Automatic Source Code Adaptation for Heterogeneous Platforms

Albert Saà-Garriga

Universitat Autònoma de Barcelona

Departament de Microelectrònica i Sistemes Electrònics

Automatic Source Code Adaptation for Heterogeneous Platforms

Albert Saà-Garriga

Memòria de Tesi
presentada per optar al títol de

Doctor en Informàtica

September 2016

Universitat Autònoma de Barcelona

Dr. Jordi Carrabina, Professor Titular del Departament de Microelectrònica i Sistemes Electrònics, Dr. David Castells-Rufas, Professor Associat del Departament de Microelectrònica i Sistemes Electrònics

Certifiquen

que la Memòria de Tesi *Automatic Source Code Adaptation for Heterogeneous Platforms* presentada per Albert Saà-Garriga per optar al títol de Doctor en Informàtica s'ha realitzat sota la seva direcció i ha estat tutoritzada en el Departament de Microelectrònica i Sistemes Electrònics de la Universitat Autònoma de Barcelona.

Directors	Dr. Jordi Carrabina	....
	Dr. David Castells-Rufas	....

........................., a ...... de .... de ....

Never send a human to do a machine’s job.
Agent Smith

Resum

La fi de l’increment de la freqüència de rellotge com a forma més fàcil de millorar el rendiment dels sistemes de computació, junt amb una creixent escletxa entre la velocitat d’accés a CPU i la memòria, així com l’increment en la intensitat aritmètica dels problemes que resol lacomunitat de Computació amb Altes Prestacions (HPC), ha donat a llum a una nova gama de arquitectures computacionals heterogènies que pretenen conduir a una important millora del rendiment.

Els processadores amb molts nuclis (many-cores), ja siguin homogenis o heterogenis respecte el sistema central de comput són una de les sol·lucions mes esteses per a intentar arribar a les prestacions requerides. No obstant això, la comple xitat d’aquestes noves arquitectures no es pot ocultar fàcilment al programador, i això dificulta la seva aplicació. En aquesta tesi proposo un conjunt d’eines que busquen facilitar la creació de codi que permeti l’aprofitament de les característiques de les arquitectures paral·leles heterogenies a través de transformacions en el codi font, amb l’objectiu de millorar el rendiment i al mateix temps incrementar l’eficiència energètica de aplicacions originariament descrites amb codi seqüencial.

En aquesta tesis es presenta una metodologia i un entorn d’eines que facilita la portabilitat d’un codi font seqüencial a la seva descripció en entorns de programació paral·lela, com OpenMP, MPI o HMPP. S’ha demostrat amb èxit que les eines faciliten aquesta tasca mitjançant exemples que posen de relleu com es redueix el temps dedicat per un programador per dur a terme aquesta transformació. Com a primer pas en la paralelització del codi es mostra una eina que ens ajuda a detectar dependències de dades en codi complexe. Un cop es detecta el paralelisme potencial, es pot anotar el codi amb pragmas OpenMP, però l’escalabilitat d’aquesta solució està limitada al nombre de processadors del node de computació. Es presenta l’eina OMP2MPI que mostra com generar automàticament codi MPI a partir de OpenMP per poder escalar fàcilment a un nombre major de processadors. Per altre banda, l’eina OMP2HMPP permet transformar codi OpenMP automàticament a codi HMPP, per a la seva execució en arquitectures GPU. A partir d’exemples concrets, es pot veure com el sistema permet transformar un codi seqüencial en OpenMP primer, i posteriorment MPI per obtenir una acceleració de 60× respecte el codi seqüencial. De manera semblant es demostra com es pot transformar automaticament un codi sequencial a OpenMP primer, i posteriorment en HMPP per obtenir una acceleració de 31× respecte al codi seqüencial i alhora augmentar l’eficiència energètica en un factor 5,86×.

Abstract

The demise of frequency scaling, which is the easiest way to improve computing performance, in addition to the growing gap between CPU and memory speeds and the increase in arithmetic intensity in current problems, has given rise to a new range of devices created to improve performance. Heterogeneous Computing (HC), and many-cores are examples of this new range of devices. However, the complexity of these new hardware architectures is not easily hidden from the programmer. In this thesis, I propose a set of tools that seek to exploit (through source-to-source (S2S) compilers) the capabilities and peculiarities of parallel computing and HC to speed up and increase the energy efficiency of originally sequential source code.

The proposed modular programs are implemented as a set of tools that help port sequential source code to OpenMP, MPI, and HMPP, demonstrating how the input code can effectively automatically be translated. Through a real-life example, I show how the proposed dependency analysis tool trivializes the task of parallelizing sequential code, breaking the first performance barrier. The OMP2MPI experiments generate code that is more than 60× faster than its sequential version and also faster than its original OpenMP code. The OMP2HMPP experiments obtain an average speedup of 31× and average increase in energy efficiency of 5.86×. Both tools were tested with OpenMP, obtaining successful results that demonstrate the feasibility of using this set of tools for exploring HC.

Acknowledgements

I am grateful to David Castells-Rufas and Jordi Carrabina for the care with which they reviewed this thesis as well as for conversations that clarified my thoughts on this and other matters. Their friendship and professional collaboration has meant a great deal to me. Thank you to all my family, specially to my parents because I am you. Toño, I know that you walk beside me every day and, Neus, thank you for being my compass and the spiritual support at critical and opportune times. To Marta, because you are my peace and my smile, you are the song that I want to hear every day, for all the time spent helping me yet being my sunshine. To Arturo, Cesc, Toni, and Miguel for all the shared lunches which gave me the energy to follow my research, the reviews of the final thesis proposal, and also for the comments that encouraged me to revise and improve it.

Contents

List of Figures

List of Tables

List of Acronyms

HPC	High-Performance Computing	1
GPGPU	General Propose Units for Graphic Processing Unit	7
GPU	Graphic Processing Unit	7
MPI	Message Passing Interface	xx
OpenMP	Open Multi-Processing	4
CUDA	Compute Unified Device Architecture	7
OMP2MPI	OpenMP to MPI	xiv
OMP2HMPP	OpenMP to HMPP	xiv
HMPP	Hybrid Multicore Parallel Programming	xiv

NoC	Network on Chip	7
MPSoC	Multi-Processor System on Chip	7
CPU	Central Processing Unit	1
FPGA	Field Programmable Gate Array	8
LLVM	Low Level Virtual Machine	12
SDSM	Software Distributed Shared Memory Architectures	15
AST	Abstract Syntax Tree	xiv
S2S	Source-to-Source Compiler	xiv
RPC	Remote Procedure Call	29
GOPS	Giga Operations per Second	47
TRMM	Triangular Matrix Matrix Multiplication	24
WTAG	Write Tag	73
SWTAG	Secure Write Tag	73
RTAG	Read Tag	73
FWTAG	Full Write Tag	74
FRTAG	Full Read Tag	74

IN	Input Variables	68
OUT	Output Variables	69
INOUT	Input Output Variables	69
DSIP	Domain Specific Instruction-Set Processors	8
ASIP	Application Specific Instruction-Set Processors	8
MCAPI	Multicore Communications API	8
HC	Heterogeneous Computing	ix
CSV	Comma-Separated Values	42
OpenCL	Open Computing Language	8
OpenACC	Open Accelerators	7
CTM	Close To Metal	7
API	Application Programming Interfaces	99
HWA	Hardware Accelerator	31
COBOL	Common Business-Oriented Language	11
CST	Concrete Syntax Tree	12
SIMD	Single Instruction Multiple Data	4

Introduction | 1

Over the years, the High-Performance Computing (HPC) community has tried to increase computer performance and overcome all the problems found in the computing process (Figure 1.1).

Some years ago, digital microelectronics circuits reached the frequency scaling limit. An increment in frequency leads to an increment in power density (in W/cm^2 according to Equation 1.1) that, in the early 2000s, reached a limit equivalent to the power density found in a nuclear reactor. This power dissipation problem made it impossible to continue to increment the frequency and limited computing performance, as shown in Figure 1.2.

$$P = C \times V^2 \times F. \tag{1.1}$$

Another problem is the continuously growing gap between Central Processing Unit (CPU) and memory speeds, as illustrated in Figure 1.3. This has been an important disadvantage for overall computer performance, as sometimes a processor is forced to stall while waiting for a memory operation to complete.

Finally, another important performance hurdle is the continuous increment in arithmetic intensity. Arithmetic intensity is defined as the number of floating-point operations needed to run a program divided by the number of bytes accessed in main memory [4]. Arithmetic intensity is found in recently proposed problems, which are more complex than old ones, but can be introduced into old problems by scaling the problem size.

The HPC community is attempting to overcome these problems by implementing new strategies that increase performance. At the same time, they continue to keep Moore’s Law [5] relevant by proposing new architectures that are able to surpass the limitations of conventional systems. Moore’s law is a self-fulfilling

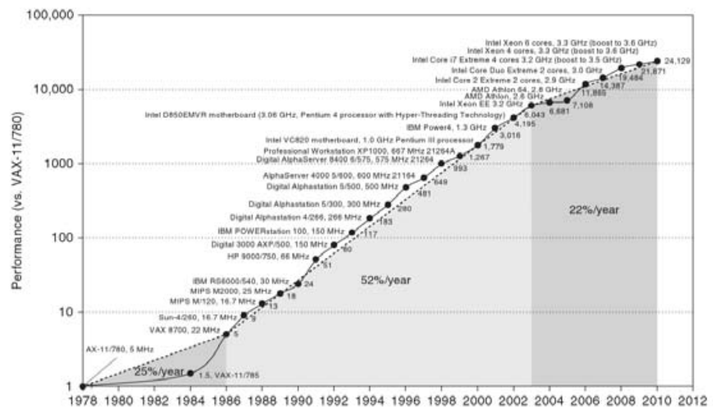


Figure 1.1 | Growth in processor performance since the late 1970s. Source: [1]

prophecy that states that, over the history of computing hardware, the number of transistors in a dense integrated circuit doubles approximately every two years, as illustrated in Figure 1.4. Initially, it was proposed in the form of an observation and forecast. Since it has become widely accepted, it has been converted into a goal for the entire industry. Moore’s Law, even with some pessimistic predictions, is still relevant. The aforementioned issues do not affect it, and the number of transistors is still doubled every 18 to 24 months.

Nonetheless, the HPC community has noted that with the end of frequency scaling, these transistors can no longer be used to increase frequency scaling, but can be used to add extra hardware, such as additional cores, to facilitate parallel computing. The clock frequencies of leading processors are now saturated, and architectural innovations are expected to keep raising the overall performance, such as via multi-core processing, which uses various cores to provide performance gains. This allows computing density to continue to double while reducing per-processor power consumption and heat.

In this context, parallel programming has become a necessary tool provided by the HPC community. However, parallel programming is complex for a variety of reasons. Applications are now no longer sequentially executed but divided into a more complex map composed of parallelizations. Further, parallel programs need to communicate among processors in order to coordinate certain tasks.

Parallel applications can be written using a variety of parallel programming paradigms, i.e., message passing, shared memory, parallel data, bulk synchronous parallel data, and mixed-modes. There are two de-facto standards for programing parallel code.

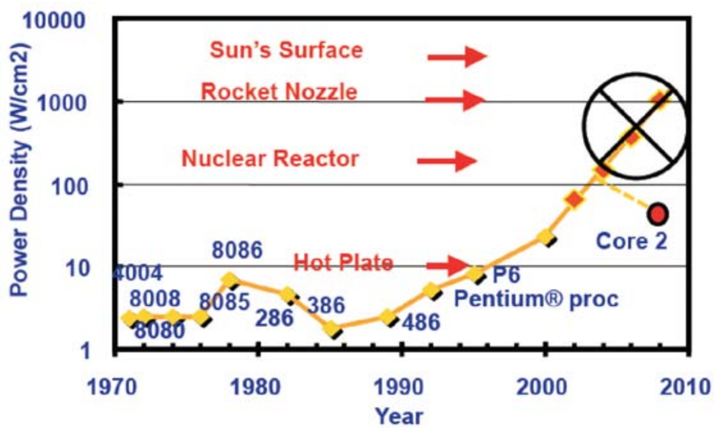


Figure 1.2 | Power density wall. Source: S. Borkar (Intel)

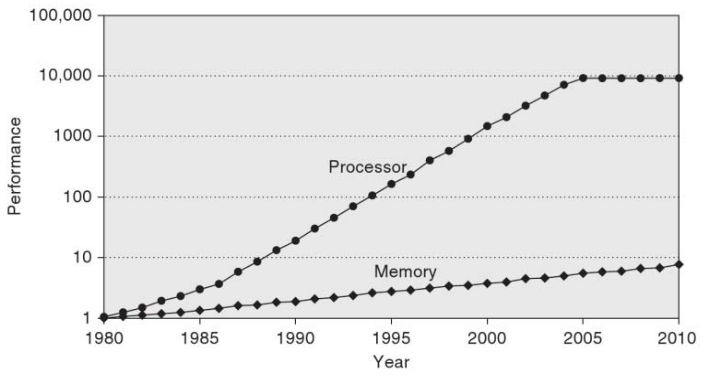


Figure 1.3 | Performance gap, measured as the difference in time between processor memory request (for a single processor or core) and the latency of the DRAM access, over time using 1980 performance as a baseline. Source: [1]

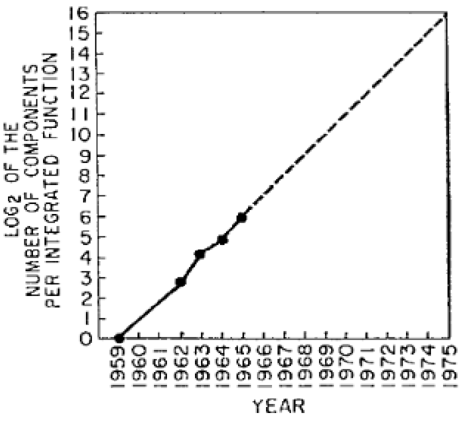


Figure 1.4 Moore’s 1965 prediction that the number of “minimum cost“ components on a chip would double each year, based on historical data and extrapolated to 1975. Source [2]

The first, MPI, advocates explicitly embedding communication primitives in the source code, making it difficult to read and maintain. The other, Open Multi-Processing (OpenMP), advocates a pragma-based approach that makes the compiler responsible for the efficient parallelism of the application through the use of a set of task division directives, as illustrated in Figure 1.5.

OpenMP version 2.0 primarily specifies ways to parallelize highly regular loops by using directives (shuch as parallel, do, section) that allow defining the blocks that have to be parallelized as well as the scope of the variables (either private or shared) of the variables inside these blocks. The standard offers other directives to define synchronization in a finer grain (such as critical, atomic) and includes some run-time functions that help implementing the necessary control of typical parallel program (like omp_set_num_threads, omp_get_threads_num).

Version 3.0 included more control with new directives, including the concept of tasks and task constructs. Version 4.0 includes support for accelerators, atomics, error handling, thread affinity, tasking extensions, user defined reduction, and Single Instruction Multiple Data (SIMD) support. One of the strengths of the OpenMP paradigm is the simplicity of its programming model. In this paradigm, the invocation of communication primitives are hidden from the programmer, as they are implicitly introduced by compilation directives working in conjunction with OpenMP at run-time. However, its use is usually limited to shared memory systems. Large HPC systems (such as the ones in the top 500 list) are often created by replicating nodes that contain some memory and a number of sockets

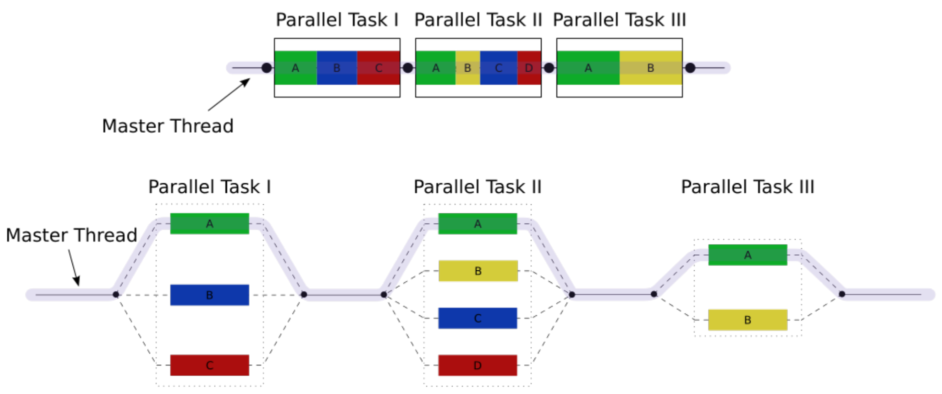


Figure 1.5 | Paralleled tasks using OpenMP blocks. Source: [3]

with multi-core processors or accelerators that can access that memory. Memory on remote nodes is not usually visible in the address space of applications running on one node. This makes OpenMP limited to the node domain. Thus, OpenMP applications are difficult to scale to a larger number of nodes (and cores) without introducing other paradigms such as MPI.

There are run-times that can overcome this limitation, usually by implementing software distributed shared memory (SDSM). They are also transparent to the programmer and, consequently, do not allow any fine tuning that could be needed to better adapt to the potential to different contexts. Moreover, they cannot be generally applied to all distributed memory platforms.

