Implementation of OpenMP on Application-Specific Architectures

Cláudia Filipa Rosa Henriques

Thesis to obtain the Master of Science Degree in

Information Systems and Computer Engineering

Supervisors:  
Doctor David Manuel Martins de Matos  
Doctor Nuno Filipe Valentim Roma

Examination Committee

Chairperson:  
Doctor Joaquim Armando Pires Jorge

Supervisor:  
Doctor David Manuel Martins de Matos

Member of Committee:  
Doctor José Carlos Campos Costa

November 2014
I would like to thank professors David Matos, Nuno Roma and Pedro Tomás for the good guidance throughout the development of this thesis. I would also like to thank Nuno Neves, for creating the embedded system where the product of this thesis was tested on, and Pedro Moreira, for helping me in the development of the embedded system support driver.

To my family and friends, I would like to thank their unconditional support during the past five years. Without you all, this journey would not be possible!
This thesis is dedicated to my sister Catarina Henriques and to my friend Marta Albardeiro.

Thank you!
Resumo

Atualmente, os sistemas embebidos heterogéneos estão presentes em qualquer lugar e as suas capacidades de processamento são cada vez maiores. Essa capacidade de processamento é resultado do constante desenvolvimento que temos vindo a presenciar no mundo computacional. Visto o desenvolvimento de programas paralelos para este tipo de sistemas ser difícil e requerer demasiado tempo, essas capacidades de processamento nem sempre são aproveitadas da melhor forma. Em sistemas ditos general purpose já existem ferramentas bastante fiáveis, como é o caso do OpenMP, que ultrapassam este problema, automatizando o processo de paralelização de um programa.

Nesta tese é proposto um modelo que suporta OpenMP em arquitecturas embebidas heterogéneas. Nesta área existe pouco trabalho feito, sendo que o trabalho disponível actualmente está mais relacionado com arquitecturas distribuídas.

O modelo desenvolvido passa pela geração do código paralelo em binários independentes (gerado conforme o conjunto de instruções suportado pelo sistema embebido), um por região paralela, que são copiados para as memórias das unidades de processamento para que possam ser executados. Todo o trabalho de gestão de recursos é feito por um sistema general purpose, onde os programas OpenMP são lançados. Aí, uma tarefa começa a executar os programas de forma sequencial até que uma região paralela seja alcançada.

As maiores características alcançadas pelo modelo desenvolvido prendem-se com a sua elevada portabilidade. A escalabilidade e performance do modelo também são caracterísitcas, apesar de estarem dependentes da arquitectura do sistema embebido.
Currently, the 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. Since the development of parallel programs for these systems is difficult and require too much time, these capacities are, generally, not properly used. For general purpose systems, like the computer we have at home, there are reliable tools, like OpenMP, that overcome these problems by automating the parallelization process of a given code.

For this thesis it is purposed a model that supports OpenMP for heterogeneous embedded architectures. In this area, there is few work done, being most of it related to distributed architectures.

The developed model generates the parallel code in independent binary files (generated depending on the Instruction Set supported by the embedded system), one per parallel region, that are copied to the private memories of each processing unit so that they could be executed. All the management work is done by a general purpose system, where every OpenMP program is started. A thread starts executing sequentially there until a parallel region is reached.

The major characteristics of the developed model relate mostly to its high portability. The scalability and performance of the model are also high, but they are restricted by the hardware architecture of the embedded system.
**Palavras Chave**

OpenMP

Sistemas Embebidos Heterogéneos

Paralelização

**Keywords**

OpenMP

Heterogeneous embedded systems

Parallelization
# Index

1 **Introduction**  
   1.1 Motivation ........................................... 2  
   1.2 Thesis Goals ......................................... 2  
   1.3 Document Organization ................................ 3  
   1.4 Summary .............................................. 3  

2 **Context**  
   2.1 Execution Model ....................................... 6  
      2.1.1 Parallel Construct ................................ 7  
      2.1.2 Work-sharing Constructs .......................... 9  
   2.2 Memory Model ......................................... 11  
   2.3 Synchronization Model .................................. 13  
   2.4 Compilation Process ................................... 14  
   2.5 Summary .............................................. 16  

3 **Related Work**  
   3.1 libEOMP: A Portable OpenMP Runtime Library Based on MCA APIs for Embedded Systems ................................. 19  
   3.2 Cyclops-64 ............................................. 22  
   3.3 Evaluating OpenMP support costs on MPSoCs .................. 26
## List of Figures

2.1 UML Activity Diagram of an application parallelized with OpenMP . . . . . . . 7
2.2 Nested parallelism . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 9
2.3 Structure of a classic compiler . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 14
2.4 Clang+LLVM compilation process . . . . . . . . . . . . . . . . . . . . . . . . . . . 16

3.1 Embedded OpenMP solution stack . . . . . . . . . . . . . . . . . . . . . . . . . . . 20
3.2 OpenUH overview . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 21
3.3 Cross Compilation process . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 22
3.4 Cyclops 64 architecture . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 23
3.5 C64 chip architecture . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 23
3.6 Scratchpad Memory with TNT descriptor . . . . . . . . . . . . . . . . . . . . . . . 24
3.7 MPSoC architecture . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 27
3.8 MCS lock structure . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 30
3.9 MCS lock execution example . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 31
3.10 Software Combining Tree initialized . . . . . . . . . . . . . . . . . . . . . . . . . . 32
3.11 Dissemination Barrier example . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 33
3.12 Tree-based barrier’s node . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 34

4.1 Basic hardware architecture . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 37
4.2 Binary files generated when compiling a C source file . . . . . . . . . . . . . . . 40
List of Tables

4.1 Supported constructs and clauses ........................................ 38

5.1 Overhead associated with the parallel construct ....................... 64

5.2 Overhead associated with the barrier construct ....................... 64

5.3 Overhead associated with the critical or atomic constructs .......... 65
Open Multi-Processing (widely known as OpenMP) is a portable and scalable programming 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 become a de-facto standard in the parallel programming scope and it is supported by many compilers and processor architectures.

The OpenMP Architecture Review Board (ARB), a non-profit corporation responsible for overseeing the OpenMP’s specification and producing/approving new versions of it, published its first OpenMP specification (1.0) for Fortran in October 1997. Until July 2011, versions ranging from 1.0 to 3.1 appeared, where support for new languages was included (C and C++) and new concepts were introduced (the concept of task and task construct, for instance). In July 2014 the specification of version 4.0 of OpenMP was released, where old constructs were improved and new ones were added. Currently, the OpenMP core elements are parallel control structures (who govern the control flow in the program), work-sharing (who distribute work among threads), data environment (who give a scope to the variables), synchronization (who coordinate thread execution) and run time functions/ environment variables (who provided/ask for indications to the parallel control structures).

Although the well-known growth of heterogeneous embedded systems over the past few decades, it is fact they still lack solutions for easily programming parallel embedded applications, thus making parallel programming an exhausting, erroneous and time-consuming task. This thesis 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 present introductory chapter describes both motivations and goals of the thesis, as well as the organization of this document.

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 thesis 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).

1.2 Thesis Goals

The product of the current thesis is intended to meet the following goals:

- High portability, providing a model for heterogeneous systems;
- High performance, maintaining good speedups when comparing to other OpenMP implementations;
• High scalability, allowing parallel algorithms to grow in number of threads with low overhead.

1.3 Document Organization

Chapter 2 contextualizes the present dissertation. It includes an overview of the OpenMP API and a description of general compilation processes (with and without OpenMP).

Chapter 3 reviews the related work, as it describes the available approaches for parallel programming on heterogeneous embedded systems. It also reviews solutions for some of the problems implicit in the implementation of the OpenMP model in a system with no Operating System (e.g: synchronization algorithms).

Chapter 4 describes the model which was developed during this thesis. The following topics are covered here:

• Execution model;

• Memory model;

• Synchronization model;

• Software architecture;

Chapter 5 evaluates the work. It starts by presenting the experimental platform which was used to test the developed model and three use cases are presented. Afterwards, testing results are shown and discussed.

Chapter 6 concludes the thesis and describes the future work.

1.4 Summary

This chapter introduces the scope of the dissertation: OpenMP on heterogeneous embedded systems. OpenMP is a portable, simple and flexible programming model for developing paral-
lel applications, whose core elements are: parallel control structures, data environment structures, work-sharing structures and run time function/environment variables. Using OpenMP, the user is given the opportunity to have coarse or fine grained control over parallelism.

The motivations of the work relate to the lack of programming models for easily develop parallel applications for heterogeneous embedded systems, although its strong growth over the past few decades. At the end of this thesis, it is supposed to have been reached a portable, scalable and high performance model.

