency of TNT threads was proven by comparing thread creation and termination in TNT threads with Pthreads. Results shown that thread creation is almost 350 times faster, while thread termination is between 580 and 860 times faster. Good scalability was also proven for applications that have enough parallelism and run well given the high intra-chip bandwidth.

Landing OpenMP on C64 implies a redefinition of descriptors, so that both physical thread units and OpenMP threads are supported. When a team of threads is created, some parameters are shared between the team (for instance, a function pointer and the arguments of the microtask). Originally, slave threads are given access to the masters descriptor. In C64, when the master thread polls workers to give them work, it copies the data into each worker descriptor. Although it increases the overhead associated to the initialization of the team, the number of references to the parent descriptor decreases significantly.

Although the good algorithms and their results, this work is focused on a specific architecture and on its properties as a distributed system.

3.3. Evaluating OpenMP support costs on MPSoCs

In [4], the costs associated with supporting OpenMP on MPSoCs (Multiprocessor Systems-on-Chip) are evaluated. The paper aims at proving that an efficient exploitation of the memory hierarchy is key to achieving a scalable implementation of the OpenMP constructs.

The evaluation is based on the hardware architecture whose number of processing elements is configurable (up to sixteen), being each of them linked to an interconnection network cross-bar bus. Synchronization mechanisms rely on hardware semaphores and all on-chip memory modules are mapped in the address space of the processors, globally visible within a single shared memory space. Accessing the local L2 memory of a different processing element is possible, but requires appropriate cache control actions.

The threading model is similar to Cyclops. At boot time, the executable image (composed by the specific program and some library) is loaded onto every processors local memory. After a common initialization step, slaves start a spinning task, waiting for useful work to do, while the master thread starts the execution of the application. When the master encounters a parallel region, it points slaves to the microtask and shared data. At the end of the parallel region, a global barrier synchronization step is performed and all slaves re-enter the spinning task. The spinning task is performed over a local buffer, stored in local L1 cache, which is modified by the master whenever a parallel region is found. The message sent by the master to the slave threads contains a task and frame pointers.

The authors introduced a new Master-Slave barrier algorithm, composed by a Gather and a Release phases. In the Gather phase, the master waits for a notification from every slave reporting their arrival to the barrier. In the Release phase, the master broadcasts a message passing-like signal. Every slave notifies its status on a separate polling flag that is allocated in local L1 SPM to reduce the polling traffic. To prove the performance of the Distributed algorithm, it was compared with a Centralized Barrier and a Master-Slave Centralized Barrier (with polling flags allocated in shared memory). It was proven that the Master-Slave algorithm scales better than the remaining algorithms. It was also proven the low impact of the Distributed Master-Slave barrier algorithm on real OpenMP program execution, in comparison to the Centralized and Master-Slave Centralized barrier algorithms.

Result of the fork-join model just described is a not significant interferent traffic on the interconnect. To synchronize threads within a parallel region, both atomic and critical constructs were implemented using hardware test-and-set semaphores.

This work also studies the allocation of shared variables. The placement of shared data and metadata is evaluated taking into account the following modes:

Mode 1 Default OpenMP placement. Slave processors access both shared data and metadata from the master core local L2 memory;

Mode 2 Since shared metadata is read-only, L1 SPM local to each core is exploited to host private replicas of metadata, while shared data is kept on the master core local L2 memory;

Mode 3 Shared variable allocation is redirected to shared L2 memory, while shared metadata is kept on the masters local L2 memory;

Mode 4 Metadata is accessed from local L1 and shared data from shared L2 memory;

Mode 5 Shared data is allowed to be placed on a cacheable region of the shared L2 memory, while metadata resides on the master core local L2;

Mode 6 Shared data is allowed to be placed on a cacheable region of the shared L2 memory, while metadata is replicated onto every L1 SPM.

In general the various allocation modes allow increasing degrees of improvement with relation to mode 1. For processor counts up to eight, it was proven mode 2 is on average faster than mode 3 and slightly slower than mode 4. For sixteen processors the behavior changes slightly, and in many cases mode 4 performs identical to modes 2 and 3.

4. Implementation

The system developed here follows an architecture where two systems are connected: the host and the guest systems (see figure 1). The host system is responsible for starting the execution of OpenMP programs, reason why it requires the assistance of an Operation System. The guest system is responsible for executing parallel regions and it does not need any Operating System support. The Instruction Set of the Guest System is not dependent on the Instruction Set of the Host system, which leads us to the heterogeneity of the whole system.