In contrast, MPI is a de-facto standard commonly used for large HPC applications. In this paradigm, the communication primitives must be explicitly coded. Introducing the communication primitives to implement the cooperation patterns makes the code larger and more difficult to read and understand. Obviously, it is more complex to learn because there are many functions, including point-to-point communication primitives as well as collective communication primitives. This coding effort is justified if it is needed for the execution of thousands of cores. MPI allows cores on different nodes to communicate. One might think that this introduces more performance overheads at the node level than OpenMP. However, this is a controversial issue with no clear answer, as shown in [6, 7]. Several versions have been presented since it was first published. Version 1.3 (commonly referred to as MPI-1) emphasizes message passing and has a static runtime environment. The following version, MPI-2.2 (MPI-2), includes new features such as parallel I/O, dynamic process management, and remote memory operations. The most recent

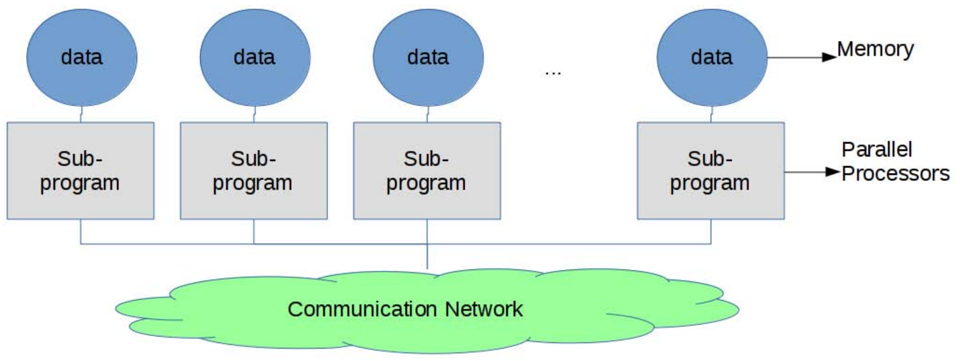


Figure 1.6 | Paralleled task using distributed memory architecture. Source: [3]

version, MPI-3.1 (MPI-3), includes extensions to the collective operations with non-blocking versions and extensions to one-sided operations[1]. Adoption of MPI-1.2 has been more extended during last years, essentially in HPC, but MPI-2.1 have been less repercusion and acceptance. We can see that many of the applications implemented using MPI-1.2 are using just a small subset of that standard, which has mean that there has been notreal need to add MPI-2 functionalities. The basic MPI functions (the complete set of functions can be found in [8]) and subroutines needed to divide tasks as illustrated in Figure 1.6. These can be divided in 4 main subsets: 1) Initialization and finalization functions to control the program flow, 2) Functions that define or report the logical topology of the distributed system (number of available slaves and execution IDs), 3) Data transference functions, dedicated to distribute data necessary for computation among the different nodes either synchronously or asynchronously . 4) Functions that control the communication process, to support, for instance the asynchronous operations (wait, test).

Another option that the HPC community uses to overcome frequency scaling, memory gap, and arithmetic complexity growth problems is to specialize processors for particular tasks, thus creating HC architectures. In those, specialized processors and conventional multi-cores will work together. Heterogeneous architectures have been presented as an architectural solution to all possible scenarios, and high-performance computers are increasingly based on HC architectures. The specialization of computational units and the adoption of different models of computation in various nodes increase system performance and energy efficiency.

[1]Summary extracted from:`https://en.wikipedia.org/wiki/Message_Passing_Interface`.

One example of these target architectures is the General Propose Units for Graphic Processing Unit (GPGPU). GPGPUs were proposed as promising vehicles for general purpose HPC and have become a popular part of such HC architectures because they obtain better speedup than parallel CPU versions in many cases. Accelerators such as NVIDIA Graphic Processing Unit (GPU) and Intel Xeon Phi are gaining market share: a 2013 survey found that 26% of systems have one or more accelerators installed [9].

HC allows a huge range of processing elements to be deployed in a single workflow and, at the same time, the platform that best fits each combination of elements can be selected. HC is different from network computing or high-performance distributed computing. However, even when MPI-based network computing is considered in order to exploit non-shared memory architectures such as distributed computing clusters, network computing can also be applied to HC because it can contain structures such as Network on Chip (NoC)-based or Multi-Processor System on Chip (MPSoC), as demonstrated in [10].

The OpenMP and MPI paradigms for parallel computing significantly increase the performance of a sequential application. The use of specialized processors on GPGPUs has been shown to achieve speedups of up to 100$\times$ those of conventional microprocessor architectures. Nevertheless, not all problems achieve the same performance increase, because any specific processor solution will be optimal for all kinds of computations.

There are two main barriers to HC use that might affect its wider adoption. First, the programming complexity required to distribute workloads across multiple processors is even more complex than in parallel computing. For example, for GPGPUs, a user must choose from a large number of alternatives proposed by the community, i.e., Compute Unified Device Architecture (CUDA) [11], HMPP [12, 13], RapidMind [14], Open Accelerators (OpenACC) [15], PeakStream [16], and Close To Metal (CTM) [17]. Even with these resources, programming a GPGPU is still complex compared to programming general-purpose CPUs, even when parallel programming models such as OpenMP [3] are used. This hurdle is combined with the additional effort needed to combine different kinds of processors. Neither issue is negligible. The potential advantages that an HC or parallel computing approach can provide have to be weighed against the cost and resources required to overcome them.

These complex and diverse scenarios open the door to new programming strategies that address them and ease the programming of emerging HC systems for non-experts. These new strategies will also be useful for expert programmers to reduce the time needed for testing and selecting the best method for performance and thus fully exploiting the capabilities of HC architectures. It is therefore essential for computer system researchers to facilitate these tasks. Researchers must offer new programming methods and tools to tightly integrate disparate computing elements on a platform with specialized processors while providing a programming path that does not require fundamental changes for software developers[18].

1.1 Motivation

Now that frequency scaling is no longer an easy way to improve computing performance, a new range of devices that exploit parallelism to drive performance have emerged. GPGPUs, and Many-Cores are examples of this new range of devices. However, the complexity of these new hardware architectures is not easily hidden from the programmer. To overcome this situation, many alternatives have been proposed, such as concurrent programming languages (e.g., Cilk and OCCAM), or extensions to existing languages (e.g., OpenMP [3], MPI [19], CUDA, Multicore Communications API (MCAPI), and [20]).

Parallel programming is not new. The HPC community has been active for a long time and usually works with a couple de-facto standards: MPI and OpenMP. The learning curve for new languages is usually steep, and it is desirable for legacy code to be executed on new platforms to take advantage of new capabilities.

The main motivation of this thesis is to ease the learning curve of parallel programming and HC architectures for the programmer. My strategy consists of letting users program in high-level C source code and provide some tools to refactor (or translate) the code into another form so that it can benefit from the specifics of the target architectures. The proposed tools fall into the category of S2S transformations, and are hence only a part of the compilation chain. Hence, they are complemented by standard front-end and back-end compilation tools. The expected benefits from these transformations are (1) better performance and (2) lower energy consumption.

Different types of parallelism can be exploited by this method depending on the targeted hardware architecture. For GPGPUs, source code transformations generally create kernels that have to be executed on the graphics accelerator. For homogeneous many-core accelerators (such as the Intel Xeon-Phi), we can combine different programming languages, models, and tools supporting Intel Architecture such as OpenMP, MPI, MCAPI, Open Computing Language (OpenCL), and Cilk. We also consider Application Specific Instruction-Set Processors (ASIP) or Domain Specific Instruction-Set Processors (DSIP). Related transformations can detect the instruction patterns for which the processors have been optimized and ensure that they are used. For devices such as Field Programmable Gate Array (FPGA) we consider HC that combines soft-core processors and custom logic so that the transformation uses custom logic as much as possible.

1.2 Objectives

The HPC community is addressing the abovementioned needs by creating new strategies to increase performance using HPC architectures that can work around the limitations of conventional systems. This thesis reviews the proposed methods and focuses on facilitating the task to fully exploit the capabilities of HC architecture and parallel computation paradigms.

The main objective of this thesis is to efficiently orchestrate the distribution of computing tasks and loads on accelerator and HC platforms by providing a diverse suite of tools. These tools allow solutions to be obtained with close-to-optimal performance for certain tasks, such as diverse algebraic level-3 applications, even when they are inherently different.

Because the current number of available target platforms is large, the scope of this research is limited to a representative subset: (1) parallel programming on shared memory architectures using OpenMP, (2) distributed memory architectures using MPI, and (3) HC using both CPUs and GPGPUs. A modular approach is used to build each of these three platforms. This modularity provides better adaptability to future input/output programming languages in a domain that is not yet stable.

The main idea is to build a set of tools that demonstrates the viability of automatically refactoring high-level C source code into the target programming languages. The proposed workflow is described in Figure 1.7.

OpenMP is defined as the common factor for all transformations because I consider it to be the simplest way to describe a parallel program while maintaining the structure of the sequential source code. OpenMP allows OMP2MPI and OMP2HMPP tools to work with the same input code. These tools generate new code that can be analyzed by time or an energy/time trade-off, if the target system is able to measure it. The code generated by these compilers for any selected architecture leads to more efficient implementations on heterogeneous platforms.

To reach this goal, the first step is to understand the differences between sequential and parallel code, especially to detect which parts of a sequential program can be parallelized without modifying the final result. These parts are mainly loops because the instructions inside them are repeated as sequential executions and therefore are fair candidates for large performance gains when distributed.

It is easier to determine whether simple (that is easy to understand) code can

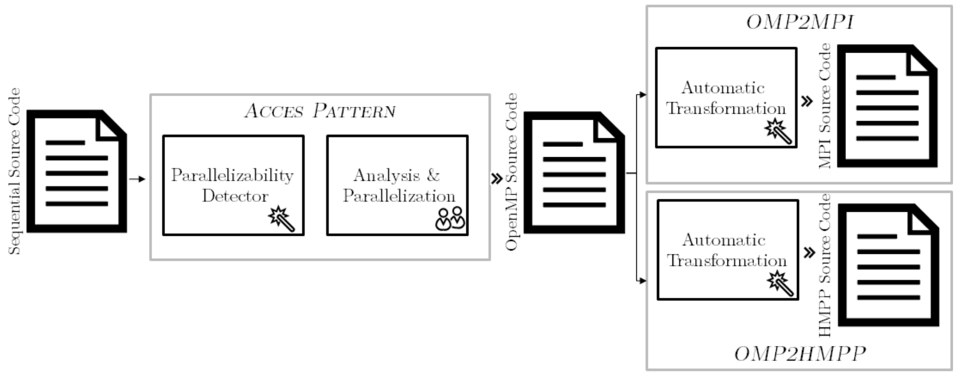


Figure 1.7 List of objectives for each of the three S2S compilers (Access Pattern, OMP2MPI, and OMP2HMPP)

be safely parallelized. However, determining this for more complex code that can include conditional clauses and function calls is not as easy. One of the fundamental roadblocks to parallelization is data dependency. When an operation depends on data that has to be previously computed, the execution of the operation has to be delayed until operands are available. Inter-dependencies between variables are very common in procedural programming languages, which were conceived for sequential execution such as C/C++, Java, and many others. The proposed tool facilitates the detection these data dependencies and illustrates them in order to clarify how the initial source code can be parallelized.

Once we have parallel code written in OpenMP, I offer two different tools that generate code able to fully exploit the capabilities of HC architecture, including parallel computation. The first focuses on GPGPUs. This new tool(OMP2HMPP) helps generate code for GPGPUs and combine it with CPU parallel code written using OpenMP.

In contrast, I also extend the use of parallel programming to distributed memory architectures through the use of MPI. I offer a tool that transforms OpenMP source code into MPI source code. The resulting code is able to execute in different kinds of DM systems, such as large HPC clusters or experimental distributed memory processors such as Intel Polaris, Ambric, or experimental FPGA based multi-soft-cores (such as [21]). Another potential use is to check if there is any performance gain when using MPI for coding a given application on the same shared memory platform or when combining it with OpenMP.

1.3 State-of-the-Art Technologies

As described in Section 1.2, this thesis proposes a new tool chain composed of three different modules. These modules are implemented as S2S compilers that transform sequential C/C++ source code so that it works in a different programming paradigm, i.e., parallel programming either on shared memory architectures, distributed memory architectures, or heterogeneous architectures that include GPGPUs.

1.3.1 Compilers

For many years, assembly code was the way to program code for computers. Each program was specific to a determined task and dependent on the kind of CPU used. With the invention of compilers, programmers where allowed to reuse their software on a variety of CPUs. These compilers became essentials in the process of software implementation. Hopper developed the first compiler, which was devoted to the A-0 programming language, in 1952. With that machine-independent programming languages adoption was extended, leading to the birth of Common Business-Oriented Language (COBOL), which is considered one of the first high-level programming languages.

However, although the most common reasons for compilers since then is to transform source code into a binary form to create an executable program or generate assembly language or machine code to produce a binary form, this thesis is mainly based on compilers that are not dedicated to this task. The extended meaning of compilers as used in the proposed tools is as a specific type of translator. This meaning encompasses the meaning of the original compilers as well.

There are six main extended operations in a compilation procedure: lexical analysis, preprocessing, parsing, semantic analysis (syntax-directed translation), code generation, and code optimization. More abstractly, compilation occurs in two phases. The first one (front-end), is dedicated to analyzing the correctness of the input code by breaking it into abstract symbols that represent lexical units (tokens). Analyzing these tokens, which must be ordered as the compiler expects, follows the description of the form that is defined for all corresponding different sequences of characters. This defines the nodes and allows semantic analysis. There are different intermediate representations of code: semantic graphs, control flow graphs, or the AST. The AST has control of the symbol table, which defines where any symbol in the source code is mapped including contextual information as the type, scope or location. The next step, the semantic analysis, is by definition more complex since it took the tokens as a group, generating sentences that have to mean together, it is normally hand written, even that in some cases can be partially or fully automated by the definition of productors of attribute grammars that encapsulate the possible group formations. We can subdivide the explained phases in subgroups: 1) lexing, that will allow scan and evaluation, 2) parsing, generate the Concrete Syntax Tree (CST), that are concrete being easy to generate and represent the input as a tree but are difficult to analyse, and 3) transforming, that translate it to an AST, more abstract, represents in a hierarchical data structure all the syntactic clutter, allowing more easy understanding for further analysis and translation stages.

This thesis focuses on understanding and modifying the AST. The AST represents the source code hierarchically as a tree of nodes that includes constants or variables (leaves) and operators or statements (inner nodes). A simple representation of a structure of the AST is shown in Figure 3.11. The AST is often used to generate the intermediate representation, also called an intermediate language, for the code generation. These intermediate languages enable the second phase of a compiler, the optimizations. We can describe many kinds of optimizations at this point, removal of useless or unreachable code, localization and propagation of constant values, relocation of computation to a less frequently executed place (e.g., out of a loop), or specialization of computation based on the context, are part of these. Finally, the code generation is optimized for a target system.

The number of phases used in a certain compiler allows us to classify them in different categories. We can think then, in different front ends for different languages combined back ends oriented to various CPUs, thanks to the separation of these phases. The extended GNU Compiler Collection, or the more new Low Level Virtual Machine (LLVM), are an example in which we can find this combinations. Those compilers combine multiple front-ends and multiple back-ends but they share analysis stage.

We jump over the complexity of parsing the syntaxes and semantics of the input source code and focuses on the S2S compilers or transcompilers. In this kind of compiler, either the input or output will be a high-level language. For S2S compiler infrastructure, there are many possible solutions, such as LLVM [22], PIPS [23], Cetus [24], ROSE [25], and Mercurium [26]. Mercurium supports C/C++ source code, which is the main reason it was chosen. As stated in [27], which describes most problems that can be found when parsing C++, this is not an easy task. Using the Mercurium infrastructure allows focus to remain on the creation of the S2S compilers by offering a friendly abstract representation through its intermediate representation (AST). AST provides easy access to the source code structure representation, the table of symbols, and the context of their context using a well-documented API that allows it to be further extended.

With access to the AST, implemented S2S compilers are able to modify it by adding new semantically correct trees while maintaining tree consistency. Our compilers are based on the premise of find and replace or reformulation. The compilers are able to find patterns, understand how they are used, and then generate code that works in the target architecture. One of the earliest S2S compilers was by Digital Research XLT86 in 1981. In this case, the compiler was able to transform .ASM code originally designed for the Intel 8080 processor into .A86 source code for the Intel 8086. There are other actual target-specific examples, such as CFRONT (C++ to C) and HIPHOP for PHP (PHP to C++), and others that are able to transform code to more than one target, such as LLVM [22], which can translate any language sported by gcc 4.2.1 to ADA, C, C++, Fortran, Java, Objective-C or Objective-C++, or from clang to C, C++, or MSIL.

The proposed S2S compilers are a part of the second subset running on top of the Mercurium platform. As explained in [28], this platform has been mainly used in the Nanos environment to implement OpenMP, but because it is quite extensible, it has also been used to implement other programming models or compiler transformations. Extending Mercurium is done using a plugin architecture, where plugins represent several phases of the compiler. These plugins are written in C++ and dynamically loaded by the compiler according to the selected configuration. Code transformations are implemented on the source code so that there is no need to know or modify the internal syntactic representation of the compiler.