Chapter 2 contextualizes the present dissertation (that is, the reader is introduced to the basic concepts of the thesis). Chapter 3 describes the work related to the subject of the thesis. Before the evaluation of the work, which is done by chapter 5, the solution of the problem is presented by chapter 4. Chapter 6 concludes the thesis and describes future work.
An advantage of OpenMP is the unification of both serial and parallel applications: although OpenMP code is required to be compiled by specific compilers (with OpenMP support), whenever OpenMP is not supported (or enabled), compilers treat OpenMP constructs as comments, thus producing serial code. This is an outcome of the simplicity and flexibility of the OpenMP API and it leads us to the fact that, generally, serial code does not need to be much modified in order to be converted into parallel code. Consequently, the chance of introducing bugs is reduced and, since there are times that the overhead associated with resource management causes the parallel code to be slower than the serial one (for instance, when the parallel work is not much or when there are hardware architectural obstructions), the cost of testing whether or not the code should be parallelized is typically irrelevant. Despite the fact that OpenMP does not include a reliable error handling system, it minimizes the problem by allowing incremental parallelism (that is, the developer is able to work on one part of the program at a time). However, debugging synchronization bugs and race conditions is still difficult.

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 the 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 thesis. It is focused on the description of the OpenMP programming model (and the behaviors behind each construct) from user perspective, including the execution, memory and synchronization models of OpenMP 4.0 (since the present work targets C language, Fortan-specific constructs are not be covered here). The classical compilation process of OpenMP programs is also discussed at the end of the chapter.
CHAPTER 2. CONTEXT

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. With OpenMP constructs (whose syntax is shown below) the user is capable of specifying its intents regarding the parallelization of the code.

```plaintext
#pragma omp <construct-name> [clause[, clause] …] newline
```

OpenMP constructs are divided into four categories: parallel constructs, work-sharing constructs, tasking constructs and SIMD constructs. Parallel constructs are responsible for creating teams of threads (so the remaining constructs have to be preceded at least by one parallel construct), while the work-sharing constructs distribute work by the threads available on the team. Since this thesis is focused on those constructs, they are described in more detail in sections 2.1.1 and 2.1.2, respectively.

Tasking constructs are used to explicitly define tasks to be executed in parallel with the code outside the task region. When a thread encounters a task construct, it may choose to execute the task immediately or defer its execution until a later time. If the task execution is deferred, then the task is placed in a conceptual pool of tasks that is associated with the current parallel region. The threads in the current team will ask tasks out of the pool and execute them until the pool is empty (so, unless the user specifies a tied clause, the thread that executes a task may not be the thread who originally encountered it). Threads are allowed to suspend the execution of a task region, when task scheduling points are found, and resume its execution later (in case their execution is not completed). If the task is tied to the thread who has found it, its execution can not be resumed by any other thread of the team.

SIMD constructs provide support for Single Instruction, Multiple Data operations. The body of SIMD constructs is always a loop whose iterations are executed by means of SIMD instructions. A variant of the SIMD construct is the loop SIMD construct, where a loop is executed concurrently, using SIMD instructions, and in parallel, with the distribution of the iterations among the team.
To illustrate the execution of an OpenMP program, figure 2.1 presents the UML Activity Diagram of a simple OpenMP program (whose code is shown alongside the diagram). The code of the program uses bold font to highlight actions, black font to highlight serial code and red font to highlight parallel code. The program consists on the print of the message “First Print”, followed by a parallel print of the message “Second Print”. After the end of execution of the parallel region, a third print is executed with the message “Third Print”.

```
#include <stdio.h>

int main()
{
    printf("First Print\n");
    #pragma omp parallel
    {
        printf("Second Print\n");
    }
    printf("Third Print\n");
    return 0;
}
```

Figure 2.1: UML Activity Diagram of an application parallelized with OpenMP.

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. So, now we truly understand the code and the UML Activity Diagram shown in figure 2.1: the initial thread executes the first print message and then it creates a team of threads to execute the second print message; when the initial thread is certain that the whole team has already finished the execution of the parallel region, the sequential execution is resumed by it and the third print message is executed.

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 \( NUM\_THREADS - 1 \).

Parallel regions do not have to be created by the initial thread. Actually, any thread can start a team in case it finds a parallel construct. Following the reasoning above, the thread who encounters the parallel construct is the master of the team it spawns (so it is assigned the thread id 0). The creation of teams by threads in the context of other teams is called nested parallelism. Figure 2.2 depicts nested parallelism in a tree whose root represents the initial thread and all white nodes represent threads in the context of a given team. Each node is labeled with id of the thread. Bold arrows represent the work-flow of the master thread, whereas simple arrows represent thread creation. The reader should notice that the master is always assigned the thread id 0, independently of its previous context.

It is provided, by the OpenMP API, a set of possibilities regarding thread affinity. When the environment variable OMP_PROC_BIND is set to false, OpenMP threads are allowed to be moved, whereas if set to true, OpenMP threads are not allowed to be moved. Alternatively, a comma separated list with the values MASTER, CLOSE and SPREAD can be used to specify
2.1. EXECUTION MODEL

the thread affinity policy for the corresponding nesting level. With the MASTER, the worker threads are kept in the same partition as the master thread. With CLOSE, those are kept close to the master thread in contiguous place partitions. And with SPREAD a sparse distribution across the place partitions is used. If undefined, OMP_PROC_BIND defaults to TRUE when OMP_PLACES is set and false otherwise. With OMP_PLACES, it is possible to specify on which CPUs the threads should be placed.

2.1.2 Work-sharing Constructs

As stated before, work-sharing constructs distribute the execution of the associated body among the members of the team that encounters it. Each thread executes a portion of the region in the context of its implicit task. Work-sharing regions have no barrier on entry but, unless a nowait clause is specified, they do have an implicit barrier at the end of the work-sharing region for synchronizing the team. The main work-sharing constructs are:

Loop Construct The loop construct ensures the iterations of one or more associated loops are executed in parallel. Every loop construct is bind to a parallel construct which creates the team of threads by whom the iterations of the loop are distributed. By default, only one loop is associated with the loop construct, although this value is changeable by the collapse clause. All associated loops must be perfectly nested, that is, there must be no intervening code nor any OpenMP directive between any two loops. 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. The values of the loop control expressions of the associated loops must be the same for all the threads in the team.

The scheduling clause determines how the iteration space should be distributed among the team. 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). Unless an ordered clause is specified, the iterations are executed in an unspecified order. Below, the available scheduling kinds are listed and described.

RUNTIME The decision regarding scheduling is deferred until run time.

AUTO The decision regarding scheduling is delegated to the compiler or run time system.

STATIC 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). When no chunk size is specified, the iteration space is divided into chunks of the same size (with the exception of the last chunk, which may have fewer iterations). The default size of each chunk is given by equation 2.1.

\[
\left\lfloor \frac{\text{#Iterations}}{\text{#Threads}} \right\rfloor
\]

(2.1)

DYNAMIC The iteration space of the loop is divided into chunks and each thread is assigned one chunk. When a thread ends up executing, it requests another chunk. This behavior lasts until there are no chunks to be executed. The size of each chunk is determined by the user, except for the last chunk. When no chunk size is specified, each chunk is composed by one iteration.
2.2. MEMORY MODEL

When the guided scheduling is requested, the iterations of the loop are assigned to threads as in the dynamic scheduling. The chunk size is determined every time a thread requests for a chunk and it is given by the equation 2.2. The chunk size is equivalent to the division \( D \) between the number of unassigned iterations and the number of threads of the current team. The size of the chunk is \( K \) (that is, the chunk size specified by the user or 1 when it is not specified) when the floor of \( D \) is greater than \( K \).

\[
\left\lfloor \frac{\#\text{UnassignedIterations}}{\#\text{Threads}} \right\rfloor
\]  

(2.2)

Sections Construct The sections construct is non-iterative and is composed by a set of section blocks. It is ensured each section is executed once by one of the threads of the current team.

Single Construct The single construct ensures that the associated block of code is executed exactly once by one of the threads of the current team.

2.2 Memory Model

The OpenMP API provides a relaxed-consistency shared-memory model. In addition to having access to a memory shared by the whole team, each thread is allowed to own a temporary view of memory. It is not a required part of the OpenMP memory model but it can be represented by any intervening structure between the thread and the memory (caches, registers or other local storage). Each thread has also access to a memory which can not be accessed by another thread, called threadprivate memory.

When a thread accepts data-sharing clauses, it is accepting the existence of two kinds of variables: shared and private. All threads of the team are thus given access to the original value of every variable, shared or private, accessed inside the body of the construct. Private variables consist on private copies of a given variable to the private memory of the threads 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, it is created a variable which is initialized by the value of the original variable. The lastprivate clause targets OpenMP loops and ensures the last iteration sets the value of the original variable. Shared variables make direct accesses to the original variables. Whenever a thread performs a write operation on a shared variable, the new value is stored in the temporary view of the thread, if existent. Flush operations are responsible for storing the contents of the temporary views in shared memory and synchronize them all. If a temporary view has several values for the same variable, only the last one is considered. Besides the explicit requests, flush operations are implied in the following cases:

- barrier;
- at entry to and exit from critical or parallel constructs;
- at entry to and exit from constructs with ordered clause;
- at exit from for, sections or single constructs;

In order to replicate a given list of variables by threads, the threadprivate directive must be specified. It guarantees the initial thread’s copy of the variable persists between any two consecutive references to the variable in the program. For non-initial threads, the thread’s copy is guaranteed to persist between two consecutive active regions if:

- Neither parallel region is nested inside another explicit parallel region;
- Both parallel regions create the same number of threads;
- The thread affinity policies used to execute both parallel regions are the same;
- It is ensured, by the user or the compiler, that the dynamic adjustment of the number of threads is disabled.