Throughout this chapter, both the parallel and the for constructs are discussed. This parallel clause suports the following clauses: (i) if; (ii) private; (iii) shared; (iv) default; (v) firstprivate; (vi) num_threads. On the other hand, the for construct supports the following clauses: (i) schedule; (ii) private; (iii) firstprivate; (iv) lastprivate; (v) shared; (vi) nowait. Additionally, the omp_get_num_threads, omp_set_num_threads and omp_get_thread_num Runtime OpenMP Libary Routines are explored.

Host System
Guest System

Figure 1: Basic hardware architecture

4.1. Execution Model

The designed model is based on the existence of host threads (threads executing on the host system) and guest threads (threads executing on the guest system). When an OpenMP program is started, a single host thread starts executing sequentially until a parallel construct is found. Unless an if clause with a false expression is specified, a team of guest threads is created and requested to execute a given parallel region. Since the host thread does not belong to the spawned team, one of the threads of the team is made Master by being assigned the thread id 0. While the team is executing, the host thread remains blocked until all threads have communicated the termination of its execution (implicit barrier). In order to create a team of threads, the host thread calls the CHTLib, who communicates the request of the user to the guest system. After the execution of the guest team, the host thread proceeds executing sequentially. Due to the heterogeneity between the host and the guest systems, parallel regions are encapsulated in microtasks, each of them emitted in a different binary file that is compiled for the Instruction Set Architecture of the Guest System (while the code to be executed by the host thread is compiled for the Instruction Set of the host system). By the time the CHTLib is called, it is imposed by the host thread to send the corresponding binary file to the new team.

As stated before, the default size of a team corresponds to the amount of processing units of the target device. In order to modify this number, both the num_threads clause and the omp_set_num threads_run time library routine were implemented. While in the first case the host thread informs the CHTLib about the size of the next team, the latter informs the CHTLib about the new default size. In order for threads to get the size of the team during execution, the calls to omp_get_thread_num are translated into the value of one of the arguments of the microtask.

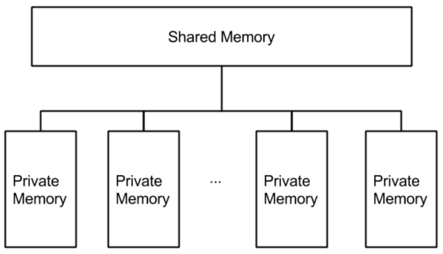

Figure 2: Memory Model

The sole difference between a generic for statement and an OpenMP for construct is that, in the second case, each thread of the current team is required to execute only a portion of the iteration space. The computation of the boundaries of each thread is dependent on the scheduling clause and chunk size specified by the user. For the static scheduling, the local boundaries are dependent on the global boundaries, the chunk size (which is calculated when it is not specified by the user) and the id of each thread. This OpenMP model allows all processing units to receive the exact same binary code, although each of them executes only a portion of the iteration space. In an initial phase, each thread executes exactly $\lfloor NIterations/NThreads \rfloor$ iterations. Before leaving the for construct, each thread verifies whether it is required to execute more iterations if the expression 1 is true (where LB refers to the current local upper bound, the TotalIts refers to the amount of the executed iterations and the GlobalUB refers to the global upper bound).

$$LB + TotalIts < GlobalUB \tag{1}$$

4.2. Memory Model

It is assumed here that the guest system is owner of a memory shared by all processing units and each processing unit is owner of a private memory that is not accessible by any other processing unit (see figure 2). Shared Memory is specially required for exchanging information within a team. In particular, OpenMP requires guest teams to synchronize (at barriers, for instance) and/or share variables. Specially for many-core guest systems, the existence of a coherent cache system improves performance significantly with faster accesses to shared memory. Without caches, the hardware overhead may increase due to the traffic generated by the many and simultaneous accesses to shared memory. Accesses to shared memory should then be avoided, if possible.