1.3.2 Transformations

As was explained in [29], to fully utilize the power of heterogeneous machines or parallel programming paradigms, programmers must efficiently parallelize and map their applications. This task is far from trivial, leading to the need to automate this process. Automatically transforming sequential code into parallelized code is more feasible on a shared memory architecture. This process could be either partially addressed by tools that analyze the code to exploit parallelism or a fully automated process. Most of the proposed tools are of the first type, in which techniques are built-in in some parallelizing compilers, but the user needs to identify parallelizable code and mark it with special language constructs. This is also the case for the tool proposed in this thesis. The compiler identifies these language constructs and analyzes the marked code for parallelization. These parts tend to be loops.

From sequential to parallel code (parallelizing compiler)

Most compilers built for automatic parallelization research consider Fortran programs [30, 31, 24, 32]. Fortran cannot produce aliasing when identifying data dependency, in contrast to languages such as C and C++. Aliasing occurs when a data location in memory can be accessed through different symbolic names in the program. Thus, modifying the data through one name implicitly modifies the values associated with all aliased names, which may not be expected by the programmer. As a result, aliasing makes it particularly difficult to determine the data dependencies of a program. Aliasing analyzers try to compute information that is useful for understanding aliasing in programs. Therefore, C/C++ is generally difficult to analyze when pointers are involved. If there are further dependencies in the identified code sections, the possibilities for parallelization decrease. Nevertheless, a few compilers take C as input source code and transform these to OpenMP. One example is [33], which uses the parallelization methodology explained in [32] and applies it to C source code or PLUTO [34]. In both cases, the user has to use a set of new pragma directives defining the scope and mark the blocks of code that should be analyzed for parallelization. That tool generates a naive OpenMP version of the sequential source code when it detects loops that can be parallelized without modifying their form. Nonetheless, it has the drawback that it only transforms originally written C source code.

An experimental comparison using some of the tools mentioned above is presented in [35]. The automation of the parallelization process is neither optimal nor ready to become a widely adopted in practice, even for commercial compilers [36, 37], because of the complexity of changing from sequential to parallel code. Most cases require a more exhaustive analysis of the parts of the code with data dependencies and their adaptations to the parallel programming paradigms. Further work is needed to provide tools to programmers for the parallelization of sequential code. These tools demonstrate the variable access patterns and theory of data dependencies. Various examples are [38], [39], [40], and [41]. Examples of more complex representation are those proposed in [22] or the polyhedral model that produces graphics showing the dependency between iterations [42]. The main characteristic of our tool is that it tries to keep the visualization simple and at the same time analyze complex AST structures that could contain function calls that result in context switching.

From shared to distributed memory

The second tool of this proposal is dedicated to generating parallel code for distributed memory architectures using MPI. This tool uses generated code to generate solutions that can be executed using distribute memory architectures. Code can come from the access pattern and data dependency analysis extracted by the previous tool or from the code of existing OpenMP implementations that will be reused. Most projects that use OpenMP code for distributed memory architecture rely on the use of a software layer to manage data placements on the nodes (Software Distributed Shared Memory Architectures (SDSM) architectures). OMNI OpenMP [43] and its optimization (proposed in [44, 45]) are examples of alternatives to support OpenMP in a distributed memory environment using SDSM as an underlying run-time system. Cluster-enabled OMNI OpenMP on SCASH is an implementation of the OMNI OpenMP compiler for SDSM system SCASH running under score cluster system software. Another important software system is Cluster OpenMP, proposed by Intel [46] (although it was discontinued few years ago). All these solutions, based on the software layer, can be used on distributed architecture without using MPI but need some kind of run-time. In contrast, OMP2MPI shows the generated solution that will be executed on the cluster to the programmer. This solution can be further optimized if needed by an expert, thus offering more flexibility about how the code is executed in the cluster.

Other similar ways to port OpenMP programs to clusters were proposed in PaRADE [47], which is based on the OMNI compiler, or included in Polaris as [48]. Both combine the data management software layer with MPI primitives.

In [49–51], the authors proposed extending OpenMP with additional clauses for streamization, as I do. Nonetheless, the most similar tools to the ones proposed in this thesis were proposed in [52, 44] and [53]. The first is based on Cetus [24] and the second on PIPS [23].

Transforming for GPU use

I also explore the use of GPGPUs to increase performance in HC systems. The dominant GPU programming models have traditionally been CUDA [11] and OpenCL [54]). In recent years, many S2S alternatives have been proposed to overcome GPGPU programming complexity. Some of them are similar to the tool proposed in this thesis. In contrast to OMP2HMPP, their methods transform code to the CUDA language, not to HMPP, which means that CUDA programming complexity is directly exposed to the final user.

Some of the proposals extend, in one way or another, the current standards such as C/C++ and OpenMP to trivialize the programming task [55], [56]. In contrast, there are proposals that do not require any language extension to transform the source code directly from CPUs to GPUs.

One example that includes language extensions is proposed in [15]. CAPS, CRAY, Nvidia, and PGI (members of the OpenMP Language Committee) published OpenACC in November 2011. OpenACC has been proposed as the standard for directive-based standard programming, as it contributes to the specification of OpenMP for accelerators. In the same way, but not with the same consensus, a programming interface called OpenMPC was presented in [57]. This paper analyzes extensively the actual state-of-the-art OpenMP for CUDA S2S and CUDA optimizers. OpenMPC provides an abstraction of the complexity of the CUDA programming model and increases its automation though user-assistance tuning systems. It takes time to understand the newly proposed directives for both OpenMPC and OpenACC and to manually optimize the data transfer between the CPU and GPU. In contrast, OMP2HMPP adds just two new OpenMP directives, and the programmer does not need to deal with new languages and their underlying optimization. Another option, as explained in[57], is the hiCUDA directive-based language [58], which is a set of directives for CUDA computation and data attributes in a sequential program. However, hiCUDA has the same programming paradigm as CUDA. Even though it hides the CUDA language syntax, the complexity of the CUDA programming and memory model is directly exposed to programmers. Moreover, in contrast to OMP2HMPP, hiCUDA does not provide any transfer optimization. Finally [59] and [60] proposed an OpenMP compiler for hybrid CPU/GPU computing architecture. In these papers, the authors propose adding a directive to OpenMP in order to choose where the OpenMP block must be executed (CPU or GPU). The process is fully hidden from the programmer and is a direct translation of CUDA. Again, it does not provide any transfer optimization.

There are fewer proposals that try to directly transform C/C++ code into CUDA without the need for any new language extension. Ref. [61] presents a tool that uses unimodular loop transformation theory to analyze the loops that could be transformed to work in parallel kernels (either OpenMP or CUDA) through the ROSE compiler [25]. Par4All [33] transforms code originally written in C or Fortran to OpenMP, CUDA, or OpenCL. Par4All uses a polyhedral model to analyze and transform C/C++ source code. It also adds OpenMP directives where it thinks this would be useful. This transformation allows the re-factorization of the newly created OpenMP blocks into GPGPUs kernels by moving OpenMP directives into the CUDA language. However, this transformation does not take into account the kernel data-flow context and this leads to non-optimal data transfer results. Regardless, both tools, as well as the tools that transform sequential code to OpenMP, could be useful for generating input code for the proposed OMP2HMPP tool. All that is required is the substitution of the code generated after the access pattern study. This is because of the inherent modular work-flow of the tools proposed in this thesis.

Visualizing Data Access Patterns in Loops to Identify Potential Parallelism 2

The objective of the first S2S process in the chain (and its related tool) is to provide a way to refactor an input C/C++ source code into another form so it can benefit from the advantages of parallel computing on shared memory architectures using OpenMP, as illustrated in Figure 1.7. This process cannot be fully automated in most cases because of the task complexity, but the tool facilitates this task by providing essential information to the user. The tool detects parts of a sequential code that can be parallelized without modifying the final result obtained by the execution of the initial application.

Simple code can be easy to understand and determine if it can be safely parallelized; however, more complex code, which can include conditional clauses and function calls, is not so easy. The proposed tool is mainly devoted to detecting data dependencies in applications that can make the parallelization of the original source code prohibitive. Inter-dependencies between variables are terribly common in procedural programming languages that were conceived for sequential execution such as C/C++, Java, and many others. When an operation depends on data that has to be previously computed, its execution has to be delayed until all operands are available.

The proposed tool, introduced in [62], mainly focuses on loops, since the instructions inside them will be repeated in sequential execution and are fair candidates for obtaining large performance gains when distributed. Taking as an example a

```c
for (int i=0; i < 10; i++) {
        a[i] = k * b[i];
}
```

Table 2.1 | Simple C parallelizable source code

```c
for (int i=0; i < 10; i++) {
        if (i==0)
            a[i] = b[i];
        else
            a[i] = b[i] - a[i-1];
}
```

Table 2.2 | Simple C non-parallelizable source code

simple C input source code, it is easy to see that some such loops can be easily executed in parallel if they do not exhibit dependency, as illustrated in Table 2.1, where all ten iterations could be executed simultaneously without any problem. Every iteration is independent because the index of access for variables A and B is the actual value of the iterator used in the loop.

Changing the problem by adding an index access with a different value than the variable iterated, as illustrated in Table 2.2, shows iterations that should be executed sequentially. In this case, iteration i needs data from a former iteration. Note that in this case, the subtraction operation is irrelevant; any operation would produce the same data dependency. Such a data dependency could mean that this piece of code cannot be parallelized. Although in the proposed problems in Tables 2.1 and 2.2, the data dependency is easy to see, there are plenty of more complex problems for which it would be necessary to perform a more exhaustive analysis.

To trivialize the parallelization task, the proposed tool provides a trace log of all the variables accessed inside any marked loop of interest and provides a tool to visualize data dependencies in an easy way.

To do this, the implemented procedure is divided into three stages: (1) instrumentation by manually introducing some pragmas in the code to identify the section to be analyzed and using a S2S compiler to complete the instrumentation, (2) execution to create a memory access trace, and (3) visualization to view the data accesses pattern in the loops of the application to detect if there are dependencies that prevent them from being unrolled.

2.1 Automatic Instrumentation

To automatically introduce a log-in function into an input source code, the implemented tool requires source code to be marked using a new pragma directive. This new directive describes the information that needs to be analyzed inside the marked block.

The created directive is *analyze_access_pattern*, and it can be completed using two clauses: *var* and *iter*. Table2.3 shows an example of the use of this new directive using both clauses. The *var* clause determines the target variables in which data dependency could appear (marked red), and the *iter* clause determines the iterators targeted in the access log (marked blue).

Having defined the variables of interest (using the *var* clause) and the iterators (using the *iter* clause) in a loop that the programmer wants to analyze, the S2S compiler traverses the AST, detecting memory accesses and including the proper log functions in the detected points.

To generate a well-formed trace log, a set of log functions are defined. These functions are introduced automatically by a new S2S compiler inside the original source code. Later execution of the generated code provides a complete trace log that has the information required to detect all the possible access patterns and related data dependencies.

The set of proposed functions is divided by its functionality: (1) to understand the context, (2) to describe the variables of interest in that context, and (3) to identify how these variables are used.

Within a loop context, the user is interested in knowing how that loop is constructed. Analysis of the AST allows us to detect the desired loop structure. Hence, the following functions are provided:

```
void mem_trace_loop_start(char* var);
void mem_trace_loop_end(char* var);

void mem_trace_iter_start(char* var, int v);
void mem_trace_iter_end(char* var, int v);
```

These functions allow which accesses are done in which operation of which loop to be identified. Every loop that the user wants to analyze during the program is uniquely identified by a name. Iterations are identified by the iteration variable and its value. The first two functions are located before and after the loop of interest. The last ones are located inside the loop body, defining the context of every iteration.

To analyze the use of the variables and their access, we need first to define them by describing their dimensions and the place they occupy in memory. By knowing the size (number of dimensions and size of each), the base memory position, and the type size of variables, we can exactly track which variable elements are accessed. The proposed tool focuses on arrays because the access pointers to them could be more difficult to follow. Finding declarations is usually a difficult task, but, by exploring the AST context, we can detect global or local variables as well as dynamic pointers and their size at a certain point in the code. To describe this variable information, the following list of four functions is defined:

```
void mem_trace_def_array1d(char* v, int d1);
void mem_trace_def_array2d(char* v, int d1, int d2);
void mem_trace_def_array3d(char* v, int d1, int d2, int d3);
void mem_trace_def_mem(char* var, void* ptr, int typeSize,
                                        int varSize);
```

The first three functions are dedicated to determining the size of each of the dimensions for each array of interest (there is a limit to the number of dimensions that define a variable, but this value could be extended in future implementations). In contrast, the last trace function is used to provide memory information.

Finally, two functions are defined to log memory accesses. These functions distinguish two kind of accesses (read or write), and are inserted before any variable calculation expression. Read operations are placed first, followed by write operations. Expressions are usually the last node in an AST. To determine if a symbol is read or written, we analyze two factors in the considered expression. The symbols contained in the second operand of the expression are always considered read access. The first operand is determined by analyzing the operation of the assignment, since some cases can also contain a mathematical expression such as addition, subtraction, multiplication, or division. There is a particular case in which the compiler finds a function call with parameters for analysis for which the type of access cannot be determined at that point. In order to solve this issue, we inline the function, following the methodology explained in Section3.2.2. In order to add the analyzed information, the following set of functions are used for labeling the expressions:

```
void mem_trace_read(char* v, void* idx);
void mem_trace_write(char* v, void* idx);
```

Just the variable name and pointer are used. In fact, the variable name is redundant, since it could be derived from pointer information.

2.1.1 Results

Table 2.3 illustrates an example of some input code and Table 2.4 shows the source code generated automatically by the proposed S2S tool.

```c
double alpha;
double A[N][N];
double B[N][N];

int main(int argc, char** argv) {
  int i, j, k;
  int n = N;
  init_array();
  #pragma analyze_access_pattern var(A,B) iter(i,j)
  for (i = 1; i < n; i++)
    for (j = 0; j < n; j++)
      for (k = 0; k < i; k++)
        B[i][j] += alpha * A[i][k] * B[j][k];
  return 0;
}
```

Table 2.3 | Input code

2.2 Execution

This is the simplest stage of the proposed work-flow. The automatically generated source code, explained on Section 2.1, is executed. The execution extracts the log of memory accesses.

We encourage working with smaller workloads and thus with higher productivity. The execution process produces a memory overhead and a large demand for memory when visualizing the accesses. Nonetheless, the nature of the data access pattern algorithm is usually independent of the size of the data. For instance, in order to capture and analyze the data access pattern of a matrix multiplication that uses 1,000 $\times$ 1,000 matrices, it is sufficient to analyze the 10 $\times$ 10 case. This simpler case can provide the information required to discover the related effective parallelization strategy.

2.3 Data Analysis

The main principle is that all memory accesses should be aggregated in the same iteration. It makes no sense to look at individual read and write operations. What it is needed is understanding of how different iterations access different memory positions. Furthermore, before visualizing any data, collected information is aggregated so that it can be usefully presented.

Data analysis basically consists of collecting information about when read or write operations happen for each variable. In the case of arrays, this information has to be obtained for every element of the array.

```c
double alpha;
double A[N][N];
double B[N][N];
int main(int argc, char **argv) {
    int i, j, k;
    int n = N;
    init_array();
    mem_trace_def_array2d("A", N, N);
    mem_trace_def_mem("A", &A, sizeof(double), sizeof (A));
    mem_trace_def_array2d("B", N, N);
    mem_trace_def_mem("B", &B, sizeof(double), sizeof (B));
    mem_trace_loop_start("loop1");
    for (i = 1; i < n; i++) {
        mem_trace_iter_start("i", i);
            mem_trace_loop_start("loop2");
            for (j = 0; j < n; j++) {
                mem_trace_iter_start("j", j);
                    for (k = 0; k < i; k++) {
                        mem_trace_read("B", &B[j][k]);
                        mem_trace_read("B", &A[i][j]);
                        mem_trace_read("A", &A[i][k]);
                        mem_trace_write("B", &B[i][j]);
                        (B[i][j] += alpha * A[i][k] * B[j][k]);
                    }
                mem_trace_iter_end("j", j);
            }
            mem_trace_loop_end("loop2");
        mem_trace_iter_end("i", i);
    }
    mem_trace_loop_end("loop1");
    return 0;
}
```

Table 2.4 | Transformed code

Using the tracing functions added by our S2S compiler, it is possible to automatically track every iteration thanks to an added keyword. This keyword is composed of the name of the loop and the value of the iteration index. If a hierarchy of loops is analyzed, the keywords are formed by the combination of each iteration identifier.

The data analysis step demands a large amount of memory because it needs as many versions of the data access information as iterations (or iteration combinations) that occur at run time.

The example code shown in Table 2.3 has three loops that iterate variables i, j, and k, respectively. However, only i and j are considered because i spans from 0 to N and j spans from 0 to N. The number of iterations is N^2. On line 9 of Table 2.3, variables A and B are defined as the variables of interest. These $N \times N$ variables are tracked using data analysis. The tool needs $2 * N^2 * N^2 = 2 * N^4$ elements to track the application access pattern.

2.4 Visualization

The memory trace log generated by the execution and data analysis of our S2S tool on the input source should be observed using a visualization interface. In our tool, a new visualization interface is proposed to allow users to obtain a visual understanding of data dependencies and each read/write operation traced.