The content of a threadprivate variable is indeterminate. Therefore, the list of variables need to be initialized before use. The most common way to initialize threadprivate variables
on entry to a parallel region is by using the copyin clause, which copies the value stored in
the master’s thread copy of the threadprivate variable to the copies of all others of the current
team. A copyprivate clause (only associated to the single construct, which is explained later)
can also be useful for the initialization of the copies since it broadcasts a value from the data
environment of one implicit task to the data environments of the other implicit tasks belonging
to the parallel region.

2.3 Synchronization Model

Atomicity is not guaranteed by OpenMP when accessing and/or modifying shared mem-
ory. 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.

The available synchronization constructs are:

Master Used to specify a block of code which will only be executed by the master thread.

Critical Restricts the execution of the associated block of code to a single thread at a time.

Barrier Specifies an explicit barrier. It is ensured that the execution of the team is suspended
until all threads of team reach the barrier.

Taskwait Specifies a wait on the completion of a child task of the current task.

Taskgroup Specifies a wait on completion of the child tasks of the current task and their de-
scendant tasks.

Atomic Ensures the specific location is accessed atomically.

Flush Executes the OpenMP flush operation, that is, all threads have the same view of memory
for all shared objects.
**Ordered** Specifies a structured block in a loop region that is supposed to be executed in the order of the loop iterations.

### 2.4 Compilation Process

In order for a processing unit to execute a given program it has to be given the object code of the specific program. Object code is machine-dependent and it is the result of a 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. Figure 2.3 shows the structure of a classic compiler.

![Figure 2.3: Structure of a classic compiler](image)

A compiler is typically composed by a sequence of three stages, each of them with a specific function:

**Front-End (FE)** 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 (ME)** 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 (BE)** 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. Recall the code given in figure 2.1. It is shown below part of the assembly code generated by GCC 4.8.1 for a 64-bit Intel machine. The reader is able to see a simple program which is started by an initial thread. It executes sequential code, until GOMP_parallel_start is called. Between the GOMP_parallel_start and the GOMP_parallel_end (which is when the code is being executed in parallel), the microtask is called. When the GOMP_parallel_end is executed, the whole team ends up executing and enter the implicit barrier. The initial thread is then allowed to proceed executing sequentially.

```
... 
.type main, @function 
main:
.LFB0:
...
      movl $main__omp_fn.0, %edi
      call GOMP_parallel_start
      movl $0, %edi
      call main__omp_fn.0
      call GOMP_parallel_end
...
.type main__omp_fn.0, @function
main__omp_fn.0:
.LFB1:
...
```

In order to generate the code, the FE identified the parallel region and outlined it in the main__omp_fn.0 function (so that it could be executed, at run-time, by the team). The arguments of the outlined function (or microtask) 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. Typically, every compiler with OpenMP support include calls to a runtime library in the generated IR, so that
behaviors like team management and barriers work as expected (GOMP\_parallel\_start and GOMP\_parallel\_end, in the code shown above). Some other behaviors are ensured by simply injecting IR code on the microtask (creation of private variables, for instance).

The current thesis works 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 ME and BE support (see figure 2.4). At the moment, the branch fully supports version 3.1 of the OpenMP API, although efforts are being made to move the support to version 4.0.

![Figure 2.4: Clang+LLVM compilation process](image)

Figure 2.4 distinguishes the Clang, an LLVM Optimizer and an LLVM \textless target\textgreater Backend. As mentioned before, the Clang is responsible for all the FE stage. The ME is ensured by the LLVM Optimizer, taking LLVM IR code (placed on LLVM Modules) as input and emitting optimized LLVM IR code. The IR code is then be converted into machine dependent assembly code for a target dependent platform. After the assembly and linking stages, the assembly is converted into an executable. Both Clang and LLVM are built with a library-based design, making them easy to be embed with other platforms. The adaptability provided by this model was crucial for the selection of the compiler.

2.5 Summary

This chapter describes the background knowledge the reader should have in order to fully understand the present thesis. It describes the OpenMP model, including the execution, memory and synchronization model. A classic compilation process, including the implications of supporting OpenMP, is also described.
The execution model presents the main constructs: the parallel, the tasking, the SIMD and the work-sharing constructs. Parallel constructs create teams of threads to execute parallel regions. Despite the thread encountering the parallel construct being promoted the master of the team, all members execute the given parallel region. At the end of the parallel region, all threads synchronize on an implicit barrier, after which the master destructs the team and proceeds executing. Work-sharing constructs distribute work among the threads of the team. The distribution is dependent on the work-sharing construct (loop, sections or a single) and, in case of loop constructs, the scheduling clause.

The memory model distinguishes between shared memory, threadprivate memory and temporary view of memory. Shared memory is directly accessed by threads every time a read or write operation is made on a shared variable. Since OpenMP does not ensure atomicity when accessing to shared memory, it provides a synchronization model for the creation of critical regions, so that data race conditions are avoided. In order to avoid much accesses to shared memory, each thread is allowed to have its temporary view of memory, where the values of the shared variables are stored. All temporary views of a team are synchronized whenever an implicit or explicit flush operation is made. Threadprivate memory is used to keep the value of threadprivate variables between parallel regions. Regarding private variables, they are allocated in the private memory of the thread and its initialization is dependent on the data-sharing clause specified.

Regarding compilation processes, it is highlighted a chain of the modules Front, Middle and Back Ends. OpenMP support influences the Front-End stage by injecting IR code, so that the expected behavior is ensured. For the development of this thesis, the Clang compiler was chosen whose structure follows the classic model presented here.
Most OpenMP solutions for heterogeneous embedded systems are architecture-specific (the solution presented on article (O’Brien, Brien, and Sura 2008), for instance) and/or propose extensions to the OpenMP API (as presented by the articles (Liu and Chaudhary 2003) and (Ayguade, Badia, Bellens, Cabrera, Duran, Ferrer, González, Igual, Jiménez-González, Labarda, Martinell, Martorell, Mayo, Pérez, Planas, and Quintana-Ortí 2010)). Although hiding implementation details, (Wang, Chandrasekaran, Chapman, and Holt 2013) emerges with an eye on generality.

This chapter reports the state of the art in the scope of OpenMP solutions for heterogeneous many-core embedded systems. Section 3.1 refers to a portable OpenMP Runtime Library, targeting heterogeneous and homogeneous systems, called libEOMP. Section 3.2 refers to an architecture-specific OpenMP implementation for a many-core embedded system. Section 3.3 presents an evaluation of OpenMP support costs on MPSoCs. Finally, section 3.4 expose synchronization algorithms (key for performance).

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

In (Wang, Chandrasekaran, Chapman, and Holt 2013), 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. Figure
3.1 shows the Embedded OpenMP solution stack proposed by the authors, which is composed by the hardware layer, the System Layer, the MCA-APIs Layer, the OpenMP Programming Layer (which supports the libEOMP) and the Application Layer.

![Figure 3.1: Embedded OpenMP solution stack](image)

MRAPI presents the following key features:

**Domain and Nodes**  A MCA Domain is an unique system global entity that may contain one or more MCA Nodes (that can be a process, thread, processor or hardware accelerator).

**Synchronization Primitives** Support for basic synchronization features (like mutexes, semaphores, reader/writer locks).

**Memory Primitives** Support for shared and remote memories. Remote memory primitives provide the feature based on the observations that modern heterogeneous embedded systems often contain multiple memory spaces which are dedicated to each core.

**Metadata Primitives** Support for hardware and application execution statistics.

OpenUH is a source-to-source OpenMP compiler. The translation works as follows:
1. The source code is parsed and translated into WHIRL IR, an intermediate representation, with OpenMP directives;

2. An Inter Procedural analyzer step is performed;

3. A Loop nest optimizer step is performed;

4. Transformation of OpenMP, i.e., OpenMP directives are translated into WHIRL representing multithreaded code with OpenMP runtime library calls;

5. A Global scalar optimizer is performed.

The remainder process is target-dependent: for Itanium platforms, a built-in code generator can be directly used to generate object files; for non-Itanium platforms, the whirl2c or whirl2f translator will be invoked instead (in this case, the code is translated into multithreaded C or Fortran code with OpenMP runtime calls).