By the time the guest team starts executing, the locations of the variables are registered in an array that is provided to the guest threads via microtask argument. At the beginning of the microtask, the locations are stored in local variables that are accessed every time a shared variable is read or written. The increase in start up time due to the copy of the array to every private memory of the team is not significant when using message broadcast. Since the array is the same for every thread of the team, it could have been allocated in shared memory (thus saving much memory resources). However, this would only be beneficial for systems with caches shared between processing units. Both critical and atomic constructs use locks (byte-width variables allocated in shared memory) for the creation of critical regions. Since the probability of all atomic and/or critical constructs being accessed by most guest threads of the current team is high, the performance of the system could benefit from spatial locality in case the guest system is owner of a coherent cache system. For that reason, and because the memory occupied by locks is not much, all locks used by atomic/critical constructs are allocated in a contiguous memory space.

The allocation of barriers is done the moment before the execution of the parallel region starts, while the release is done the moment after. The allocation of barriers is dependent on the existence of team synchronizations within the parallel region. The address returned by the allocation of the barrier is sent to the team via microtask argument. Since all barrier synchronizations within the same parallel region work on the same barrier resource, there is no need to allocate multiple barriers. By the time the host thread notices end of the execution of the team, all shared resources are freed from memory (with some exceptions like barriers). The value of shared variables is copied back to the hosts memory, so that the sequential execution proceeds with up-to-date values.

The private memory of each processing unit is divided into the following sub-address spaces (as shown by figure 3): (i) Program Memory, where the code being executed is stored; (ii) Thread Id, where the id of each thread is stored; (iii) Arguments, where the arguments of the microtask are stored; (iv) Stack, which occupies the remaining space. An alternative to save a specific space for the storage of the ids would be the inclusion of the thread ids in the arguments of the microtask. However, this would preclude the broadcast of the arguments of the microtask to the guest system. As such, the arguments are able to be broadcasted, thus decreasing the hardware overhead. Although the arguments own a specific location, they are pulled to stack at when the threads start executing the parallel regions. Afterwards, the space occupied by the arguments is able to be overriden. The remaining spaces are not. Every time a given thread asks for its own thread identification number, the value stored in the

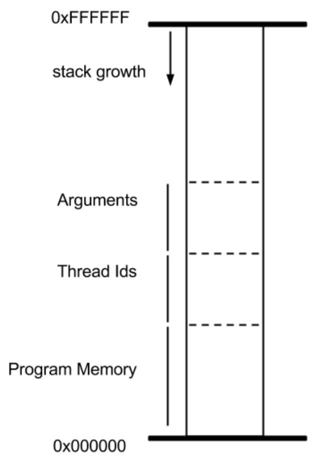

Figure 3: Private Address Space

specific location is loaded.

4.3. Synchronization Model

The current synchronization model explores two types of synchronization: synchronization of accesses to shared resources and synchronization between the threads of a given team. It is ensured each synchronization construct has exclusive access to a lock. Every time a critical or atomic construct is encountered by a given thread, it tries to acquire the specific lock by setting its value from 0 to 1, using test-and-set instructions. By the time it succeeds acquiring the lock, the critical region is executed and the lock is released. Every time a lock is released, an unconditional store instruction is performed, changing the state of the lock from 1 to 0. Waiting threads spin on the lock, trying to acquire it. This algorithm is known by simple test-and-set spin lock and it is the most basic algorithm from the following list: (i) Simple test-and-set locks; (ii) Ticket locks, where every time a thread fails acquiring a given lock, it asks for a ticket and waits for its turn; (iii) Array-based queuing locks, where each process uses a fetch_and_increment instructions to obtain the address on which to spin (i.e., the next array element whose content changes when its turn arrives); (iv) MCS locks, where locally accessible lock structures are created, in which the thread spins until being notified about its success acquiring the lock. Although it provides the worst results regarding hardware overhead, it is actually the most suitable for our guest system. Both ticket and array-based queuing locks require a fetch_and_increment instruction which is not found in some Instruction Set Architectures. MCS locks, on the other hand, require the processing units to have access to each others private memories. The cost of choosing simple test-and-set locks is particularly high in case of guest systems without coherent caches due to the traffic generated by the waiting threads.