The proposed tool targets non-expert users. The task of refactoring a sequential program into its parallel version requires a set of rules to be defined to deal with the incorrect actions than these users can propose. A typical example is when a user analyzes a loop that contains many variables but cannot determine which ones are read-only and will not lead to variable conflicts (i.e., will not produce variable dependence). The visualization tool detects and discriminates such read-only variables by analyzing all intermediate matrices stored for that variable and detecting if there are no write operations. With this information, the visualization tool will propose to ignore them. All analyzed variables are shown in a check box list, and the ones that are detected as Read-Only are marked with an additional label, as illustrated in Figures 2.1 and 2.2. However, the tool keeps this information because it can be interesting to view the data access patterns of read-only variables. For instance, this is the case when we would like to have real data to mitigate specific problems such as cache misses.

Thus, the main idea around our visualization tool is to provide information to the user about which written memory positions are read in further iterations, indicating data dependency. The tool shows in a simple interface if an iteration in the same loop reads a previously written position.

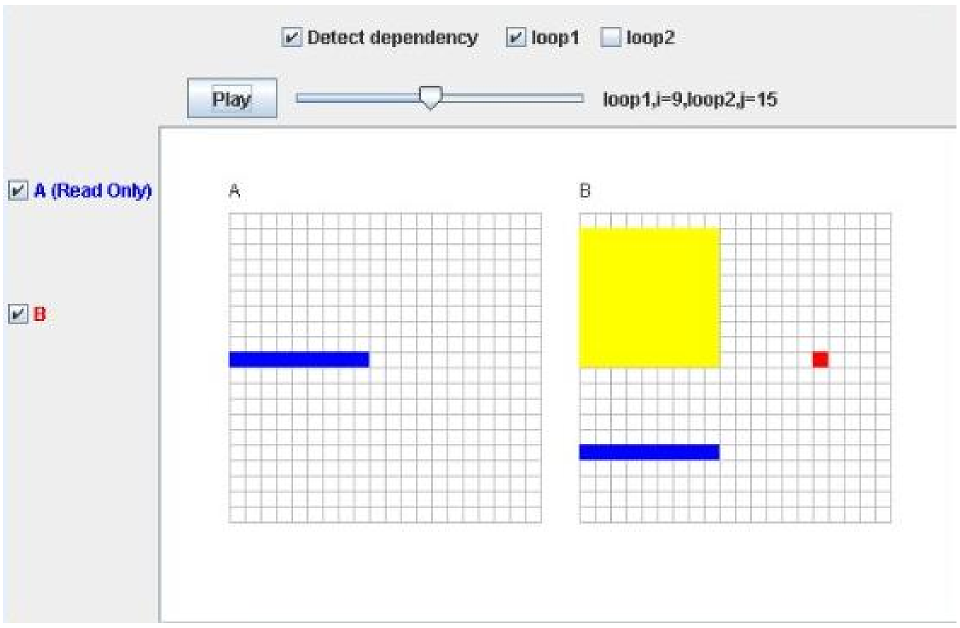
At the same time, it is also possible to observe if any read memory position has been written in a previous loop by maintaining a loop matrix. If the code reads a previously written position, the user should be notified. When any loop iteration finishes, the current iteration is consolidated with the previous one.

To more easily detect collisions, the tool displays the detected collisions. To do this, the matrix accessed at iteration i is consolidated as (i-1) in the next iteration. Comparing interaction matrices at execution time allows collisions to be indicated.

2.5 Case Study: Triangular Matrix–Matrix Multiplication

In order to illustrate how the presented set of tools (i.e., the S2S compiler and visualization tool) work together, I selected a problem from the set of codes proposed in the Polybench benchmark [63]: Triangular Matrix Matrix Multiplication (TRMM) is very suitable for analysis because of its peculiarities. The TRMM algebraic problem, illustrated in Tables 2.4 and 2.5, shows the code generated after adding the function traces during the S2S compilation process.

The trace log for further analysis is shown in Figure 2.1. When analyzing the loop 1 accesses of Table 2.5 (iteration i), it is clear that the memory positions of the elements of matrix B that were read during that iteration were written in previous iterations. Larger values of i mean that the number of read-after-write collisions is higher. It is also clear that loop 1 is not directly parallelizable.



loop1,i=9,loop2,j=15

Figure 2.1 | Original TRMM *loop1* visualization [1]

Loop 2 of Table 2.5 (iteration j) illustrates that collisions only occur when j is equal to i (as shown in Figure 2.2). When j is lower than i, the operations are computing the values of the first positions of the i-th row, which were not written in this iteration. When j is higher than i, the computed values are written for the last positions of the i-th row that were not read.

After analyzing this behavior, I propose a new calculation structure that restruc-

[1]Video available on: `https://youtu.be/iqi1LlDyuqI`
[2]Video available on: `https://youtu.be/p3z6AaIwuQw`

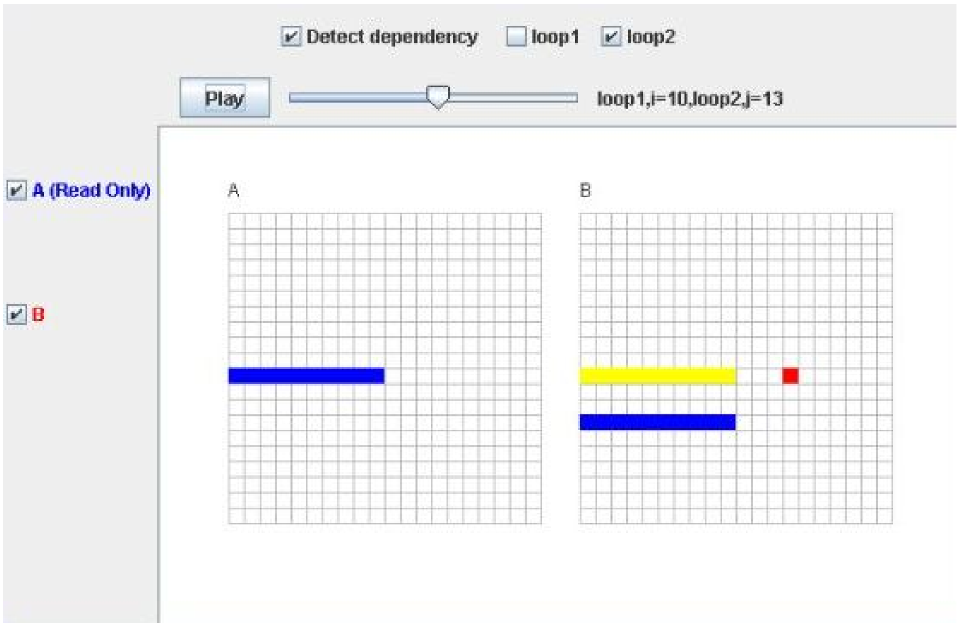


Figure 2.2 | Original TRMM *loop2* visualization [2]

tures the way the inner loop is computed. The resulting code is illustrated in Table 2.4. The new structure was designed to allow parallelization of the inner loops.

When the tool is executed again after this change, it shows that loop 1 still has dependencies, while loops 2 to 4 have no dependency and can be safely parallelized (as illustrated in Figure 2.3).

As a parallel version for the TRMM was readily available, it was tested by executing the resulting code on 1,000 $\times$ 1,000 matrices on 64 CPUs E7-4800 with 2.40 GHz (Bullion quadri module). The speedup obtained for the generated version was computed by comparing its execution time with the execution time of the sequential version. The results are illustrated in Figure 2.4, where the vertical axis shows the speedup for the TRMM example and the horizontal axis indicates the number of processors.

[3]Video available on: `https://youtu.be/SK1VYpoTSrU`

```
//loop1
for (i = 1; i < n; i++) {
  //loop2
  for (j = 0; j < n; j++) {
    for (k = 0; k < i; k++) {
      B[i][j] += alpha * A[i][k] * B[j][k];
} } }
```

Table 2.5 | TRMM original code

```
 // loop1
for (i = 1; i < n; i++) {
    // loop2
    for (j = 0; j < i; j++) {
        for (k = 0; k < i; k++) {
           B[i][j] += alpha * A[i][k] * B[j][k];
        }
    }
    // loop3
    for (j = i; j <=i; j++) {
        for (k = 0; k < i; k++) {
           B[i][j] += alpha * A[i][k] * B[j][k];
        }
    }

    // loop4
    for (j = i+1; j < n; j++) {
          for (k = 0; k < i; k++) {
             B[i][j] += alpha * A[i][k] * B[j][k];
          }
    }
}
```

Table 2.6 | TRMM rectified source code

2.6 Concluding Remarks

This chapter presents a set of tools proposed to allow users to easily analyze data dependencies in sequential source code. These tools are essential to allow efficient code transformation into equivalent parallel source code. The tool is composed of an S2S compiler and a visualization tool. Its power was shown on the TRMM problem from the Polybench benchmark. This tool will help users easily determine when it is useful to add OpenMP for directives in any sequential source code to obtain the correct results together with execution speedup. The number of parts in a sequential source code that could be analyzed for parallelization can be high. This could lead to a complex visualization of their access patterns, thus making the analysis in the visualization stage difficult. For this reason, I propose a new pragma directive that allows users to mark blocks of interest (mainly addressed to *for* loops) and to fix, using clauses, which variables are of interest in their study.

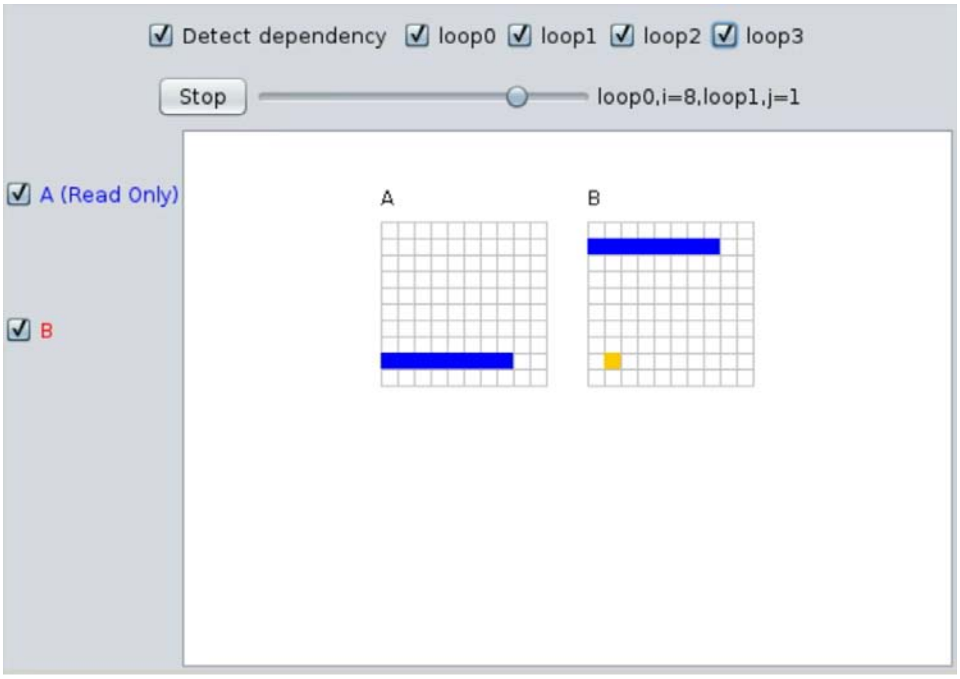


Figure 2.3 | Refactored TRMM visualization [3]

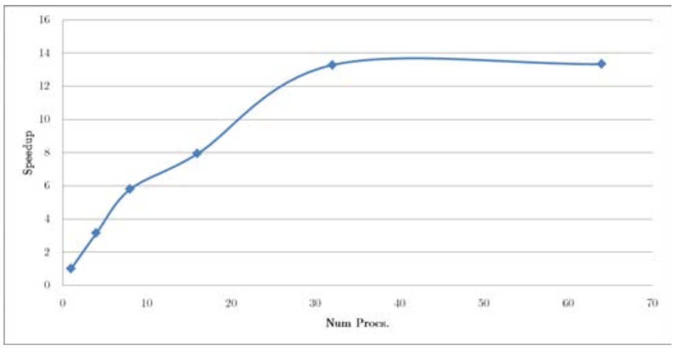


Figure 2.4 | TRMM problem speedup

However, there are still some aspects that can be improved. The size of the marked blocks can lead to a point where some problems are not scalable with respect to the visualization interface. This is the reason why, in the current version, the user has to minimize the size of the problem before the S2S compiler is executed.

Future versions will automate this task by automatically scaling the problem to its minimum number of feasible dimensions.

Another issue to consider refers to the fact that if non-expert users use the visualization tool, they can obtain more than the minimum amount of information required. Future versions will contemplate this option and instead of generating logs, the collision test will be passed to the execution stage instead of the visualization one. This feature will facilitate the test-error parallelization, in which the user will put OpenMP directives in loops and check the output error to validate their correct use.

OMP2HMPP: Compiler Framework for Energy-Performance Analysis of Automatically Generated Code 3

OMP2HMPP is a new tool that we propose for running specific sections of code on accelerators so that the program will execute them more efficiently on a heterogeneous platform. The new S2S compiler transforms an OpenMP input source code into HMPP, as already shown in Figure 1.7. OMP2HMPP was initially developed for the systematic exploration of a large set of possible transformations to determine the optimal one in terms performance (for both energy and time consumption). OMP2HMPP extends upon this with knowledge of the existing alternatives for GPGPU programming by studying the use of one of the most promising options yet developed: HMPP. HMPP offers the easiest way to migrate from sequential to parallel code because it is a directive-based language. The meta-information added to the source code of the application does not change the semantics of the original code, thus simplifying the kernel creation. In addition, it offers an incremental method for migrating applications by first declaring and generating kernels to later manage data transfers and next optimizing kernel performance and data synchronization. HMPP directives address the Remote Procedure Call (RPC) of functions or regions of code on GPUs and many-core accelerators as well as the transfer of data to and from the target device memory. Most existing alternatives rely on a stream programming style. However, a program written for a given platform cannot run on another one. HMPP takes a radically different approach. A HMPP application can be compiled with an off-the-shelf compiler and run without any specific run-time to produce a conventional native binary.

Moreover, thanks to its dynamic linking mechanism, an HMPP application is able to make use of either a new accelerator or an improved codelet without having to recompile the application source. In this way, it preserves legacy codes and isolates them from the frequent hardware platform changes that are typical for hybrid multicores, e.g., fast GPU architecture evolution. As mentioned above, programming with existing alternatives (including HMPP) for GPGPUs is still more complex than programming general-purpose CPUs and parallel programming models such as OpenMP, because of the related hardware complexity. For this reason, OMP2HMPP is thought to avoid this complexity for the programmer.

In order to simplify the transformation task, OMP2HMPP can reuse the existing source code that describes parallelism using OpenMP directives. The HPC community commonly uses two standards: MPI and OpenMP. OpenMP is better oriented towards supporting multi-platform shared memory parallel programming in C/C++. It defines an interface for parallel applications on a wide range of platforms: from desktops to supercomputers. As a natural extension, OpenMP could be combined with the use of HMPP to analyze and improve the performance and energy efficiency trade-off.

On this basis, I developed the OMP2HMPP tool to do the following:

- Run specific sections of the proposed code on accelerators. OMP2HMPP combines the use of CPU parallel code and GPU code so that the program can execute more efficiently on a heterogeneous platform.

- Systematically explore the large set of possible transformations from pragma-annotated code to determine the optimal transformation.

- Introduce GPGPU acceleration that can provide a good trade-off between performance and development effort (not necessarily an optimal solution), avoiding the creation of a new platform learning curve.

- Compare all the possible configurations ofOpenMP and HMPP in terms of time and energy spent for the execution of each code section.

3.1 HMPP Directives

The proposed tool is able to combine the following HMPP directives with the original OpenMP directives:

- **Callsite:** Specifies the use of a codelet at a given point in the program. Related data transfers and synchronization points that are inserted elsewhere in the application must use the same label.
- **Codelet:** Specifies that a version of the following function must be optimized for a given piece of hardware.
- **Group:** Allows the declaration of a group of codelets.
- **Advanced Load:** Uploads data before the execution of the codelet.
- **Delegate Store:** Downloads output data from the Hardware Accelerator (HWA) to the host. (This is the opposite of the advanced load directive.)
- **Synchronize:** Specifies waiting until the completion of an asynchronous callsite execution.
- **Release:** Specifies when to release the HWA for a stand-alone codelet or group of codelets.
- **No Update:** Specifies that the data is already available on the HWA. When this property is set, any further transfer will be done on the considered argument.
- **Target:** Specifies one or more targets platforms for which the codelet must be generated. This means that if the hardware and the corresponding codelet implementation for it is available, that one will be executed. Otherwise, the next target configuration specified in the list will be tried. In the current version, OMP2HMPP always uses CUDA. Tests are performed using a server without OpenCL support.
- **Group:** Allows the declaration of a group of codelets. The parameters defined in this directive are applied to all codelets belonging to the group.