Figure 3.2: OpenUH overview

Figure 3.3 shows the overview of the resulting compilation process, for a given application app.c with OpenMP primitives. OpenUH compiler, presented in article (Liao, Hernandez, Chapman, Chen, and Zheng 2007), is used for source-to-source translation, resulting in
an intermediate C file app.w2.c with runtime library function calls. The file is then fed into the back-end native compiler (Power Architecture gcc compiler, in figure 3.3). Afterwards, all object codes are linked together with runtime libraries libopenmp and libmca (which are previously compiled by the native compiler).

Figure 3.3: Cross Compilation process

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 scalability 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 (see figure 3.4). 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 (see figure 3.5). All processors are connected to a crossbar network, so each of them is able to access to other processor’s 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.

Figure 3.4: Cyclops 64 architecture

Figure 3.5: C64 chip architecture
The threading model, specially designed for C64 (called Tiny-Threads or TNT), is described in (del Cuvillo, Zhu, Hu, and Gao 2005). 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. A scratch-pad memory with an TNT descriptor is shown in figure 3.6.

![Figure 3.6: Scratchpad Memory with TNT descriptor](image)

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.

The article (Sato, Shigehisa, Kusano, and Tanaka 1999) describes an OpenMP implementation for Symmetric Multi-Processor clusters. Omni OpenMP exploits Software Distributed Shared Memory systems so that coherence-maintenance network traffic is minimized. To improve the power of Cyclops-64, IBM has ported Omni-1.6 OpenMP Compiler to C64 and optimized the resulting implementation (whose results are reported in (del Cuvillo, Zhu, and Gao 2006)). The original Omni OpenMP system uses POSIX threads, but C64 uses Tiny-Threads.

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 micro-task). Originally, slave threads are given access to the master's 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 (in fact, only collective operations such as reduction forces the worker to access the master descriptor).

The optimized synchronization model includes:

- A spin lock mechanism that is available for applications that work under low memory contention. Since the C64 has no data cache, under high contention a thread spinning on a lock interferes with other threads by generating traffic on the crossbar network;

- A mutex mechanism that puts a thread to sleep when it fails to lock the mutex. When asleep, a thread unit does not execute instructions until another thread unit generates a wake up signal;

- Hardware-based barriers.

The optimizations reduced overhead in 80% when compared to the original Omni run time library.
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 (Marongiu, Burgio, and Benini 2010), 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 shown in figure 3.7. The number of processing elements is configurable (up to sixteen), 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 al-
3.3. EVALUATING OPENMP SUPPORT COSTS ON MPSoCs

Figure 3.7: MPSoC architecture

gorithm, 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.

One of the strengths of this work is the study of 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 master’s 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.

For this implementation the default behavior of private variables was kept, by duplicating their declarations at the beginning of the parallel region code.

3.4 Synchronization Algorithms

In January 1991, the paper (Mellow-Crummey and Scott 1991) was written with the purpose of reviewing, introducing and comparing synchronization algorithms (for spin locks and barriers). Below those algorithms are explain, as well as the conclusions of the authors.

3.4.1 Spin-lock algorithms

Three spin lock algorithms are discussed: simple test-and-set locks, ticket locks and array-based queuing locks. Additionally, a new list based queuing lock algorithm is introduced, MCS.
3.4. SYNCHRONIZATION ALGORITHMS

3.4.1.1 Simple test-and-set locks

Each processor repeatedly checks the lock to see if it is available. If true, it marks it as unavailable. Test-and-set is an instruction that does the behavior just described atomically.

The principal shortcoming of this algorithm is the contention for the flag. Introducing a delay between consecutive probes reduce the problem. The delay may grow exponentially.

3.4.1.2 Ticket locks

In order to ensure fairness and eliminate the possibility of starvation, ticket locks use two counters: one for counting the amount of waiting threads, and another for counting the number of times the lock has been released. Every time a thread fails to acquire a lock, it increments the first counter and keeps the returned value (ticket). When the owner releases the lock, the second counter is incremented. Counters are modified atomically by fetch_and_set instructions. Since waiting threads repeatedly check the second counter, waiting for its turn to acquire it, this algorithm also suffers from memory and network contention. Once again, introducing a delay between consecutive probes reduce the problem. This time the delay should be computed as the difference between a newly obtained ticket and the current value of the release counter.

3.4.1.3 Array-based queuing locks

Each process uses a fetch_and_increment instruction to obtain the address on which to spin (i.e., the next array element). The address is stored in cache. Whenever the owner releases the lock, it wakes up the next thread by setting its value in the array.

Array-based queuing locks reduce traffic, in case of ensured cache-coherence, and ensure FIFO ordering.
3.4.1.4 MCS locks

This algorithm guarantees FIFO ordering of lock acquisitions, spins on locally-accessible flag variables and work equally well without coherent caches. In addition, it introduces the least amount of interconnection contention.

MCS locks’ operation is based on the structure shown by figure 3.8. A MCS lock is allocated in shared memory and it is composed by a boolean value (expressing the state of the lock) and a distributed queue of waiting threads (exemplified by figure 3.9, situation 1). Whenever a thread desires to acquire a lock, it allocates a local MCS lock structure.

Using an unconditional atomic exchange operation, it stores the address of its own structure in the lock’s next field and marks the main lock as taken. The atomic exchange will return the previous value of the next pointer. Since that pointer was null, the acquiring thread knows it was successful in acquiring the lock (situation 2). If a second thread does an atomic swap on the main lock, it will get the previous contents of the next field - the pointer to the first thread’s MCS lock structure. The non-NULL value tells the second thread that the lock is not available, while the specific pointer value says who is ahead in line for the lock. The second thread will respond to this situation by storing a pointer to its local lock structure in the next field of the first thread lock’s structure (situation 3). Once this assignment is done, the second thread will spin on the locked value in its own lock structure rather than the value in the main lock. When owner of the lock finally tries to release the lock, it will perform a compare-and-swap operation on the main lock, trying to set the next pointer to NULL on the assumption that this pointer still points to its own structure. If that operation succeeds, the lock was never contended and the job is done. If some other thread has changed, the compare-and-swap will fail. In that case, first thread will not change the main lock at all; instead, it will change the locked value in second thread’s structure and remove itself from the situation. Once its copy of is_locked changes, the second thread will break out of its spin and become the new owner of the lock (situation 4).
3.4. SYNCHRONIZATION ALGORITHMS

3.4.1.5 Discussion

MCS and, for cache-coherent machines, the array-based queuing locks scale better than the algorithms above in the list. The other two elements present good scalability only when an appropriate backoff is chosen, but they always introduce more network load. However, they are not suitable when locks are being competed for a large number of threads or when threads compete for a large number of locks because it introduces too much memory and network traffic. Comparing the four algorithms, MCS is the best choice.

3.4.2 Barrier Algorithms

Four algorithms are discussed: centralized barriers, software combining-tree barriers, dissemination barriers and tournament. Additionally, a new tree based barrier is introduced.
Chapter 3. Related Work

3.4.2.1 Centralized Barriers

Each thread updates shared state (a counter, for instance) to indicate its arrival to the barrier. When all threads have reached the barrier, they continue past the barrier. The spinning that occurs on a single shared location degrades the performance of the system.

3.4.2.2 Software Combining-Tree Barrier

The Software Combining-Tree Barrier is based on a tree of counters. The threads are divided into groups, being each of them assigned to a leaf. The value initialized for each node is equivalent to the size of the group assigned to it. When a thread arrives to the barrier, it decrements the shared node of the tree. In case the value is set to 0, the thread repeats the procedure on the parent of the node. By the time the root is set to 0, the whole team is ensured all threads have arrived to the barrier. Figure 3.10 depicts a software combining tree initialized, for a team of thirty threads.

3.4.2.3 Dissemination Barrier

The Dissemination Barrier is based on pairwise synchronizations between a set of threads. After \([\log_2(n\_threads)]\) rounds, all threads have already synchronized between them. Figure 3.11 exemplifies the execution of the algorithm for 6 threads (3 rounds). Each node specifies the threads known as terminated, in black, and the threads the node is waiting for, in red.
3.4. SYNCHRONIZATION ALGORITHMS

3.4.2.4 Tournament Barrier

The Tournament Barrier is based on a tournament of threads whose winner of each round is statically determined. On each round the winner thread 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. The execution of the algorithm involves binary trees, where each player starts the algorithm on a leave of the tree.

This algorithm is appropriate for multiprocessors that use broadcast to maintain cache consistency. Otherwise, it will cause too much interconnect traffic. The algorithm can be modified so that Thread 0 broadcasts wake-up messages to all threads, instead of setting a global flag. Consequently, each thread could spin on locally allocated flags, thus solving the problem.