Regarding team synchronization, at barriers, the tournament algorithm was selected from the following list: (i) Centralized barriers, in which each thread atomically updates shared state to indicate its arrival to the barrier; (ii) Software Combining-Tree Barrier, in which tree of centralized barriers is created; (iii) Dissemination Barrier, in which all threads synchronize with each other through pairwise synchronizations; (iv) Tournament Barrier, in which a tournament between the threads is created. On each round the winner thread (which is statically determined) ensures the losers have already arrived to the barrier, before proceeding to the next round. Whenever a thread loses a round, it exits the tournament and waits for the winner of the tournament to announce the end of the tournament. At this moment, the whole team is ensured that all threads have already arrived to the barrier; (v) New Tree-based Barrier, in threads are organized within a tree through which they communicated with each other the arrival to the tree. Besides the fact that both the centralized and the combining tree barriers require fetch and increment instructions, they cause an unsustainable network traffic on systems without caches. Since there is no direct communication between processing units, the tree-based barrier is also not suitable because the processing units can not spin locally. Between the dissemination and the tournament barriers, the later was selected due to the fewer network traffic it generates.

4.4. Software Architecture

Figure 4 represents a data flow diagram where the whole communication process is demystified. In addition to the Clang compiler, the software architecture in which the developed model relies on includes two more libraries:

CHTLib Called in run time, by the host thread, in order for it to communicate with the guest system. Depending on the requests of the host, the CHTLib manages the contents owned by the guest system.

Device Driver Responsible for resource management on the guest system. More specifically, the driver is responsible for allocating and releasing memory resources, writing and reading contents to guest memory and job scheduling.

The Device Drivers structure is composed by a low and an high levels (see figure 5). While the first is responsible for the direct communication with the device (or guest system), the second processes the requests from the user (the CHTLib) and manages the devices resources according to the request. To be more precise, the low level layer supports

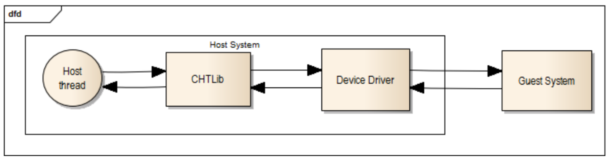

Figure 4: Run time communication process

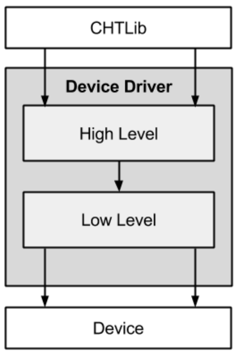

Figure 5: Device Driver structure

memory transfers between the hosts and the devices (shared or private) memories, the execution of thread blocks and the waiting for the termination of the execution of thread blocks. The whole management work is left to the high level layer. One of the resources managed by it is the shared memory, in which the Buddy algorithm is used. With the buddy memory allocation technique, the memory is divided into partitions (to be more precise, it splits memory into halves) to try to give the best-fit. A split block can only be merged with its unique buddy block, which then reforms the larger block they were split from. The advantages of this algorithm relate to its simple implementation and fast memory allocation and deallocation. However, some space is wasted in internal fragmentation since all requests are rounded up to a power of two.

Regarding the CHTLib, it performs the following actions in order for a parallel region to be executed (through the order specified): (i) Allocation of space for each shared variable; (ii) Allocation of space for the locks used by critical and atomic regions; (iii) load of the binary code and the arguments of the microtask in private memories; (iv) Execution of the parallel regions; (v) Implicit barrier (vi) Copy back the values of the shared variables from the shared memory of the guest system to the hosts memory; (vii) Release of shared resources. The actions just described are ordered to the Device Driver, who is able to manage and establish contact with the guest system. Although simple, this library plays an important role since it allows the host thread to be freed from the concerns related to the management of the guests resources.

5. Evaluation

5.1. Experimental Platform

The present chapter presents some case studies and discusses their applicability on the presented model. The model is applied here to an architecture composed by an an Intel Core i7 3770K @ 3.5GHz (host computer) and a Field Programmable Gate Array (Virtex-7). The host computer is composed by an Application Layer and a Device Driver, whereas the accelerator is composed by an Internal Control Memory (that decodes the instructions to be executed by the clusters), a Controller (to select clusters) and a Cluster Interconnection Network (communication bridge between the controller and the clusters). In order for the host computer to communicate with the Accelerator, a PCI Express is used, whose PCI endpoint on the host device is the Device Driver mentioned above. The guest system ensures the access to an external memory, shared between all processing clusters, through the Cluster Interconnect Network. Regarding the processing clusters, each one is composed by a set of cores, a Network Interface (that selects the cores to enable), a Core Interconnection Network (to establish communication between the Network Interface and the cores). Each core is composed by another Network Interface, a Processing Element, a Core Controller and a Core local memory.