HMPP directives are described as “meta-information“ added to the application source code. We consider that this meta-information is safe in the sense that it does not change the original code behavior. The directives address the RPC of a function as well as the transfer of data to/from HWA memory. With these directives, OMP2HMPP is able to create a version that, in most cases, rarely differs from a hand-coded HMPP version of the original problem.

3.2 S2S Transformations

The OMP2HMPP implementation is a pipeline consisting of S2S compiler transformations devoted to moving OpenMP code to HMPP. This pipeline has two compiler phases devoted to two different transformation steps: outline and inline phases.

The first phase (outline phase) transforms the pragma OpenMP block into an HMPP codelet and callsite. The second phase checks the scope of all the variables used in the codelet in order to solve any problems related to the use of global variables inside the HMPP kernels or with function calls not allowed in the device. At the same time, the second phase transforms inline function calls inside the codelet (inline phase). Figure 3.1 shows the workflow of the transformation process, which is detailed in the following sections.

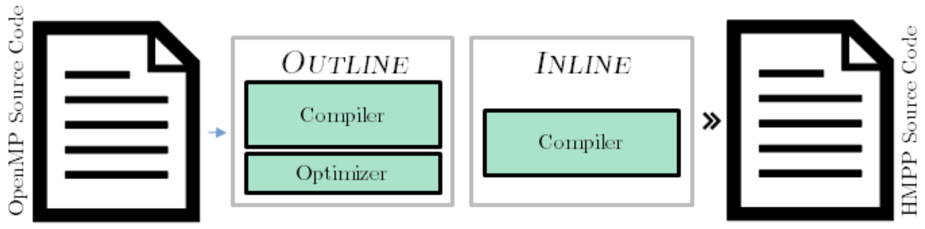


Figure 3.1 | S2S transformation process.

OMP2HMPP generates multiple implementations that differ in their use of different HMPP pragma parameter configurations. It then compiles and executes them, collecting the elapsed time and energy consumed during each execution. Results can be plotted to obtain the trade-off curves that allow the optimal working point to be selected. Figure 3.2 depicts the multiple execution and single plot processes that are performed after S2S transformation.

3.2.1 Outline Phase

The outline phase is responsible for the outline transformation (transforming OpenMP blocks to HMPP kernels and blocks). This phase finds all the OpenMP pragma instructions in the input code and then detects the start and end of the pragma block.

Once this detection is complete, the outline phase declares a function with the same functionality as the pragma block. Table 3.1 gives an example of the transformation

OpenMP

```
int main()
{
 ...
#pragma omp parallel for
 for(i=0;i<row;++i){
  for(j=0);j<col;++k){
   result[i][j] = 0;
   array[i*j] = mat[i][j];
   for(k=0;k<row;++k){
    a=0;
    while(a<10) {
     result[i][j] += mat1[i][k]*mat2[k][j]*array[i*j];
     a++
    }
   }
  }
 }
...
}
```

HMPP

```
#pragma hmpp __intr__for___ol__3__main codelet, target = CUDA,
             args[result,array].io=inout, args[array].size={row*col}, &
#pragma hmpp & args[*].transfer=auto
void __intr__for___ol__3__main(int i, int row, int j, int result[row][col],
      int *array, int mat1[row][col], int k, int a, int mat2[row][col]) {
for(i=0;i<row;++i){
  for(j=0);j<col;++k){
   result[i][j] = 0;
   array[i*j] = mat[i][j];
   for(k=0;k<row;++k){
    a=0;
    while(a<10) {
     result[i][j] += mat1[i][k]*mat2[k][j]*array[i*j];
     a++
}}}}}

int main() {
...
#pragma hmpp __inst__for___ol__3__main callsite
__instr__for___ol__3__main(i,row,j,col,result,array,mat1,k,a,mat2);
...
}
```

Table 3.1 Example of an OMP2HMPP S2S transformation process using a simple code as input.

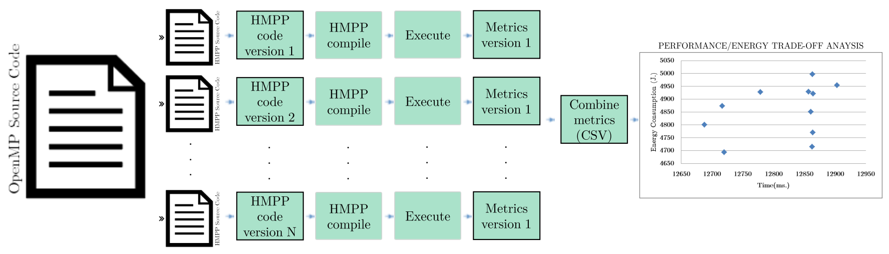


Figure 3.2 | S2S transformation workflow.

implemented in this phase. The code on top of the table shows the original code, while code at the bottom shows the code after the outlining transformations.

The outline phase is divided into two stages. The first one, the compilation stage, is liable for the S2S transformation, understanding the programmer's source code, and the generation of its different versions. The second stage, optimization, is devoted to improving the code proposed by the programmer.

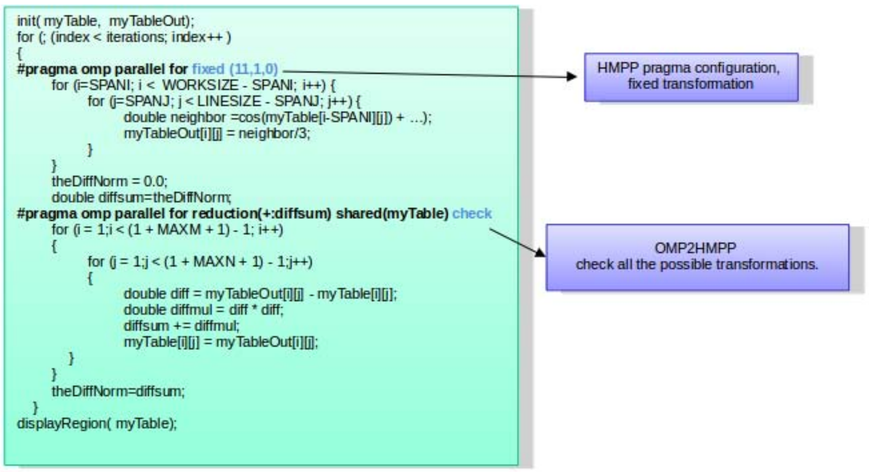
Compilation

The compiler has to deal with all the OpenMP standard directives, e.g., shared, private, and reduction. There are two possible scenarios: (1) OpenMP blocks are expressed as simple blocks where the OMP2HMPP tool transforms them into an HMPP codelet, or (2) they are expressed as a group of blocks and the compilation transforms OpenMP parallel blocks either into HMPP codelet groups (thus sharing variables if needed), or dividing them into simple blocks (thus specifying where each of these blocks will be computed). Table 3.2 shows how such techniques can be used to explore the capabilities of the HC architecture.

Every OpenMP block in the source code is transformed into its new version. OMP2HMPP explores all the possible HMPP configurations that can be used in these blocks and the combination of these configurations on different blocs. This implies that the number of generated versions grows exponentially with the number of OpenMP blocks to transform. It can be described as b^c, where b is the number of blocks and c the number of possible configurations per block. To solve this, two new OpenMP directives were created that allow the user to generate a smaller number of explorations than all possible versions that can be generated. These directives allow the programmer to explore the generated versions block by block. The set of new directives is described in the following list and demonstrated in Figure 3.3.

CHECK The OpenMP blocks with a pragma set "*#pragma omp parallel for **check***", are transformed using all possible HMPP configurations, including no transformation, which keeps the current OpenMP block as in the original code.

FIXED The programmer fixes a transformation for a certain block by setting information flags for the next execution of OMP2HMPP, i.e., "*#pragma omp parallel for **fixed** (10,1,0)*".



```
init( myTable,  myTableOut);
for (; (index < iterations; index++ )
{
#pragma omp parallel for fixed (11,1,0)
        for (i=SPANI; i <  WORKSIZE - SPANI; i++) {
                for (j=SPANJ; j < LINESIZE - SPANJ; j++) {
                        double neighbor =cos(myTable[i-SPANI][j]) + ...);
                        myTableOut[i][j] = neighbor/3;
                }
        }
        theDiffNorm = 0.0;
        double diffsum=theDiffNorm;
#pragma omp parallel for reduction(+:diffsum) shared(myTable) check
        for (i = 1;i < (1 + MAXM + 1) - 1; i++)
        {
                for (j = 1;j < (1 + MAXN + 1) - 1;j++)
                {
                        double diff = myTableOut[i][j] - myTable[i][j];
                        double diffmul = diff * diff;
                        diffsum += diffmul;
                        myTable[i][j] = myTableOut[i][j];
             }
        }
        theDiffNorm=diffsum;
   }
displayRegion( myTable);
```

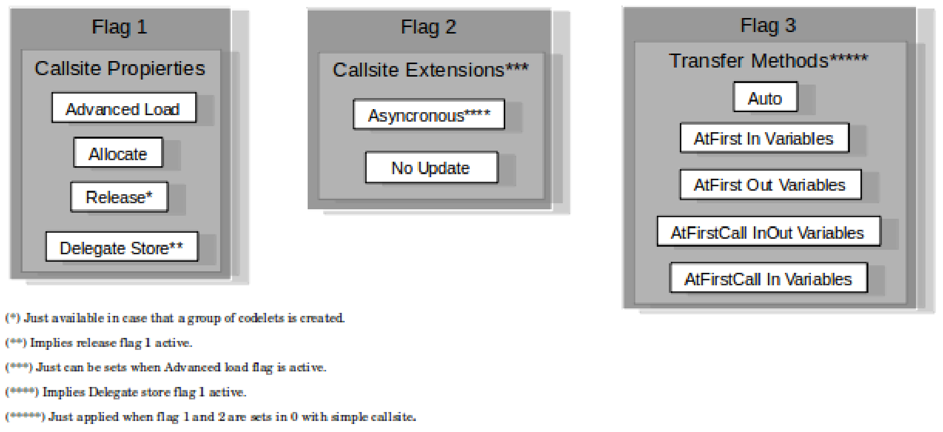
HMPP pragma configuration, fixed transformation
OMP2HMPP check all the possible transformations.

Figure 3.3 | Example of the location and explanation of new directives.

The programmer can explore all the possible configurations of HMPP for any given OpenMP block. The best configuration will be selected, as explained in Section 3.3. The selected configuration can be blocked for the following executions of OMP2HMPP using the FIXED directive. The FIXED directive is complemented with three flags, as described in Figure 3.4. These flags have an internal binary representation that is transformed into a decimal one to compress its length in the OpenMP pragma instruction specification.

As the detection of each variable is passed as value, copy, or reference (i.e., in Table 3.3), this allows variable context to be distinguished inside every block. OMP2HMPP has to treat array and matrix parameter passing differently. HMPP's distinction requirements impose detection when both of them are used, as shown in Table 3.1 for variables *result*, *mat1*, and *mat2* in the case of matrices or variable *array* in the case of arrays.

The compilation stage takes more control of how the outlined kernel division detects



(*) Just available in case that a group of codelets is created.
(**) Implies release flag 1 active.
(***) Just can be sets when Advanced load flag is active.
(****) Implies Delegate store flag 1 active.
(*****) Just applied when flag 1 and 2 are sets in 0 with simple callsite.

Figure 3.4 | FIXED directive flags explanation

the variables that define the two outer for-loops. Using this information, it defines the grid division using the “hmppcg gridify“ directive. Table 3.4 illustrates this case using an example.

As mentioned above, OMP2HMPP deals with all the OpenMP directives. A reference example is used to show how it works with the reduction directive. Reduction is a safe way of joining results from all threads after the construct phase. OMP2HMPP solves it by transforming the original code and simulating pass by reference with variables in reduction directive *diffsum*. The result of this transformation is shown in Table 3.4.

Optimization

This stage improves the code proposed by the programmer by exploiting context situations. The optimization reduces the number of transfers between the CPU and GPU by capturing and understanding the variable context of the OpenMP kernels.

In order to implement the optimization, OMP2HMPP must perform an accurate contextual analysis of the original code. To do this, OMP2HMPP studies contextual information, taking care of every array or matrix variable needed in each of the OpenMP kernels marked for transformation. Through this analysis, OMP2HMPP, is able to understand the following variable contexts:

- the kind of access (write/read)

- the host where it is used (CPU/GPU)
- the scope of the instruction where it is used (loop detection)

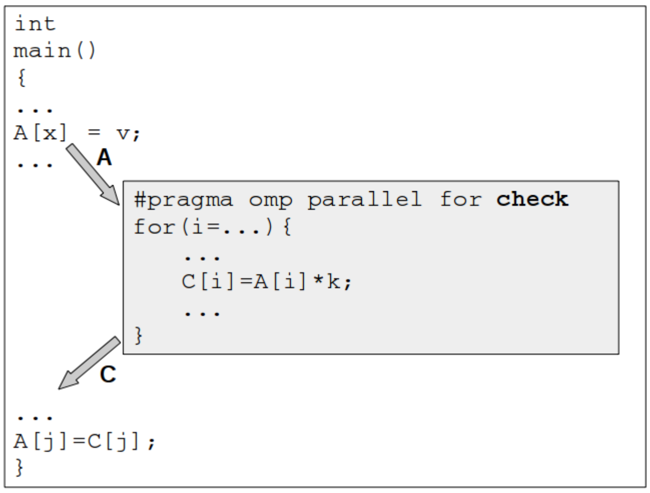

```
int
main()
{
...
A[x] = v;
...
        #pragma omp parallel for check
        for(i=...){
            ...
            C[i]=A[i]*k;
            ...
        }
...
A[j]=C[j];
}
```

Figure 3.5 | Context example

With this information, OMP2HMPP stores the contextual analysis for each kernel, as shown in Table 3.5 for the first of the marked pragma OpenMP blocks. The top code in Table 3.5 shows the original code while the bottom source code in Table 3.5 shows the different usages of each studied variable. The table illustrates the last read/write (if it exists) in the CPU and GPU before the OpenMP block and the first read/write (if it exists) in the CPU and GPU after the OpenMP block. These proprieties are specified in Table 3.5 by their labels: F (first), L (last), R (read), and W (write). The hosts are further specified as CPU or GPU. There are two additional parameters, as described in Table 3.5. The first one, inside parentheses, identifies the number of lines where the loop begins (if the use is actually inside a loop), and -1 (if it is not inside a loop). The last parameter is the line number in the actual context. Here, this number refers to lines of interest i.e., assignments, declarations, HMPP pragma, OpenMP blocks, and function calls. Nevertheless, the OMP2HMPP tool is capable of understanding pragma directives without variable usage information, as in the case of HMPP synchronization or the release directive. In these cases, OMP2HMPP searches the name of the callsite and looks for the call of this callsite in the AST with the same scope. Next, OMP2HMPP analyzes the kernel codelet in order to extract information on the variables implicated in the HMPP directives.

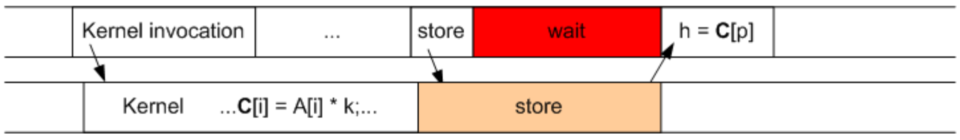


Figure 3.6 Delegate store directive optimization. Variables are downloaded as far as possible from the kernel finish, next to the first CPU read.

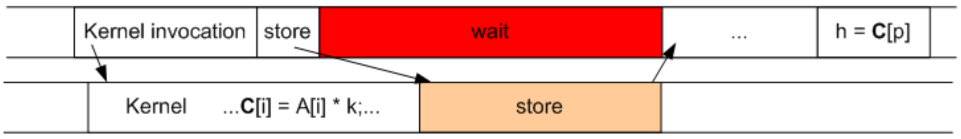


Figure 3.7 Delegate store directive optimization. Variables are downloaded when the kernel finishes.

Figure 3.5 shows a simple example of such context understanding. Variables A and C are used inside the OpenMP block. OMP2HMPP uses that information to select the best use of HMPP directives that minimizes the number of data transfers. In Figure 3.5, OMP2HMPP has two variables to analyze: A and C. In the case of A, it has to be uploaded to the GPU. However, there is no need to download it after the kernel call because it is not read until the end of the code. Variable C has to be downloaded from the GPU to CPU, but there is no need to upload it to the GPU because the kernel does not perform any read of C. With that information, OMP2HMPP uses an *advancedload* for A and places that directive as close as possible to the last write expression, as shown in Figure 3.8 to optimize the data transfer and improve the performance of the generated code. For C, OMP2HMPP places a *delegatedstore* directive as far as possible into the kernel call, as shown in Figure 3.6, thus increasing the performance of the generated code. Figures 3.7 and 3.9 illustrate the selection of an incorrect transfer policy using the same examples.