3.4.2.5 New Tree-Based Barrier

The tree-based algorithm starts by assigning an unique tree node (whose structure is represented by figure 3.12) to each thread. Each node is linked into an arrival tree by a parent link.
CHAPTER 3. RELATED WORK

Figure 3.12: Tree-based barrier’s node

(\texttt{parent\_ptr}) and to a wakeup tree by a set of child links (\texttt{children\_ptr}). Each node also owns a \texttt{childnotready} and a \texttt{parentsense} flags. Upon arrival at the barrier, a thread checks whether the \texttt{childnotready} flag is clear for all its children (leaf nodes are always clear, so deadlocks cannot arise). If true, the thread immediately re-initializes them for the next barrier and clears its own \texttt{childnotready} flag. All threads (except the root) spin on their locally-accessible \texttt{parentsense} flag. When the root node’s associated thread arrives at the barrier and notices that all of the root node’s \texttt{childnotready} flags are clear, it sets the \texttt{parentsense} flag in its children. By the time a \texttt{parentsense} flag is set by the parent of the given node, its associated thread is able to leverage the barrier.

3.4.2.6 Discussion

The space needs are constant for the centralized barrier, linear for the tree-based barrier and for the combining tree barrier, and \(O(P \log P)\) for the dissemination and tournament barriers. Regarding network transactions the dissemination barrier requires a total of \(O(P \log P)\) transactions, whereas the tournament and tree-based barriers require \(O(P)\). The centralized and combining tree barriers require \(O(P)\) on machines with broadcast-based coherent caches and a potentially unbounded number on other machines. On a machine in which independent network transactions can proceed in parallel, the critical path length is \(O(\log P)\) for all but the centralized barrier, which is \(O(P)\). On a machine that serializes network transactions (e.g., on a shared bus) this logarithmic factor will be dominated asymptotically by the linear (or greater) total number of network transactions.
3.5 Summary

This chapter reported the state-of-the-art. Majority of OpenMP solutions for heterogeneous embedded systems are architecture-specific and/or propose extensions to the OpenMP API.

The chapter is introduced by presenting libEOMP, an OpenMP Runtime Library based on MCA APIs for Embedded Systems. This work results from the union of an already existent OpenMP source-to-source translator (OpenUH) with the MCA APIs (that is, an API specially developed by the Multi-Core Association for communicating with heterogeneous embedded systems). Then, the Cyclops-64 (an OpenMP model for distributed shared-memory many-core system) is presented, as well as an evaluation of the OpenMP support costs on MPSoCs. In both of them, the threading and synchronization models are discussed. In addition, the latter includes an evaluation of the placement of the shared variables.

The chapter is finished with an exposure of synchronization algorithms, where the following spin-lock algorithms are discussed: Simple test-and-set locks, Ticket locks, Array-based queuing locks, MCS locks. Regarding barriers: Centralized barrier, Software Combining Tree barrier, Dissemination barrier, Tournament barrier, New Tree-Based Barrier.
The system developed for this thesis follows the architecture shown by figure 4.1, where two connected systems are represented: the Host and the Guest systems. The Host system is responsible for starting the execution of OpenMP programs, reason why it requires the assistance of an Operating 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. The intent here is to produce a general model, suitable for every Guest system that meets the basic requirements presented during this chapter.

![Figure 4.1: Basic hardware architecture](image)

Equation 4.1 gives the speedup of a parallel execution in relation to a sequential execution (where STime means sequential time, while PTime means parallel time). Since the overhead may have two sources (hardware and software), it is reasonable for us to consider equation 4.2.

\[
\frac{\text{STime}}{\text{PTime} + \text{Overhead}}
\]  

(4.1)

\[
\frac{\text{STime}}{\text{PTime} + \text{HardwareOverhead} + \text{SoftwareOverhead}}
\]  

(4.2)
The majority of OpenMP solutions target homogeneous systems like general purpose computers. Those implementations are fully assisted by Operating Systems, facilitating the communication between threads. The cost of managing software threads is the increase on software overhead associated with thread management. Since general purpose systems have a small processing capacity when compared with embedded systems, the parallelization of programs is much more limited by hardware. Then, embedded systems are capable of producing higher speedups by processing programs in lower PTime. In order for this to happen it is key to ensure low hardware and software overheads.

Throughout this chapter the constructs and clauses shown on Table 4.1 are discussed. Additionally, the following Runtime OpenMP Library Routines are explored: omp_get_num_threads, omp_set_num_threads and omp_get_thread_num.

<table>
<thead>
<tr>
<th>Construct</th>
<th>Clauses</th>
</tr>
</thead>
<tbody>
<tr>
<td>PARALLEL</td>
<td>if(expr)</td>
</tr>
<tr>
<td></td>
<td>private(list)</td>
</tr>
<tr>
<td></td>
<td>shared(list)</td>
</tr>
<tr>
<td></td>
<td>default (shared=none)</td>
</tr>
<tr>
<td></td>
<td>firstprivate (list)</td>
</tr>
<tr>
<td></td>
<td>num_threads(integer=expr)</td>
</tr>
<tr>
<td>FOR</td>
<td>schedule(type [,chunksize])</td>
</tr>
<tr>
<td></td>
<td>private(list)</td>
</tr>
<tr>
<td></td>
<td>firstprivate(list)</td>
</tr>
<tr>
<td></td>
<td>lastprivate(list)</td>
</tr>
<tr>
<td></td>
<td>shared(list)</td>
</tr>
<tr>
<td></td>
<td>nowait</td>
</tr>
</tbody>
</table>

Table 4.1: Supported constructs and clauses

The execution model is presented by section 4.1, while the memory and synchronization models are discussed by sections 4.2 and 4.3, respectively. The chapter terminates with a description of the software architecture of the model.

### 4.1 Execution Model

Accordingly to the OpenMP’s specification, a program starts executing by an initial thread, which creates and destructs teams of threads as it encounters parallel constructs. Each team is responsible for executing a given parallel region and, if required by the user, creating addi-
tional teams of threads. Section 4.1.1 describes how this behavior is achieved. Work-sharing constructs, probably the most helpful for OpenMP programmers, distribute an iteration space of a given for statement by the threads available on the current team. Section 4.1.2 describes how this behavior is achieved. Throughout this section, as well as section 4.2, it is assumed that the library CHTLib handles the communication between the host and the guest systems.

### 4.1.1 Parallel Construct

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. Figure 4.2 demonstrates the files generated when the test.c source file (whose code is shown below the figure) is compiled. Three files are generated (since the compiler is not the focus of the figure, and it is discussed later, it is represented by a big black dot): one to be executed by the host thread (a.out) and two others, encapsulating the first and second parallel regions (test-omp_microtask_0-CHT.o and test-omp_microtask_1-CHT.o), that are archived in a static library.

```cpp
#include <stdio.h>
```
int main() {
    /* parallel code */
    #pragma omp parallel
    {
        int a = 0;
        a++;
    }
    /* sequential code */
    printf("sequential code\n");

    /* parallel code */
    #pragma omp parallel
    {
        int b = 7;
        b++;
    }

    return 0;
}

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. The case of `omp_get_thread_num` is handled in a differently and it is discussed later.

### 4.1.2 For Construct

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 only scheduling clause implemented for this thesis, 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. As said before, the static scheduling assigns chunks to the threads of the current team in a round-robin fashion (ordered by thread id). This model allows all processing units to receive the exact same binary code. The following pseudo-code depicts the execution of an OpenMP for construct, with static scheduling and no chunk size specified. As the nowait was not specified, a barrier is executed at the end of the for region. Barriers executed exclusively by the guest system are discussed later. Although suitable, the implemented model does not support the collapse neither the ordered clauses. However, it is possible to nest for constructs (the code is emitted as content of the outer for loop).

```plaintext
NIts ← GlobalUB − GlobalLB
if GlobalLB = 0 then
    NIts ← NIts + 1
end if
if NIts < NThrs then
    if ThrdId > NThrs then
        enter Barrier
    end if
    ChunkSize ← 1
```
else
    \text{ChunkSize} \leftarrow \frac{\text{NIts}}{\text{NThrs}}
end if

\text{TotalIts} \leftarrow \text{ChunkSize} \times \text{NThrs}
\text{LB} \leftarrow \text{GlobalLB} + \text{ChunkSize} \times \text{ThrdId}
\text{UB} \leftarrow \text{LB} + \text{ChunkSize} - 1

\text{omp.loop.main:}
\text{for \text{idx} := LB to UB do}
    \text{execute code}
\text{end for}

\text{if \text{LB} + \text{TotalIts} > \text{GlobalUB} then}
    \text{enter Barrier}
\text{end if}

\text{LB} \leftarrow \text{LB} + \text{TotalIts}
\text{if \text{LB} + \text{TotalIts} < \text{GlobalUB} then}
    \text{\text{UB} \leftarrow \text{LB} + \text{ChunkSize} - 1}
\text{else}
    \text{\text{UB} \leftarrow \text{GlobalUB}}
\text{end if}

\text{goto omp.loop.main}

Figure 4.3 exemplifies the execution of the algorithm above. It is represented an iteration space of size seven whose execution is distributed by two threads. In an initial phase, each thread executes exactly \( \lfloor \frac{\text{NIterations}}{\text{NThreads}} \rfloor \). Before leaving the for region, each thread makes sure it has no more iterations to execute with the expression 4.3, where \text{LB} refers to the current local upper bound, the \text{TotalIts} refers to the amount of the executed iterations and the \text{GlobalUB} refers to the global upper bound.

\[
\text{LB} + \text{TotalIts} < \text{GlobalUB}
\] (4.3)
4.2 Memory Model

Besides private memory, the OpenMP’s specification uses a memory model based upon the existence of a memory shared by all processing units of the system, a thread private memory whose contents persist across parallel regions and a temporary view of memory. As shown by figure 4.4, in order to reproduce this model it is assumed by the current thesis that the guest system is owner of a memory shared by all processing units (see section 4.2.1) and each processing unit is owner of a private memory that is not accessible by any other processing unit (see section 4.2.2). Thread private storage and temporary view of memory are not included here. Hardware overhead and memory management are the main concerns throughout this chapter.
4.2.1 Shared Memory

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.