5.2. Case studies

The case studies included here are:

Prime Numbers Calculates the prime numbers from 1 up to N. With it, both the parallel and for constructs is tested, besides the shared, private and firstprivate data-sharing clauses.

Factorial Calculates the factorial of N. Besides testing the parallel and for constructs, this algorithm tests the synchronization constructs. Since both the atomic and the critical constructs are implemented in the exact same way, only one of them is tested. One of the main purposes here is to understand the effect of the lack of cache systems on the model.

N Consecutive Barriers Although there is an implicit barrier at the end of each work-sharing construct, the effect of the variation of the number of barriers is not seen on the case-studies above. This relation between the number of barriers, the number of threads executing the given block and the overhead is associated with the barrier execution is taken from the variation of the number of explicit barriers executed within a simple parallel construct (no for construct required).

The execution of the case studies on the suggested experimental platform, using the model described throughout the present document, shown that the model excels mostly in portability. The implementation of the model on purposed hardware is straightforward since it does not dependent on the guest system. The fact that the binary files containing the parallel code are not dependent on the binary file which is executed on the host system reflects on the portability of the model too. As a result, the difference between Instruction Sets is not an issue. A third cause of the portability of the model is the choice of algorithms whose requirements are not too high and owned by most hardware architectures. However, the scalability of the model is highly dependent on the hardware architecture of the guest system. In case a coherent cache system is not included, a significant amount of traffic is generated by simultaneous accesses to shared memory. In case a coherent cache system is included, the model achieves good scalability, taking advantage, in some cases, of both the temporal and spacial principles to reduce the number of real accesses to shared memory.

5.3. Results

The results of this work show the overhead associated with each construct in function of the growth of the size of the team. Each test is the result of inserting the respective construct in a parallel region without any additional statements. In the overhead associated to an empty parallel construct there is not a significant variation in function to the size of the team (see table 1). Given that the accelerator in utilization does not include any cache system, the results shown here reflect the worse case due to the increased traffic generated.

1	8	16	40	60
0.363	0.293	0.334	0.346	0.315

Table 1: Overhead associated with the parallel construct in function to the size of the team

Table 2 shows that the overhead associated with the barrier construct is higher for bigger teams, as it was expected. Regarding the influence of the amount of barriers, within the same parallel region, for the overhead associated to each barrier, it is noticeable that the first barrier has the higher overhead associated, whereas the following are practically instantaneous. This behavior is due to the fact that all barrier synchronizations use the same barrier, whose allocation causes most of the overhead generated (and not the synchronization itself).

Regarding the overhead associated with the critical and atomic constructs (since they are both implemented the exact same way, they produce the exact same overhead), the growth of the overhead proved not to be significantly influenced by the growth of the size of the team, but influenced by the number of critical/atomic constructs within the same parallel region. This is due the fact that each critical/atomic construct uses a specific lock, so the time wasted on the allocation of the locks increases with the growth of the amount of critical/atomic constructs. Since the critical regions used during these tests were empty, the traffic generated is not much and consequently the influence of the number of threads in the team is not noticeable on the results shown in table 3.

	1	8	16	40	60
1	0.377	1.476	2.852	9.065	13.866
2	0.36	1.389	2.688	9.732	13.047
5	0.351	1.498	2.905	9.735	11.939
10	0.386	1.431	3.107	8.716	12.971
25	0.346	1.77	2.735	8.859	13.668
40	0.385	1.565	2.722	9.288	14.016

Table 2: Overhead associated with the barrier construct. The rows show the variation of the number of consecutive barriers, whereas the columns show the variation of the size of the team.

	1	8	16	40	60
1	0.565	0.546	0.516	0.542	0.652
2	0.447	0.645	0.59	0.565	0.68
5	0.797	0.7	0.633	0.807	0.744
10	0.777	0.818	0.846	0.946	0.954
20	2.088	2.187	2.392	2.682	2.521

Table 3: Overhead associated with the critical or atomic constructs. The rows show the variation of the number of consecutive critical/atomic constructs, whereas the columns show the variation of the size of the team.

5.4. Discussion

The execution of the case studies on the suggested experimental platform, using the model described throughout in present document, shown that the model excels mostly in portability. The implementation of the model on purposed hardware is straightforward since it does not dependent on the guest system. The fact that the binary files containing the parallel code are not dependent on the binary file which is executed on the host system reflects on the portability of the model too. As a result, the difference between Instruction Sets is not an issue. A third cause of the portability of the model is the choice of algorithms whose requirements are not too high and owned by most hardware architectures.