Moreover, OMP2HMPP can deal with context situations in which the source code contains nested loops. OMP2HMPP determines if an operation on a variable is performed inside a loop and adapts its data transfer to the proper context situation. Figures 3.10 and 3.11 illustrate an example of possible context situations. In the first figure, when OMP2HMPP wants to compute the loop in GPU, it needs to load the value of variable A, which is required to compute the value of C. Because the last write by the CPU of A is inside a loop with a different nested level than the GPU block, OMP2HMPP has to backtrack the nesting of loops to find the block shared

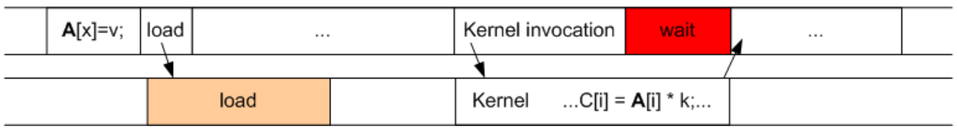


Figure 3.8 Advanced load directive optimization. Variables are loaded as close as possible to the last CPU write.

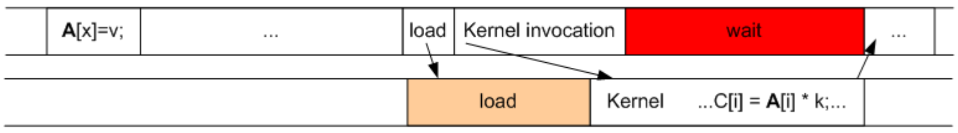


Figure 3.9 Advanced load directive optimization. Variables are loaded when the kernel is invoked.

by both loops. Then, similar to the process shown in Figure 3.5, OMP2HMPP optimizes the load of A by adding the *advancedload* directive as close as possible to the end of the loop. Figure 3.11 shows the opposite problem when changing the block that is computed in the GPU. In this figure, the result of the GPU kernel is needed by the CPU to compute C, which is not at the same loop level. In that case, the optimum way to add the *delegatestore* directive is just before the start of the nested loops, where the computation of C is located.

Table 3.6 shows a more complex example of loop context understanding. OMP2HMPP is able to understand complex context situations inside loops, thus detecting unnecessary transfer repetitions between CPU→GPU and GPU→CPU. OMP2HMPP understands that a previous read inside the same loop context (of the marked OpenMP block to transform) is at the same time the next read by unrolling the loop. This ability is shown in the first advanced load HMPP directive of Table 3.6, which is placed outside the loop where variables *myTable* and *myTableOut* are last used. This also affects the other appearances of the same directives in Table 3.6. In these cases, OMP2HMPP avoids uploading variable *myTable* in each iteration of the loop because the OpenMP block marked to be transformed is inside a loop and the previous CPU instructions read this variable. Table 3.6 also illustrates how OMP2HMPP downloads this variable from the GPU to the CPU after finishing the current iteration in order to update its value, which has been modified in the GPU.

The problem was extended to better understand of the use of the contextual information of the *group*, *mapbyname*, *noupdate*, and *asynchronous* directives. After the use of any variable by the GPU, if that variable is not written in the CPU

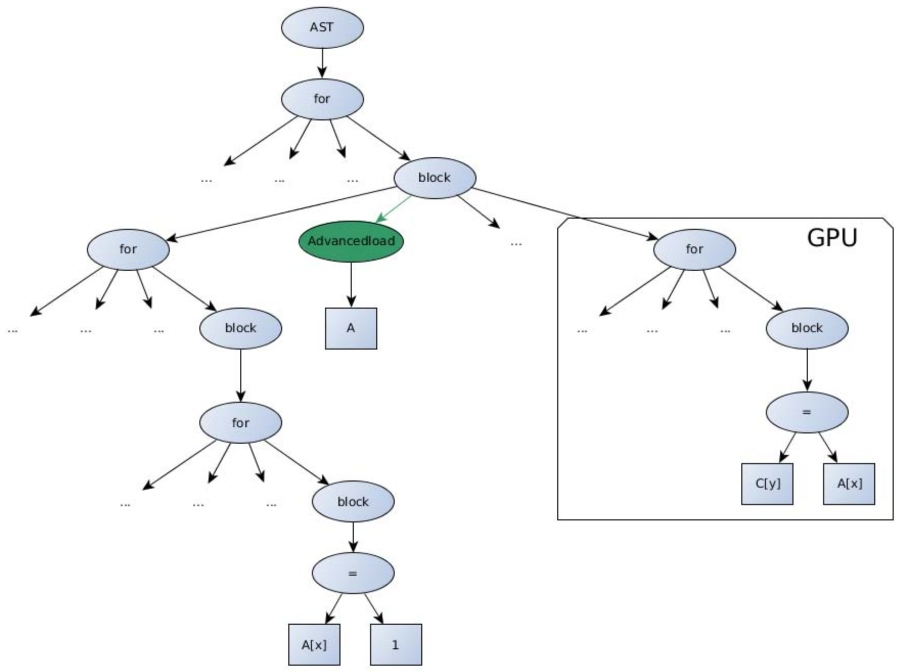


Figure 3.10 | Example of data transfer in loops

before the next read in GPU, the optimization phase will keep the variable in the GPU without downloading it to the CPU and automatically create a group of HMPP codelets using the *group* directive, even if it was not specified in the original OpenMP source code. The creation of this group allows both kernels to share this variable using just one load transfer CPU→GPU and one download transfer GPU→CPU with the use of the *mapbyname* directive.

OMP2HMPP includes the option of using asynchronicity in kernel invocation when it can be beneficial. OMP2HMPP extracts the information about the next usage of the kernel variables and adds the *asynchronous* HMPP directive. Finally, the *noupdate* directive is used to keep variables in the GPU that are not updated in the CPU, as shown in Table 3.6, which illustrates the kernel call, and in Table 3.7, which shows the codelet transforming these variables into input parameters. This transformation keeps the variables *myTable* and *myTableOut* in the GPU and performs just one load/download transfer. Figure 3.12 shows the difference between the synchronous and asynchronous calls of an HMPP codelet.

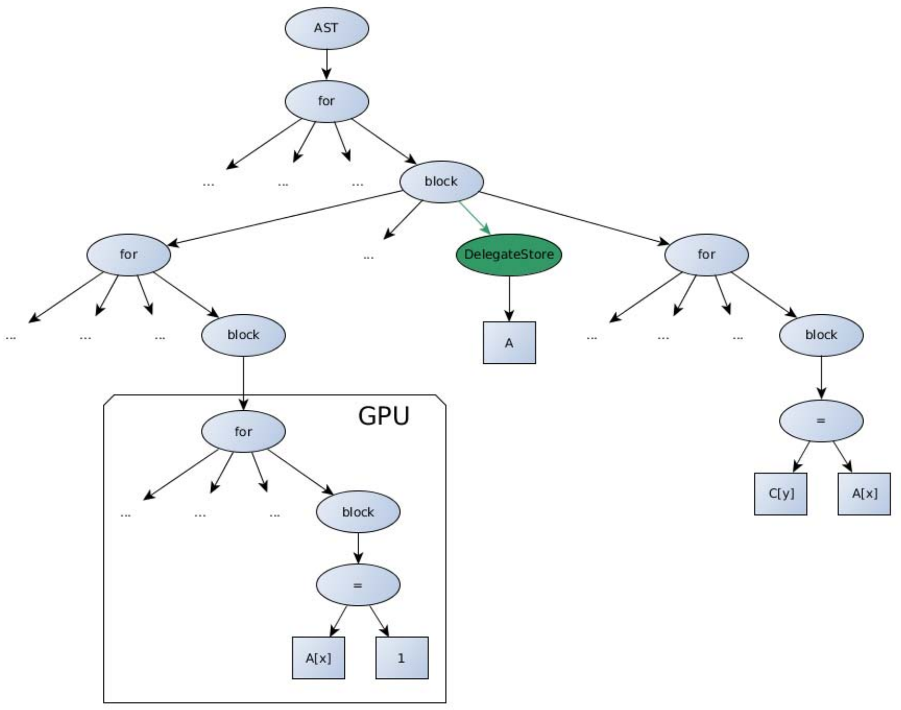


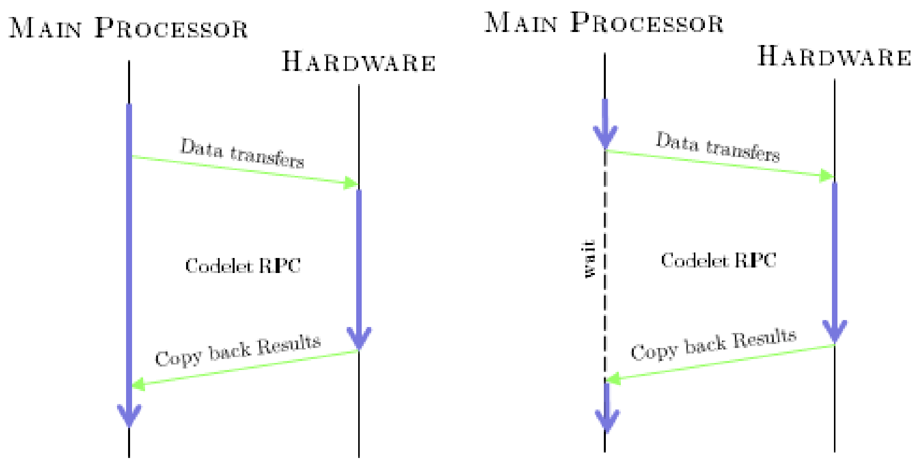
Figure 3.11 | Example of data transfer in loops

3.2.2 Inline Phase

This phase takes any function call in the input code and detects its variable and body declaration. It thus identifies the declaration and function parameters required to create a block of code that implements the same function. At the same time, this phase is responsible for checking the scope of the variable to detect any incorrect usage of global variables inside an HMPP codelet.

Table 3.8 shows an example of this procedure. The left side shows the original code, and the right shows the code after inline transformation.

A declaration of all the needed inline parameters is added to change their name inside this block according to the pattern name *__p__x__f__y*, where x is the position of the parameter in the function call, f is the name of the inlined function, and y is an index in order to avoid re-declaration. Index y increases each time a function is inlined. In addition, new variables *ret_fy*, *ret*, and *_return_y* are added to deal



(a) | Asynchronous **(b)** | Synchronous

Figure 3.12 | Asynchronous and synchronous differences in a codelet call.

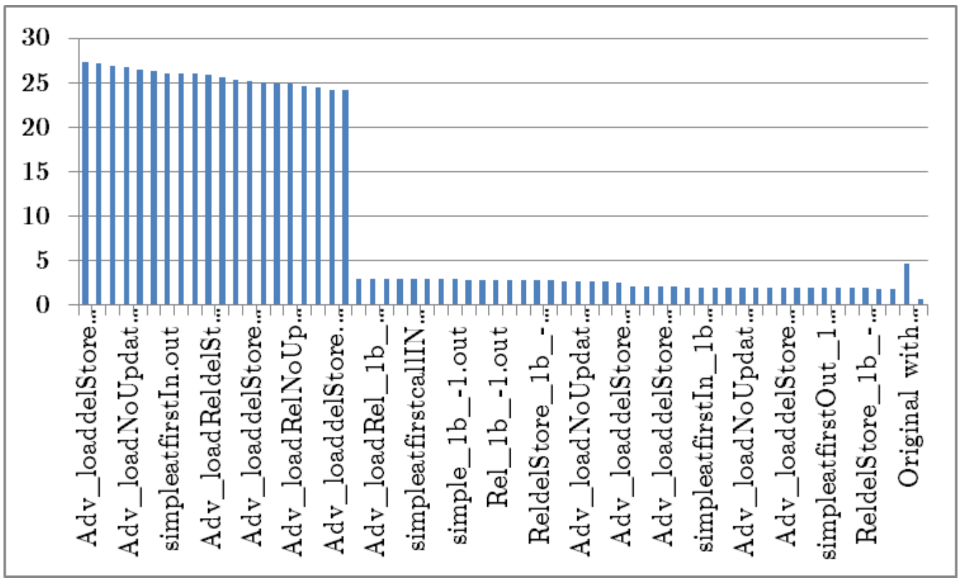
with the return parameters of the original function. The inline phase divides the expression that is formed by a mathematical expression of the results of a set of function calls into each function call. All results of the function calls are stored in *_return_y* variables and then the original expression is computed using these new variable values.

The global informative variables appear in the first lines of the code transformed by OMP2HMPP. These variables are created to inform the programmer which functions have been inlined.

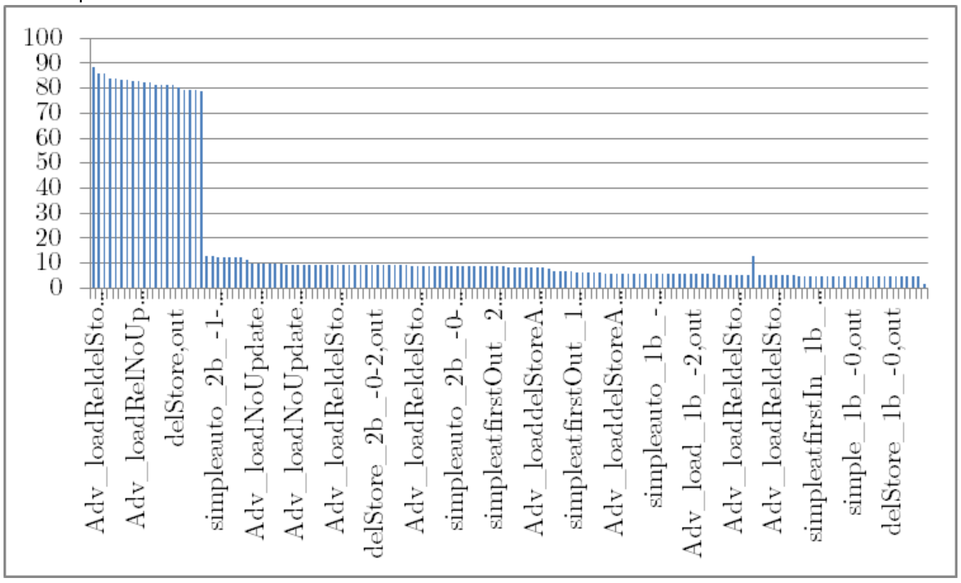
3.3 Results

The elapsed time and consumed energy for the parallel execution of the code with different options is presented in the report file generated by the OMP2HMPP tool. This file is a Comma-Separated Values (CSV) file that has all the information needed for further analysis. OMP2HMPP can perform several executions of each generated version on different input code to extract the median time and energy consumed by execution.

Table 3.9 presents example CSV results in a spreadsheet form. The first column

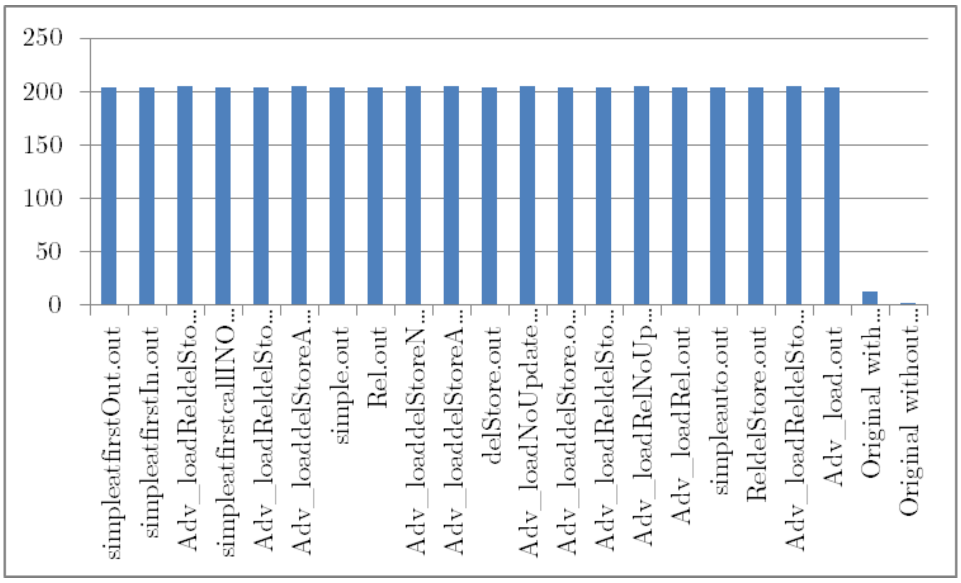


(a) | 2mm

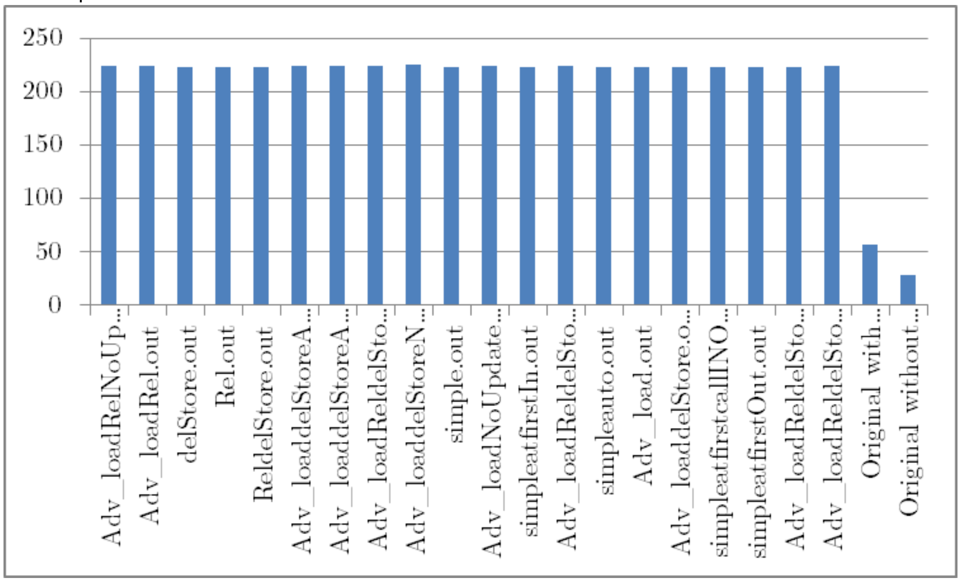


(b) | 3mm

Figure 3.13 | GOPS/W. Problem subset of the Polybench benchmark. (1/3)