Besides other contents, the shared variables accessed within a parallel region are copied to the shared memory of the guest system the moment before the execution of the team starts. The simultaneous execution of multiple programs can lead to memory fragmentation, thus resulting in a non-contiguous allocation of the variables. 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.

As mentioned before, OpenMP does not ensure atomicity when accessing shared resources. Consequently, the user is required to avoid data races by accessing to shared resources within critical regions. 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.

For team synchronization, at barriers, shared resources also have to be allocated on the guest system. The allocation is done the moment before the execution of the parallel region,
while the release is done the moment after. However, the CHTLib is only asked to allocate and release memory when a barrier is detected on the source code of 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. Section 4.3 discusses the algorithms used by locks and 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 host’s memory, so that the sequential execution proceeds with up-to-date values.

### 4.2.2 Private Memory

Figure 4.5 generalizes the usage of private memory by a given guest thread. Although part of the address space seen by a guest thread is shared between all members of the team, the address space represented in figure 4.5 ranges between the addresses 0x000000 and 0xFFFFFFF for simplicity reasons. The address space is divided into four sub-address spaces:

1. **Program Memory:** address space where the binary code being executed is stored;

2. **Thread Ids:** The thread identification number of the guest thread is stored here. So, calls to `omp_get_thread_num` are translated into loads of this memory space. An alternative would be the inclusion of the thread id in the set of arguments of the microtask. However, this would prevent the arguments to be loaded on memory by message broadcast (since each thread own a different thread id), thus increasing hardware overhead.

3. **Arguments:** Every time an host thread asks the CHTLib to create a guest team for the execution of a given parallel region, the arguments of the microtask are loaded here. By the time a guest thread starts executing, the contents of this address space are loaded on stack before the microtask is called. Later it is discussed how this behavior is achieved.

4. **Stack memory.**
Private variables are local variables whose initialization is dependent on the data-sharing clause specified by the user. Firstprivate variables are initialized with the original value of the given variable (stored in shared memory). On the other hand, private and lastprivate variables are not initialized at all (so the user has to explicitly initialize them inside the parallel region). Lastprivate variables are specific to work-sharing constructs and certify that, before the termination of its execution, the thread responsible for executing the last iteration updates the value (of the given variable) stored in shared memory with the value kept by its local copy. By the time the user copies back the shared variables to host’s memory, it will be provided with the new value of the lastprivate variables.

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. For
both cases, an algorithm was selected from a list of algorithms (previously discussed by section 3.4), considering the requirements of the system. The model described here assumes that guest threads can not communicate directly, so shared resources are used for indirect communication. It is also assumed that once a guest processing unit starts executing, it will proceed until completion of the task or in case of errors.

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:

- Simple test-and-set locks;
- Ticket locks;
- Array-based queuing locks;
- MCS locks.

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 an instruction which is not found in some Instruction Set Architectures (for instance the Microblaze’s Instruction Set, which was used to test this implementation): fetch and increment. 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:

- Centralized Barriers;
• Software Combining-Tree Barrier;

• Dissemination Barrier;

• Tournament Barrier;

• New Tree-based Barrier.

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 traffic it generates. For the benefit of the algorithm, the Instruction Set Architecture of the guest system is assumed to include a test_and_set instruction, besides generic load and store instructions. The resources allocated for the execution of barriers can be seen in figure 4.6. It depicts the following entities:

**Announcements** A two-sized boolean array where the champion of the tournament announces the final decision. The size of the array allows the whole structure not to be reinitialized every time a barrier synchronization is executed.

**BarrierCount** A boolean value used to indicate the position of the Announcements array to use on the next barrier synchronization.

**Answers** An matrix with \( \text{ceil}(\log(num\_threads - 1)) \times num\_threads \) positions where, for each instance of the tournament, the arrival of each loser is communicated to the winner of the game.

For each parallel region with barrier synchronizations, the host system is required to initialize the barrier structure, a sole time, with the algorithm below.

```latex
\textbf{function} \textsc{InitBarrier}(num\_threads)\\
\text{BarrierCount} \leftarrow 0
```
4.3. SYNCHRONIZATION MODEL

Figure 4.6: Barrier

\[ \text{Announcement}[0] := \text{Announcement}[1] := \text{false} \]

\[
\text{for instance} := 0 \text{ to } \text{ceil}(\log(\text{num\_threads} - 1))\text{ do}
\]

\[
\text{for thrid} := 0 \text{ to } \text{num\_threads} - 1 \text{ do}
\]

\[ \text{Answers}[\text{thrid}][\text{instance}] \leftarrow \text{false} \]

\[
\text{end for}
\]

\[
\text{end for}
\]

\[
\text{end function}
\]

The algorithm below demonstrates a barrier synchronization. On each instance of the algorithm, a pre-chosen winner (the one with a lower thread id) waits for the arrival of the pre-chosen loser to the barrier. After losing, the loser waits the final decision of the champion (the master of the guest team) by listening to the content of the position BarrierCount of the Announcements array. The function below is generated on the module of each parallel region with barrier synchronization.

\[
\text{function BARRIER(thread\_id, num\_threads)}
\]

\[
\text{power} \leftarrow 1
\]

\[
\text{LocalBarrierCount} \leftarrow \text{BarrierCount}
\]

\[
\text{for instance} := 0 \text{ to } \text{ceil}(\log(\text{num\_threads} - 1))\text{ do}
\]

\[
\text{Opponent} \leftarrow \text{thread\_id} \text{ XOR power}
\]

\[
\text{if thread\_id mod } 2^{\text{instance}} \neq 0 \text{ then}
\]

\[
\text{break}
\]

\[
\text{else if thread\_id > Opponent then} \quad \triangleright \text{loser}
\]

\[
\text{end function}
\]
Answer[thread_id][instance] ← true

else if Opponent ≥ num_threads then
    /* Win by default, if no opponent.*/
else

    while Answer[Opponent][instance] = false do wait

end while

Answer[Opponent][instance] ← false

end if

power ← power ⋆ 2

end for

if thread_id = CHAMPION then

    BarrierCount ← (BarrierCount + 1) mod 2
    Announcement[BarrierCount] ← false
    Announcement[LocalBarrierCount] ← true

else

    while Announcement[LocalBarrierCount] = false do wait

end while

end if

end function

Figure 4.7 demonstrates the execution of the Tournament Algorithm within a team of five threads. Leafs represent thread, nodes represent games, and dotted lines represent imaginary entities. Within each node is the id of the winner thread. Only two instances are necessary for the termination of the tournament.

4.4 Software Architecture

It has hitherto been considered that the communication between the host and guest systems is handled by CHTLib. Figure 4.8 represents a data flow diagram where the whole communication process is demystified. Besides the host thread and the Guest System, the following entities are depicted:
4.4. SOFTWARE ARCHITECTURE

**CHTLib** Called in run time, by the host thread, to communicate with the guest system. Depending on the requests of the host, the CHTLib manages the contents owned by the guest system. Its operation is detailed by section 4.4.2.

**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.

Its operation is detailed by section 4.4.1 and assumes that multiple applications can execute simultaneously.

**Clang Compiler** Responsible for the compilation of the source code. If -fopenmpcht enabled, the code encapsulated by OpenMP parallel constructs is exported to different llvm Mod-
ules, so that different binary files can be generated for different parallel regions. By the time the Back-End stage of the compilation process starts executing, the llvm Modules containing parallel regions are ignored and the llvm Module targeting the host system is compiled.

**Device Driver** Generates binary code to be run on the guest system (arrows 2 and 3 on the figure). LLC, a llvm tool, is called by the driver so that the input llvm modules (arrow 1) are compiled for the target Instruction Set Architecture.