Regarding scalability and performance, the model is dependent on the hardware architecture of the guest system. In case a coherent cache system is not included, a significant amount of traffic is generated by simultaneous accesses to shared memory. In case a coherent cache system is included, the model achieves good scalability, taking advantage, in some cases, of both the temporal and spacial principles to reduce the number of real accesses to shared memory. The barrier construct is specially affected by the lack of a cache system, although we were able to infer that the size of the team does not influence significantly the overhead associated to the synchronization of the team.

6. Conclusions

This dissertation presented a model that supports the development of OpenMP for heterogeneous embedded systems. Its execution is divided into phases, each one of them with a specific concern. The first phase targets compilation process of the input file, where the parallel regions are outlined to new functions, which are emitted in new binary files. In order to support the heterogeneity of the systems, the generation of the binary files targeting the embedded system is independent of the generation of the binary files targeting the host system. The remaining phases are executed in run time. By the time the program is started, an host thread starts executing sequentially. Whenever a parallel construct is encountered, a team of threads is created on the guest system. While the generated guest threads do not terminate its execution, the host thread enters an barrier. When the barrier is overcome, the host thread proceeds executing sequentially. During the execution of the guest threads, some work-sharing and synchronization mechanisms are allowed. The performance of the model is highly dependent on the hardware architecture of the guest system. The model tries to reduce the hardware overhead as much as possible (through broadcast of messages when possible, for instance), although some restrictions (like the absence of a coherence cache system) are not possible to workaround. Overall, the goals were met.

6.1. Future work

For future work, this work would include the support for nested parallelism. The problem with nested parallelism relates with the fact that the host thread has to be informed about a management matter during the execution of the guest team. The problem would be solved by separating the parallel region into three parallel regions: the first encompassing the code before, the nested region, the second encompassing the nested region and the third encompassing the remaining code. Each of the generated microtasks would have to be carefully informed about the context from which the given region was expanded (more precisely, about what variables are in execution and what are its values). This situation could be seen as a tree of parallel regions, whose context of each region is dependent on the context of its parent.

Another point to be included for future work is the support for the dynamic scheduling on work-sharing constructs. The current model is already prepared for this scheduling, since it is possible to re-execute a given kernel on a given core from the point of view of the Device Driver.

Acknowledgements

The author would like to thank professors David Matos, Nuno Roma and Pedro Tomas for the good guidance throughout the development of this work. I would also like to thank Nuno Neves, for creating the embedded system where the product of this work was tested on, and Pedro Moreira, for helping me in the development of the embedded system support driver.

References

[1] E. Ayguade, R. M. Badia, P. Bellens, D. Cabrera, A. Duran, R. Ferrer, M. Gonzàlez, F. Igual, D. Jiménez-González, J. Labarda, L. Martinell, X. Martorell, R. Mayo, J. M. Pérez, J. Planas, and E. S. Quintana-Ortí. Extending OpenMP to Survive the Heterogeneous Multi-Core Era. *International Journal of Parallel Programming*, 38(5-6):440–459, 2010.

[2] J. del Cuvillo, W. Zhu, Z. Hu, and G. R. Gao. Tiny Threads: a Thread Virtual Machine for the Cyclops64 Cellular Architecture. In *Parallel and Distributed Processing Symposium, 2005. Proceedings. 19th IEEE International*, 2005.

[3] F. Liu and V. Chaudhary. Extending OpenMP for Heterogeneous Chip Multiprocessors. *Proceedings 2003 International Conference on Parallel Processing*, pages 161–168, 2003.

[4] A. Marongiu, P. Burgio, and L. Benini. Evaluating OpenMP Support Costs on MPSoCs. In *DSD'10*, pages 191–198, 2010.

[5] K. O'Brien, K. O. Brien, and Z. Sura. Supporting OpenMP in Cell. *International Journal of Parallel Programming*, 36(3):289–311, 2008.

[6] C. Wang, S. Chandrasekaran, B. Chapman, and J. Holt. libEOMP: A Portable OpenMP Runtime Library Based on MCA APIs for Embedded Systems. In *PMAM'13*, pages 83–92, 2013.