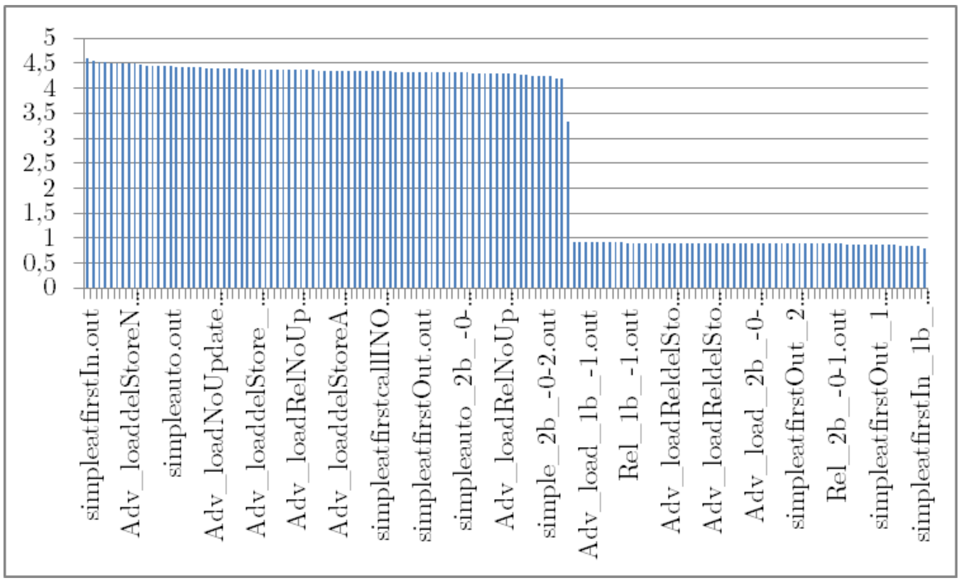


(a) | GEMM



(b) | LU

Figure 3.14 | GOPS/W. Problem subset of the Polybench benchmark. (2/3)



(a) Covariance

Figure 3.15 GOPS/W. Problem subset of the Polybench benchmark. (3/3)

shows the name of the generated file, the second column shows a unique signature (referred to the selected version of HMPP directives, OMP2HMPP FIXED directive, and useful for further executions), and in the following columns, the values for the time and energy measurements.

The generated versions were executed on a B505 blade equipped with 2 quad-core Intel Westmere-EP E5640 2.66 GHz CPUs, 24 GB of memory, and two Nvidia Tesla M2050 GPUs. This blade was equipped with energy meters that can be accessed by the Baseboard Management Controller and an embedded micro-controller that manages the blade. This microcontroller can power the blade on and off and measures its temperature, ventilation, and the energy consumed. The energy consumed by the components of the chassis (e.g., AC/DC transformer, chassis management module, interconnect switches) is not taken into account because these components are outside the blade. The energy consumption is obtained in Watt-hours (Wh) These values were converted to Joules by applying the corresponding conversion of $3,600 Joule = 1Wh$. The measured energy includes the computing units:

- Active CPU
- Idle CPU
- Memory
- GPUs

An example performance analysis of the code generated by OMP2HMPP, is shown on a set of code extracted from the Polybench benchmark [63]. The execution of the code resulting from OMP2HMPP was compared with the original OpenMP version and also with a hand-coded CUDA version and a sequential version of the same problem. Figure 3.16 compares the speedup for the selected problems. This figure shows that OMP2HMPP produces a good transformation of the original OpenMP code, obtaining an average speedup of 113×. In contrast, the hand crafted CUDA version achieves an 1.7× speed-up compared to the automatic generated programs. Moreover, the average speedup obtained when comparing the generated code to the original OpenMP version is 31×, which is a large gain in performance for a programmer that additionally does not need to have any deep knowledge about GPGPU programming.

OMP2HMPP measured the energy and time for all the generated code of all the benchmark problems. Figures 3.17, 3.18, 3.19, 3.20, and 3.21 show those measurements. These results allow the right implementation to be selected according to the desired working point. The upper plot in these figures shows the full set of generated versions and the bottom plot is an enlargement of the top one, showing in more detail the best ones. Figures 3.17, 3.18, 3.19, 3.20, and 3.21 illustrate that in some cases, there is a real trade-off between energy and time.

Finally, Figures 3.13,3.14, and 3.15 present the energy efficiency (in Giga Operations per Second (GOPS)/W) for all cases. These results show that the generated versions increase the number of operations per watt that can be obtained after the re-factorization performed by OMP2HMPP.

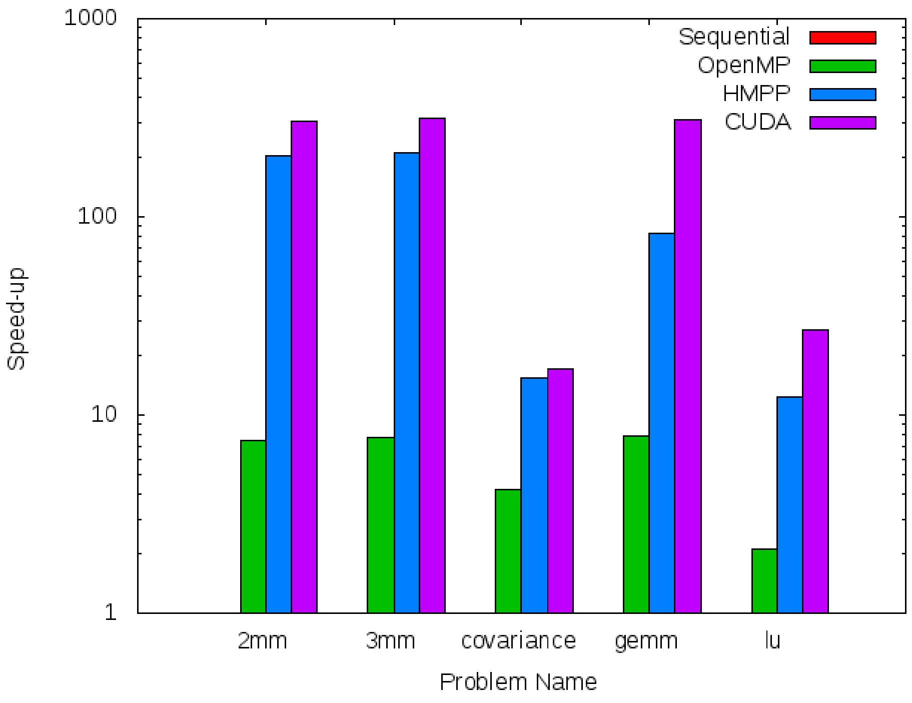
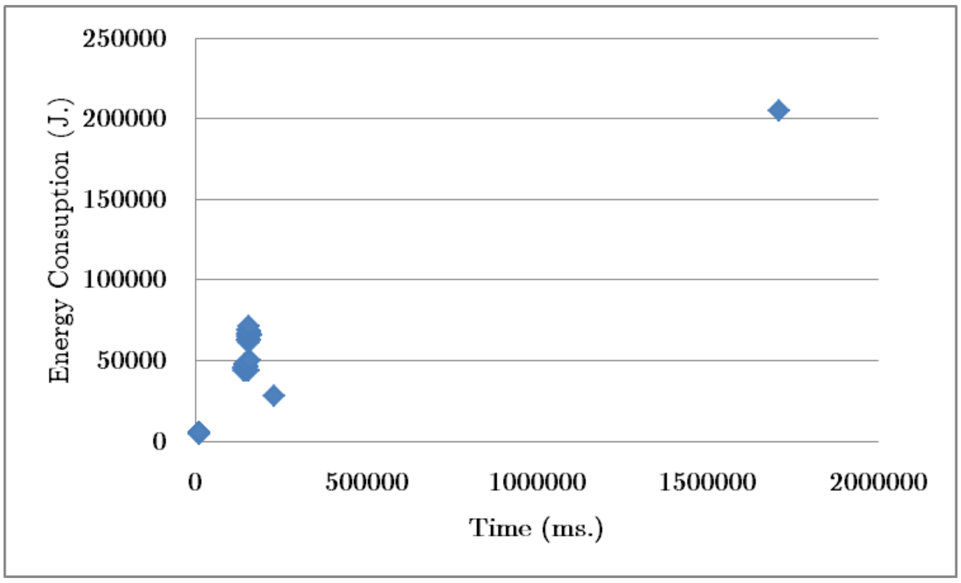


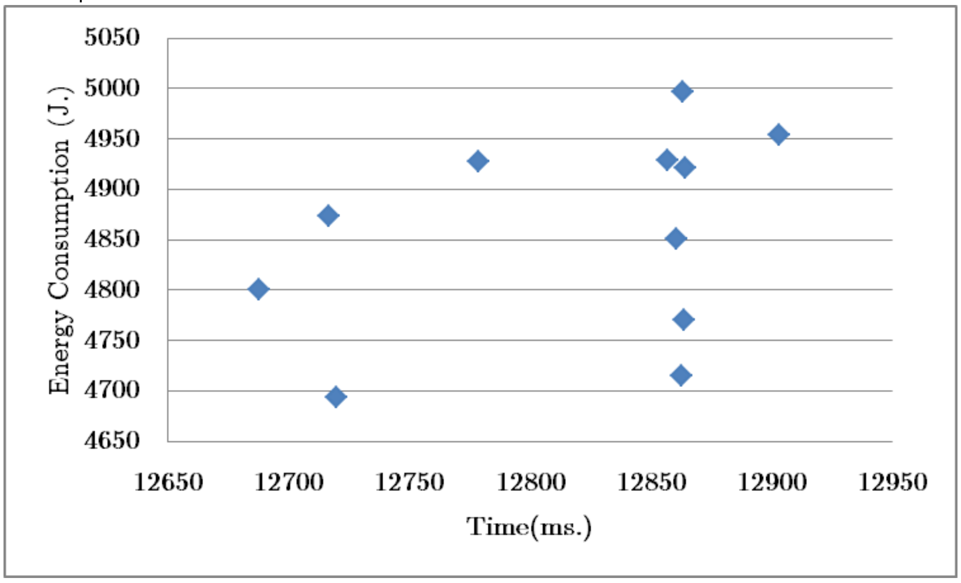
Figure 3.16 Speedup Comparison. Problem subset of the Polybench benchmark.

3.4 Concluding Remarks

I built an OMP2HMPP S2S compiler to provide a powerful tool for GPGPU programmers to facilitate the study of all the transformations that can be generated from an OpenMP block into any possible HMPP codelet and callsite. These transformations can be compared at the same time with the non-transformed version

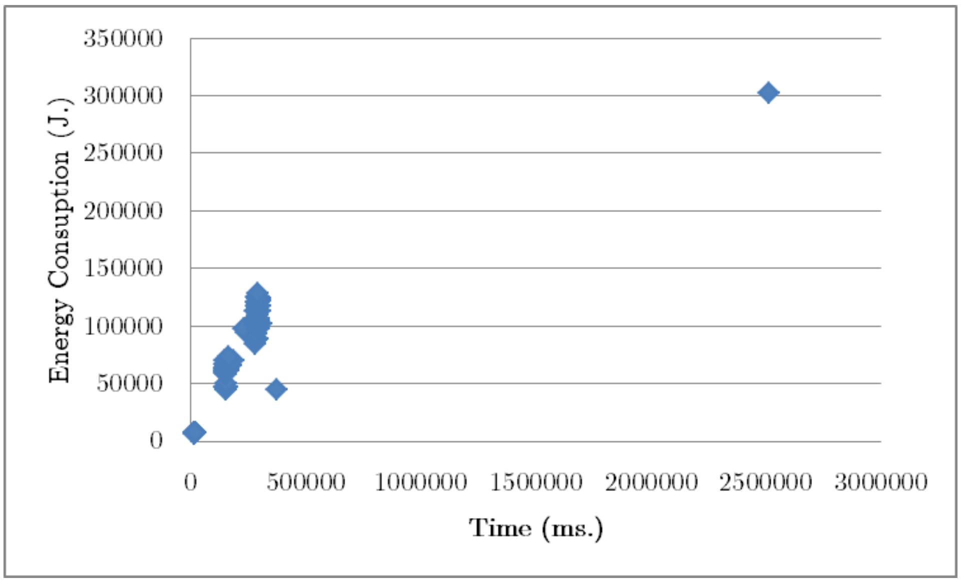


(a) | 2mm

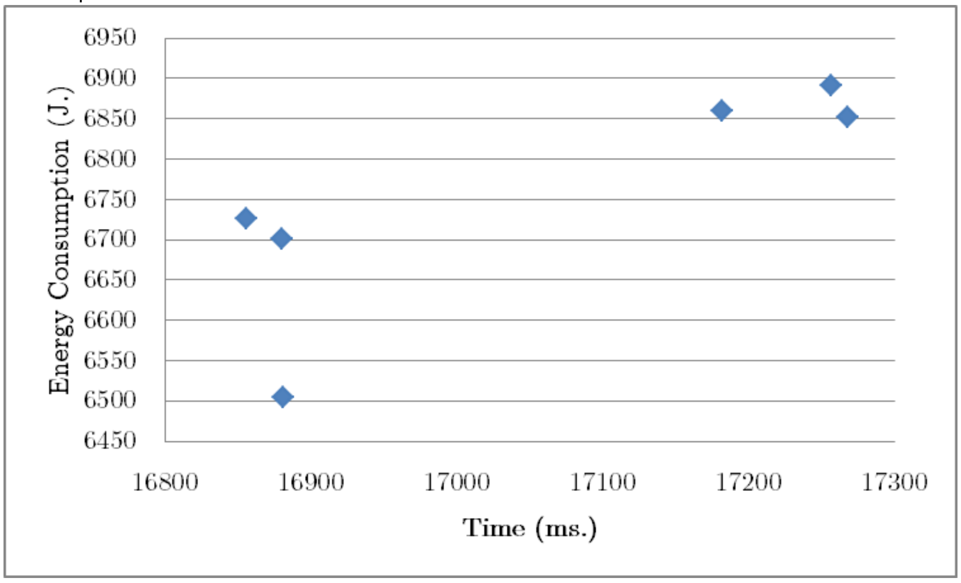


(b) | 2mm detail

Figure 3.17 | Energy/Time trade-off. Problem subset of the Polybench benchmark. (1/5)

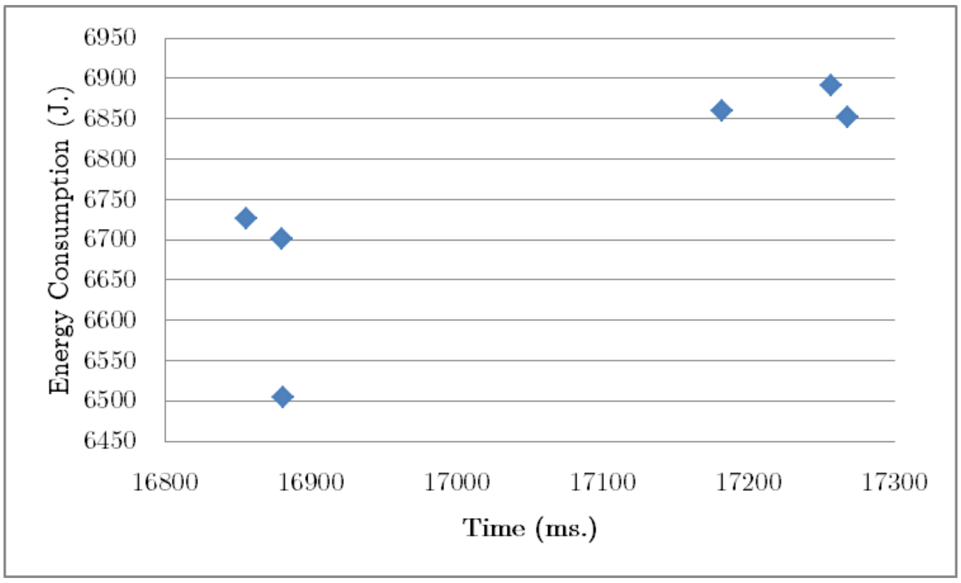


(a) | 3mm

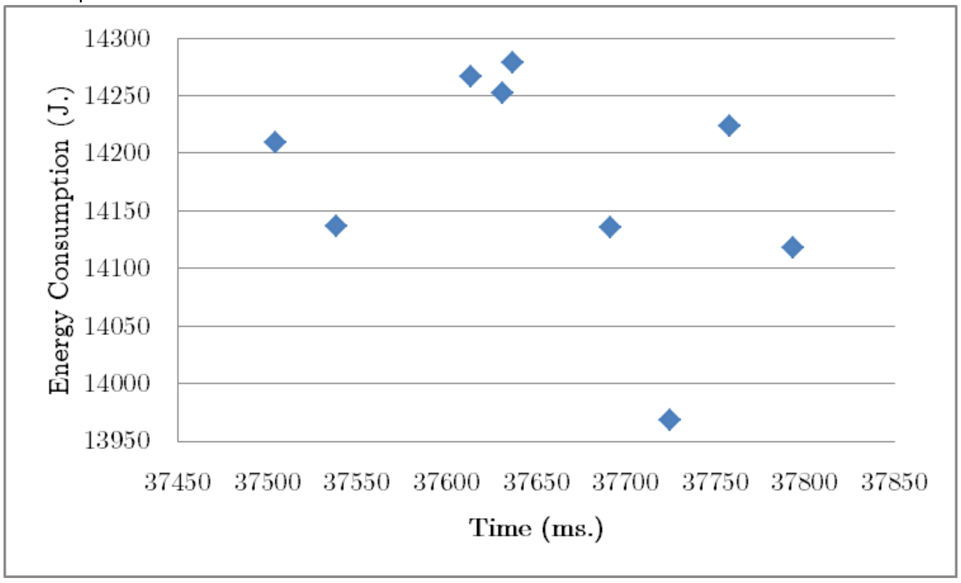


(b) | 3mm detail

Figure 3.18 | Energy/Time trade-off. Problem subset of the Polybench benchmark. (2/5)



(a) | Covariance



(b) | Covariance detail

Figure 3.19 | Energy/Time trade-off. Problem subset of the Polybench benchmark. (3/5)

of the proposed block that combines CPU and GPU using parallel shared memory computation (CPU and GPGPUs) exploring the use of HC. With the automatic transformations, the programmer avoids needing to learn the meaning of HMPP directives and, more importantly, obtains a good performance analysis. This allows the code implementation to be intelligently selected according to requirements. Using OMP2HMPP on the Polybench benchmark subset, I obtained an average speedup of 31× and an average increase in energy efficiency of 5.86× compared with the best results of the OpenMP version. The OMP2HMPP tool produces solutions that rarely differ from the best HMPP hand-coded version. Note that a CUDA hand-coded version obtains an speedup of near 1.7× compared with the best speedup of OMP2HMPP.