![Figure 4.9: Compilation process of a C source file](image)

### 4.4.1 Device Driver

The Device Driver is responsible for allocating and releasing memory resources, writing and reading contents to/from guest memory and run thread blocks (equivalent to teams of thread) on the guest system. So, the driver’s structure is composed by a low and an high levels (see figure 4.10). 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.

The low level layer supports memory transfers between the host’s and the device’s (shared or private) memories, the execution of thread blocks and the waiting for the termination of the execution of thread blocks. In order for a thread block to be executed, the user is required to:

1. Load shared contents on the device’s shared memory;
2. Load private contents (including the program to be executed) on the private memories of a previously selected set of processing units;

3. Instruct the previously selected set of cores to execute.

The whole management work is left to the high level layer. For the management of shared memory resources, 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 job scheduling, the reader should assume the following concepts:

**Core**  Processing unit with private memory and access to shared memory address space.

**Cluster**  Set of cores. Although OpenMP does not include this concept, the processing units of most embedded systems are grouped in this way;

**Kernel**  Binary code that should be processed by a given number of cores.
**Job** Descriptor of the current execution’s state, specifying the size of the thread block that should execute a given kernel and how many threads have already started. A job is created every time the user asks for the execution of a given kernel with a given set of arguments.

**CmdExec** Encapsulates a kernel, its arguments and a queue of jobs (the execution of multiple thread blocks with the same kernel and arguments is seen as multiple jobs in the queue of the same CmdExec).

**ThreadPool** Pool of CmdExecs.

Each time a CmdExec is submitted by the user of the driver, a set of free clusters (whose cores are all available) is assigned to execute the jobs. In case there are not enough clusters to execute all jobs of a CmdExec, the execution of the assigned jobs is proceeded, while the remaining executions are delayed until the assignment of more clusters to the CmdExec. Each time a cluster ends up executing, waiting jobs are searched in the thread pool. By the time the execution of a thread block is finished, the user is informed (so, the execution of a thread block is non-blocking). To understand the disadvantages of the algorithm, please imagine a device with two clusters of eight cores each. Now, imagine two different CmdExecs were submitted, where the first one requires a team block of size 6, while the second one requires a team block of size 2. In this case, resources will be wasted because both CmdExecs will not be assigned to the same cluster. However, low overheads related to transfers to private memory (to be more precise, the overhead associated with the copy of the kernel and the arguments to the memories) are produced by the algorithm. By the time a given cluster is assigned to a CmdExec, both the kernel and the arguments are transferred to the private memories of all cores belonging to that cluster. When multiple jobs are assigned to the same CmdExec, there is only one transfer of the private contents. This algorithm benefits the dynamic scheduling, while not damaging the performance of static scheduling. Threads may not be moved between processing units and the user is not able to select the thread affinity policy.
### 4.4.2 CHTLib

The main purpose of CHTLib library is to prepare the guest environment and to execute parallel regions there. The library performs the following actions (by the order specified):

1. Allocation of space for each shared variable;
2. Allocation of space for the locks used by critical and atomic regions;
3. Creation of a CmdExec;
4. Loading of the arguments of the microtask on the CmdExec;
5. Execution of the parallel region;
6. Waiting for the DeviceDrive to call back the CHTLib, when the execution of a thread block finishes.
7. Copy back the values of the shared variables from the shared memory of the guest system to the host’s memory;
8. Release of shared resources.

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 guest’s resources.

### 4.5 Summary

This chapter presents the proposed model for the implementation of OpenMP on heterogeneous embedded systems. The execution, memory and synchronization models are discussed. The chapter is finished with a description of the software architecture implied on the proposed system, which is composed by the Clang compiler, the CHTLib and a DeviceDriver specially created for the communication between an host system and a (not specific) guest system.

The OpenMP application starts executing on an host thread. Whenever a parallel construct is encountered, a guest team of threads is spawned and asked to execute a parallel region. Each
guest thread is given the binary code (previously compiled for the specific target ISA) of the parallel region to be executed. The execution of the parallel region is finished by an implicit barrier, where the host thread waits for the termination of execution of the guest team. In case a for construct is specified, an iteration space is distributed by the threads of the current team. The division respects the static scheduling, where each thread is able to preview which iterations it is supposed to execute. If no nowait clause is specified, an implicit barrier is executed entirely on the guest system. Regarding the synchronization model, those previews the execution of barriers, critical constructs and atomic constructs. Barriers implement the tournament algorithm, which is based on a tournament whose losers and winners are statically determined. Winners wait for losers and losers wait for the end of the tournament to proceed executing. Atomic and critical constructs are based on simple locks. Each of them is owner of a specific lock that is locked whenever a thread is executing the critical construct.

The memory model is based upon the existence of a shared memory and one private memory per processing unit. Shared contents are placed in shared memory before the guest system starts executing and all guest threads are given access to the locations of the shared contents. Private contents are obviously placed in private memory in run time.
As mentioned before, the goals of this thesis rely on the production of a portable, scalable and high performance model for the support of OpenMP on heterogeneous embedded systems. In order to validate the developed model, a specific architecture was used.

The present chapter is initialized by a description of the experimental platform used here, followed by the description of the case studies. The model developed during this thesis is then embedded in the hardware architecture described below and a discussion is made regarding the results.

5.1 Experimental platform

The hardware architecture that supports the case studies presented here is composed by an Intel Core i7 3770K @ 3.5GHz (host computer) and a Field Programmable Gate Array (Virtex-7). Figure 5.1 depicts the architecture of both systems. 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 (shown by figure 5.2), 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.
Each processor is based on MB-Lite, a robust, light-weight soft-core implementation of the MicroBlaze architecture. Since the LLVM Backend support for the microblaze architecture is obsolete for about a year ago, it had to be recovered. Additionally, the backend support had to be modified since the MB-Lite is Little-Endian (unlike the Xilinx Microblaze implementation). The MB-Lite is open-source and it is described by the article (Kranenburg and van Leuken 2010). The Microblaze architecture is fully described in its reference guide (Xilinx 2013).

Figure 5.1: Hardware Architecture

Figure 5.2: Processing Cluster’s Architecture

Figure 5.3 contextualizes the hardware architecture just described in the model presented throughout this dissertation. As the reader is able to see, the whole software architecture pur-
posed by section 4.4 is run in user space. For simplicity reasons, the Device Driver which supports the communication with the PCIe is named pciDriver throughout this section. As stated before, the Device Driver developed for this thesis is divided into the High Level and the Low Level layers. The Low Level layer is responsible for any memory transfers between both systems, thread block execution and waits for the termination of execution of the blocks. For the present platform, the implementation of the Low Level layer is dependent on the pciDriver. In order for the Low Level layer to order requests to the PCIe, the PCIe endpoint configuration registers are mapped to the address space of the Low Level Device Driver through calls to the function pd_open of the pciDriver. Additionally, to provide access to the address space of the accelerator, the pd_mapBAR is used to map one of the PCIe apertures to user-space. Both actions are performed in a setup_device stage. From this point, the Device Driver is able to control the configuration registers and the PCI aperture for the benefit of the application (that is, they are modified as the application asks for memory transfers, for the execution of thread blocks or when the execution status of the processing clusters/cores is controlled).

![Figure 5.3: Processing Cluster’s Architecture](image)

Thus, this system meets all the requirements imposed by the present thesis. The fact that the architecture does not include any cache system, allows us to testify the worst cases for each case study.
5.2 Use Cases

The following case-studies are considered throughout the current section:

**Prime Numbers** Calculates the prime numbers from 1 up to N. With it, the parallel and for constructs are tested, besides the shared, private and firstprivate data-sharing clauses. No synchronization constructs are required by the algorithm, so the lack of cache system is not evident here. With this algorithm, conclusions are reached about the relation between the number of threads and the speedup.

**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 of this benchmark 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 can not be seen on the previous algorithms. 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).

5.2.1 Prime Numbers

Below it is shown the pseudo-code of the Prime Numbers test. By the time the parallel construct is encountered by the host thread, the CHTLib is asked to generate a block of \( n_{threads} \) threads on the accelerator, taking into account: the microtask containing the parallel region, an array containing the locations (on the host) of the variables used within the parallel region (independently of the data-sharing clause) and the indication that it is necessary to allocate a barrier on the shared memory of the guest system. Given that the CHTLib communicates with the Accelerator through the Device Driver, the Device Driver is immediately asked to allocate
memory for each variable of the array and to write its value there. When the CHTLib is ensured that all memory allocations and writings have terminated (for both the variables and the barrier resources), a CmdExec is created on the Device Driver (that is, an instance of an object containing both the microtask and the arguments of the microtask), whose arguments include the size of the team, the location of the barrier on shared memory and an array containing the locations of the variables allocated in shared memory. Afterwards, the Device Driver is asked to execute the given CmdExec. When the execution of the CmdExec is completely terminated, the CHTLib copies back the value of each variable from the shared memory of the guest system to the host memory, through the Device Driver.

\textbf{function} \textsc{Prime}(N, nthreads)

\begin{verbatim}
omp_set_num_threads(nthreads)

#pragma omp parallel for firstprivate(N) private(prime, div1, div2, rem, i) shared(flags)

for i = 0; i < (N – 3)/2; ++i do

    prime ← 2 * i + 3

    div1 ← 1

    repeat

        div1 ← div1 + 2

        div2 ← prime/div1

        rem ← prime%div1

    until rem != 0 & div1 == prime

if rem != 0 div1 == prime then \triangleright prime is really a prime

    flags[i] ← 1

else

end
\end{verbatim}
flags[i] ← 0

end if

end for

end function

The Device Driver is fully aware of the memory structure of the accelerator. With that information, it is able to implement the Buddy memory allocation technique on the shared memory and to copy contents to the (shared or private) memories of the accelerator. The fact that it is also aware of which cores are executing at a given moment allows the Device Driver to assign free clusters to waiting CmdExecs. By the time the Device Driver knows that the CmdExec could be run by a Cluster, a closed set of cores belonging to the specific cluster is chosen. After loading the kernel, the arguments and the thread id of each core on its private memory, the cores start executing the microtask. Its first action is to locate variables in shared memory. The flags array is the only content shared between the team, besides the barrier, and its location is accessed whenever a thread is sure whether a given number is prime or not. Regarding the private variables, a copy of variable $N$ is made on each private memory, whereas the remaining private variables are allocated in private memory but not initialized. Since each position of the flags array is accessed only one time by only one thread, there is no need for synchronization constructs. However, since the accesses to the shared memory of the accelerator are serialized, the simultaneous accesses to the flags array, by multiple cores, increases the hardware overhead. Assuming that all threads of the thread block are asked to be executed nearly the same time, the hardware overhead associated to the accesses to the flags array is higher for a higher number of threads.

5.2.2 Factorial

The factorial test (whose algorithm is shown below) operates analogously to the Prime Numbers test. The additional critical region uses a lock, which is allocated on the shared memory of the accelerator the moment before the processing units start executing. Each core knows the location of the lock via microtask argument.
Although this algorithm is not a good example of parallelization, it demonstrates the overhead introduced on the system by the constant synchronization requirements. The hardware overhead is higher for bigger teams.

**function** FACTORIAL\((N, nthreads)\)

```c
omp_set_num_threads(nthreads)

#pragma omp parallel for private(i) shared(fact)

for \(i = 1; i <= N; ++i\) do

#pragma omp atomic

fact ← fact * i

end for
```

**end function**

### 5.2.3 N Consecutive Barriers

The intent of this test is to compare the overhead associated with explicit barriers, as well as to analyse the variation of the overhead as the number of threads and the amount of consecutive barrier synchronizations vary. Since the number of threads influence the amount of pairwise synchronizations, the software overhead associated with the team synchronization is directly influenced by the amount of threads of the team. Furthermore, the hardware overhead is also influenced by the size of the team since the system does not include a coherent cache system and accesses to shared memory are serialized. If a coherent cache system was included, the accesses to shared memory would be fewer and the hardware overhead would decrease substantially (the principle of spatial locality would be helpful for caches shared between the cores of each cluster, whereas the principle of temporal locality would be helpful for caches exclusive to each core). In both cases (whether a cache system is included or not) the hardware and software overheads remain practically constant between consecutive synchronizations since the device has no traffic on the bus, at the moment the next barrier synchronization starts.
5.3 Results

The results in the present section 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. The overhead associated to an empty parallel construct can be seen in Table 5.1, where it is noticeable the non-significant variation of the overhead, in function to the size of the team. 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.

<table>
<thead>
<tr>
<th>parallel</th>
<th>1 thread</th>
<th>8 threads</th>
<th>16 threads</th>
<th>40 threads</th>
<th>60 threads</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>0.363</td>
<td>0.293</td>
<td>0.334</td>
<td>0.346</td>
<td>0.315</td>
</tr>
</tbody>
</table>

Table 5.1: Overhead associated with the parallel construct

Table 5.2 shows the overhead associated with the barrier construct. As it was expected, the overhead associated with the barrier construct grows for larger teams. 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).

<table>
<thead>
<tr>
<th>barriers</th>
<th>1 thread</th>
<th>8 threads</th>
<th>16 threads</th>
<th>40 threads</th>
<th>60 threads</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 barrier</td>
<td>0.377</td>
<td>1.476</td>
<td>2.852</td>
<td>9.065</td>
<td>13.866</td>
</tr>
<tr>
<td>2 barriers</td>
<td>0.36</td>
<td>1.389</td>
<td>2.688</td>
<td>9.732</td>
<td>13.047</td>
</tr>
<tr>
<td>5 barriers</td>
<td>0.351</td>
<td>1.498</td>
<td>2.905</td>
<td>9.735</td>
<td>11.939</td>
</tr>
<tr>
<td>10 barriers</td>
<td>0.386</td>
<td>1.431</td>
<td>3.107</td>
<td>8.716</td>
<td>12.971</td>
</tr>
<tr>
<td>25 barriers</td>
<td>0.346</td>
<td>1.77</td>
<td>2.735</td>
<td>8.859</td>
<td>13.668</td>
</tr>
<tr>
<td>40 barriers</td>
<td>0.385</td>
<td>1.565</td>
<td>2.722</td>
<td>9.288</td>
<td>14.016</td>
</tr>
</tbody>
</table>

Table 5.2: Overhead associated with the barrier construct

Table 5.3 shows 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 is not significantly influenced by the growth of the size of the team, but it is 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
5.4. DISCUSSION

The execution of the case studies on the suggested experimental platform validates the portability of the model developed throughout this thesis. The fact that the binary files containing parallel code are not dependent on the binary files containing serial code allows the guest system to be sufficiently independent on the host system (for instance, the Instruction Set of the host is independent on the Instruction Set of the guest system). As a result, the implementation of the model on the purposed hardware is straightforward. Additionally, the choice of the algorithms enhances the portability due to its few (and owned by most hardware architectures) requirements.

Regarding scalability and performance, the model depends on the hardware architecture of the guest system. When the guest system includes a coherent cache system, the model achieves good scalability (taking advantage, most of the times, of both the temporal and spacial principles to reduce the number of accesses to shared memory). When the guest system does not include a coherent cache system, the scalability is compromised due to the increased amount of traffic generated. 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.

<table>
<thead>
<tr>
<th>Critical Constructs</th>
<th>1 thread</th>
<th>8 threads</th>
<th>16 threads</th>
<th>40 threads</th>
<th>60 threads</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 critical</td>
<td>0.565</td>
<td>0.546</td>
<td>0.516</td>
<td>0.542</td>
<td>0.652</td>
</tr>
<tr>
<td>2 criticals</td>
<td>0.447</td>
<td>0.645</td>
<td>0.59</td>
<td>0.565</td>
<td>0.68</td>
</tr>
<tr>
<td>5 criticals</td>
<td>0.797</td>
<td>0.7</td>
<td>0.633</td>
<td>0.807</td>
<td>0.744</td>
</tr>
<tr>
<td>10 criticals</td>
<td>0.777</td>
<td>0.818</td>
<td>0.846</td>
<td>0.946</td>
<td>0.954</td>
</tr>
<tr>
<td>20 criticals</td>
<td>2.088</td>
<td>2.187</td>
<td>2.392</td>
<td>2.682</td>
<td>2.521</td>
</tr>
</tbody>
</table>

Table 5.3: Overhead associated with the critical or atomic constructs
5.5 Summary

The present chapter validates the model purposed by this dissertation. The model was validated upon an heterogeneous embedded system composed by an host Intel Core i7 3770K @ 3.5GHz computer and a Virtex-7 FPGA whose processing elements have access to an external shared memory. In order for the host computer (more specifically, the Low Level layer of the Device Driver) to communicate with the accelerator, a PCI express is used. Its operation is supported by a pciDriver, which is installed on the host system.

Three case studies are studied here: the Prime Numbers, the Factorial and the N Consecutive Barriers tests. They are used to infer the portability, scalability and performance of the OpenMP constructs implemented for this thesis. These case studies show how the overheads are influenced by the software choices (for instance, the size of the team), as well as the hardware choices (for instance, the lack of a coherent cache system). Unlike the portability of the model, its scalability and performance revealed to be hardware architecture dependent. The overheads associated with the parallel, barrier and critical/atomic constructs are shown, from which it is possible to infer what can influence those results.
Conclusions

This dissertation presented a model that supports the development of OpenMP for heterogeneous embedded systems. The model is portable, scalable and high performance (although the last two dependent on the hardware architecture of the embedded system). 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 purposed by this thesis were met.

6.1 Future Work

For future work, the present thesis 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.

Regarding the work already done throughout this thesis, the loop construct would be tested by following the same reasoning as the one used here, so that the conclusions would be inferred from the results.
Bibliography