The automatic translation provided by our tool can be also useful for experimenting users that want to obtain, with minimal effort, a GPGPU code that can be used as starting point for further optimization. This staring point will provide a flavor of the code partitioning and mapping for any given problem, from which better performance with additional clever transformations can be obtained.

The current version of the tool has some limitations with the regard to the OMP2HMPP expansion. One of its limitations is that the actual version of OMP2HMPP generates a maximum of twenty-one possible configurations for each simple OpenMP block. The number of generated versions grows exponentially when trying to transform many simple OpenMP blocks at the same time or groups of OpenMP parallel blocks. This is because the versions that OMP2HMPP proposes either consider the possibility of full HMPP source code or create possible solutions that combine OpenMP and HMPP. Therefore, this results in a huge number of versions that can considerably complicate the final analysis. OMP2HMPP will solve this issue in future versions in order to minimize the execution time needed for a higher number of versions. This will be done by improving the optimization stage so that it deletes some redundant versions or a priori inefficient versions. OMP2HMPP will then propose the smallest set of possible versions with a smarter pragma combination of OpenMP and HMPP. OMP2HMPP will perform this task, separating every OpenMP annotated pragma into a new process generated using its outline and variable context. After that, each sub-program will be compiled and executed to establish if it is efficient (in terms of its potential to obtain good results) to optimize the CPU parallel block under a pre-established metric. This ability will save testing time, but OMP2HMPP will still present the versions needed to determine the best trade-off between execution time and energy consumption.

OpenMP

```
int main()
{
 ...
#pragma omp parallel shared(myTableOut,myTable) check
  for (; (index < iterations); index++ )
    {
    #pragma omp for
    for (i=SPANI; i <  WORKSIZE - SPANI; i++) {
      for (j=SPANJ; j < LINESIZE - SPANJ; j++) {
             ...
      }
    }
      theDiffNorm = 0.0;
      diffsum=theDiffNorm;
#pragma omp for reduction(+:diffsum)
     for (i = 1;i < (1 + MAXM + 1) - 1; i++) {
        for (j = 1;j < (1 + MAXN + 1) âĂŞ 1;j++) {
           âĂę
       }
     }
theDiffNorm=diffsum;
    }
#pragma omp parallel for reduction(+:diffsum) fixed(10,1,0)
    for (i = 1;i < (1 + MAXM + 1) - 1; i++){
        for (j = 1;j < (1 + MAXN + 1) - 1;j++){
           ...
        }
    }
displayRegion( myTable);
  return 0;
}
```

HMPP

```
int main()
{
#pragma hmpp <group1> group, target=CUDA
#pragma hmpp <group1> mapbyname, myTableOut
...
#pragma hmpp <group1> __instr__for4__ol__13__main advancedload, args[myTableOut],
                      args[myTableOut].addr="myTableOut"
    int a = 0;
    double diffsum = 0.0;
    for (;(index < iterations);index++)  {
#pragma hmpp <group1> __instr__for4__ol__13__main callsite,
                      args[myTableOut].noupdate=true
        __instr__for4__ol__13__main(i, j, a, myTable, myTableOut);
        theDiffNorm = 0.0;
        diffsum = theDiffNorm;
#pragma hmpp <group1> __instr__for4__ol__13__main delegatedstore,
                       args[myTableOut], args[myTableOut].addr="myTableOut"
#pragma omp parallel for reduction(+:diffsum) shared(myTable) private(i, j)
        for (i = 1; i < (1 + 5000 + 1) - 1; i++) {
            for (j = 1; j < (1 + 5000 + 1) - 1; j++) {
               ...
            }
        }
        theDiffNorm = diffsum;
    }
#pragma hmpp <group1> __instr__for4__ol__20__main callsite,
                      args[myTableOut].noupdate=true
    __instr__for4__ol__20__main(i, j, myTableOut, myTable, a, &diffsum);
    displayRegion(myTable);
#pragma hmpp <group1> release
    return 0;
}
```

Table 3.2 Different OMP2HMPP transformations on each of the OpenMP parallel for pragmas inside an OpenMP parallel block.

OpenMP

```
int main()
{
 ...
#pragma omp parallel for
 for(i=0;i<row;++i){
  for(j=0);j<col;++k){
   result[i][j] = 0;
   for(k=0;k<row;++k){
     result[i][j] += mat1[i][k]*mat2[k][j]*array[i*j];
    }
   }
  }
...
}
```

HMPP

```
#pragma hmpp _intr_for___ol_3_main codelet, target = CUDA,
            args[result].io=inout, args[mat1,mat2].io=in
```

Table 3.3 Example of variable access inside the kernel, as distinguished by OMP2HMPP, which determines if these variables are read or modified inside it.

OpenMP

```
#pragma omp parallel for reduction(+:diffsum) shared(myTable) check
for (i = 1;i < (1 + MAXM + 1) - 1; i++)
{
  for (j = 1;j < (1 + MAXN + 1) - 1;j++)
  {
    double diff = myTableOut[i][j] - myTable[i][j];
    double diffmul = diff * diff;
    diffsum += diffmul;
    myTable[i][j] = myTableOut[i][j];
  }
}
```

HMPP

```
#pragma hmpp _instr_for__ol_75_main codelet, target = CUDA,
             args[myTable].io=inout, args[myTableOut].io=in,
             args[diffsum_reduced].io=inout, args[diffsum_reduced].size={1}
void _instr_for__ol_75_main(int i, int j, double myTableOut[5002][5002],
                            double myTable[5002][5002],
                            double *diffsum_reduced)
{
    double diffsum = *diffsum_reduced;
#pragma hmppcg gridify(i, j), reduce(+:diffsum)
    for (i = 1; i < (1 + 5000 + 1) - 1; i++)
    {
        for (j = 1; j < (1 + 5000 + 1) - 1; j++)
        {
            double diff = myTableOut[i][j] - myTable[i][j];
            double diffmul = diff * diff;
            diffsum += diffmul;
            myTable[i][j] = myTableOut[i][j];
        }
    }
    *diffsum_reduced = diffsum;
}
```

Table 3.4 Example of grid division, as determined by OMP2HMPP using the *hmppcg gridify* directive. This example transformation shows an OpenMP block that contains an OpenMP reduction directive.

OpenMP

```
int main()
{

  int index =0;
  double theDiffNorm = 1;
  double RefDiffNorm = 0;
  int iterations = 99;
  int worksize=WORKSIZE, linesize=LINESIZE;
  int i,j,o;
  double diffsum,diff,diffmul;
  init( myTable,  myTableOut);
  for (index=0; (index < iterations); index++ ) {
#pragma omp parallel for shared(myTableOut) check
    for (i=SPANI; i <  WORKSIZE - SPANI; i++) {
      for (j=SPANJ; j < LINESIZE - SPANJ; j++) {
        double neighbor =cos(myTable[i-SPANI][j]) +sin(myTable[i][j-SPANJ])
                        +sin(myTable[i][j+SPANJ]) +cos(myTable[i+SPANI][j]);
        myTableOut[i][j] = neighbor/3;
      }
    }
      theDiffNorm = 0.0;
      diffsum=theDiffNorm;
#pragma omp parallel for reduction(+:diffsum) shared(myTable) check
      for (i = 1;i < (1 + MAXM + 1) - 1; i++) {
            for (j = 1;j < (1 + MAXN + 1) - 1;j++) {
                diff = myTableOut[i][j] - myTable[i][j];
                diffmul = diff * diff;
                diffsum += diffmul;
                myTable[i][j] = myTableOut[i][j];
            }
        }
        theDiffNorm=diffsum;

    }
  displayRegion(myTable);
  return 0;
}
```

HMPP

```
myTableLR(CPU)(-1): 9 -> init(myTable, myTableOut);
myTableLW(CPU)(-1): 9 -> init(myTable, myTableOut);
myTableFR(GPU)(64): 16 -> #pragma omp parallel for reduction(+:diffsum)
                                                shared(myTable) check
for (i = 1;i < (1 + 5000 + 1) - 1; i++)
{
    ...
}
myTableFW(GPU)(64): 16 -> #pragma omp parallel for reduction(+:diffsum)
                                                shared(myTable) check
for (i = 1;i < (1 + 5000 + 1) - 1; i++)
{
    ...
}
myTableFR(CPU)(-1): 18 -> displayRegion(myTable);
myTableFW(CPU)(-1): 18 -> displayRegion(myTable);
---------------------------
myTableOutLR(CPU)(-1): 9 -> init(myTable, myTableOut);
myTableOutLW(CPU)(-1): 9 -> init(myTable, myTableOut);
myTableOutFR(GPU)(64): 16 -> #pragma omp parallel for reduction(+:diffsum)
                                                   shared(myTable) check
for (i = 1;i < (1 + 5000 + 1) - 1; i++)
{
    ...
}
```

Table 3.5 | Example of OMP2HMPP use of contextual analysis.

OpenMP

```
int main()
{
  int index =0;
  double theDiffNorm = 1;
  double RefDiffNorm = 0;
  int iterations = 99;
  int worksize=WORKSIZE, linesize=LINESIZE;
  int i,j,o,a;
  double diffsum,diff,diffmul;
  init( myTable,  myTableOut);
  for (index=0; (index < iterations); index++ )
    {
 #pragma omp parallel for shared(myTableOut) check
    for (i=SPANI; i <  WORKSIZE - SPANI; i++) {
      for (j=SPANJ; j < LINESIZE - SPANJ; j++) {
        double neighbor=cos(myTable[i-SPANI][j]) +sin(myTable[i][j-SPANJ])
                       +sin(myTable[i][j+SPANJ]) +cos(myTable[i+SPANI][j]);
        myTableOut[i][j] = neighbor/3;
      }
    }
      theDiffNorm = 0.0;
      diffsum=theDiffNorm;
      a=0;
    #pragma omp parallel for reduction(+:diffsum) shared(myTable) check
      for (i = 1;i < (1 + MAXM + 1) - 1; i++)
        {
            for (j = 1;j < (1 + MAXN + 1) - 1;j++)
            {
                diff = myTableOut[i][j] - myTable[i][j];
                diffmul = diff * diff;
                diffsum += diffmul;
                         a=2;
                myTable[i][j] = myTableOut[i][j];
            }
        }
        theDiffNorm=diffsum;

    }
 displayRegion( myTable);
  return 0;
}
```

HMPP

```
int main()
{
#pragma hmpp <group0_12> group, target=CUDA
#pragma hmpp <group0_12> mapbyname, myTable,myTableOut
    int index = 0;
    double theDiffNorm = 1;
    double RefDiffNorm = 0;
    int iterations = 99;
    int worksize = (1 + 5000 + 1), linesize = (1 + 5000 + 1);
    int i, j, o, a;
    double diffsum, diff, diffmul;
    init(myTable, myTableOut);
#pragma hmpp <group0_12> _instr_for12_ol_12_main advancedload, args[myTable,
                         myTableOut], args[myTable].addr="myTable",
                         args[myTableOut].addr="myTableOut"
    for (index = 0;
        (index < iterations);
        index++)
    {
#pragma hmpp <group0_12> _instr_for12_ol_12_main callsite,
                          args[myTable, myTableOut].noupdate=true
        _instr_for12_ol_12_main(i, j, myTable, myTableOut);
        theDiffNorm = 0.0;
        diffsum = theDiffNorm;
        a = 0;
#pragma hmpp <group0_12> _instr_for12_ol_17_main callsite,
                         args[myTableOut, myTable].noupdate=true
        _instr_for12_ol_17_main(i, j, diff, myTableOut, myTable, diffmul,
                                &diffsum, a);
        theDiffNorm = diffsum;
    }
#pragma hmpp <group0_12> _instr_for12_ol_17_main delegatedstore,
                         args[myTable], args[myTable].addr="myTable"
    displayRegion(myTable);
#pragma hmpp <group0_12> release
    printf("theDiffNorm:%.12g␣RefDiffNorm=%.12g;", theDiffNorm, RefDiffNorm);
    return 0;
}
```

Table 3.6 | Example of contextual analysis with the *noupdate* directive.

HMPP

```
#pragma hmpp <group0_12> __instr_for12__ol__12__main codelet,
                              args[myTable, myTableOut].io=in
void __instr_for42__ol__12__main(int i, int j, double myTable[5002][5002],
                                 double myTableOut[5002][5002]) {
#pragma hmppcg gridify(i, j)
     for (i = 1; i < (1 + 5000 + 1) - 1; i++) {
          for (j = 1; j < (1 + 5000 + 1) - 1; j++) {
               double neighbor = cos(myTable[i - 1][j]) + sin(myTable[i][j - 1])
                                   + sin(myTable[i][j + 1]) + cos(myTable[i + 1][j]);
               myTableOut[i][j] = neighbor / 3;
} } }
```

Table 3.7 OMP2HMPP transformation. Data transfer between the GPU and CPU is decreased by using the HMPP *noupdate* directive.

OpenMP	HMPP

OpenMP:

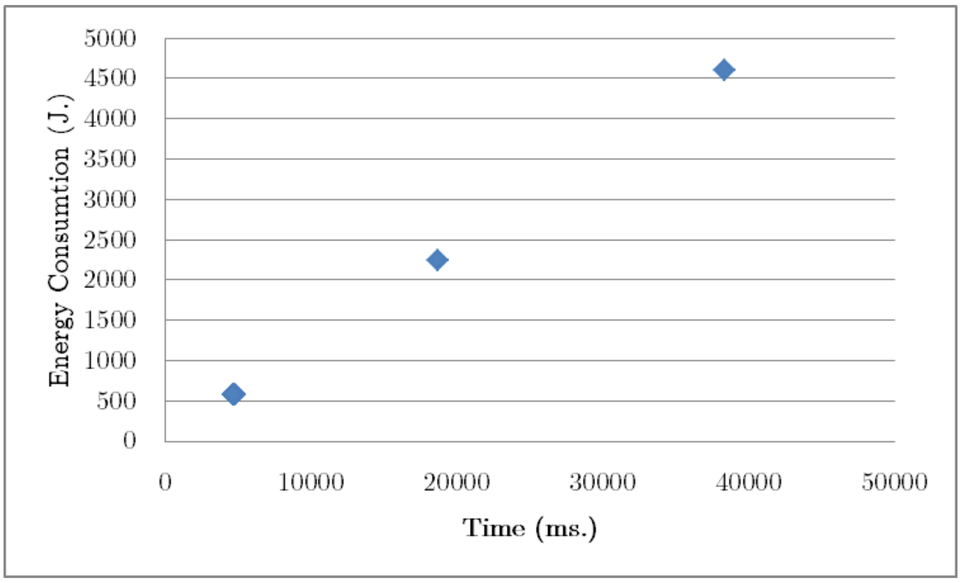
```
void g(int &a,int b) {
     int r=2;
     int c=1;
     a=a+r+2;
     int ret = a+b+c+r*2;
}
int f(int a) {
     return a+1;
}

int main() {
  int l;
  int x=2;
  l=f(1)+f(2)+g(x,6);
  l=l*g(x,2);
  return 1;
}
```

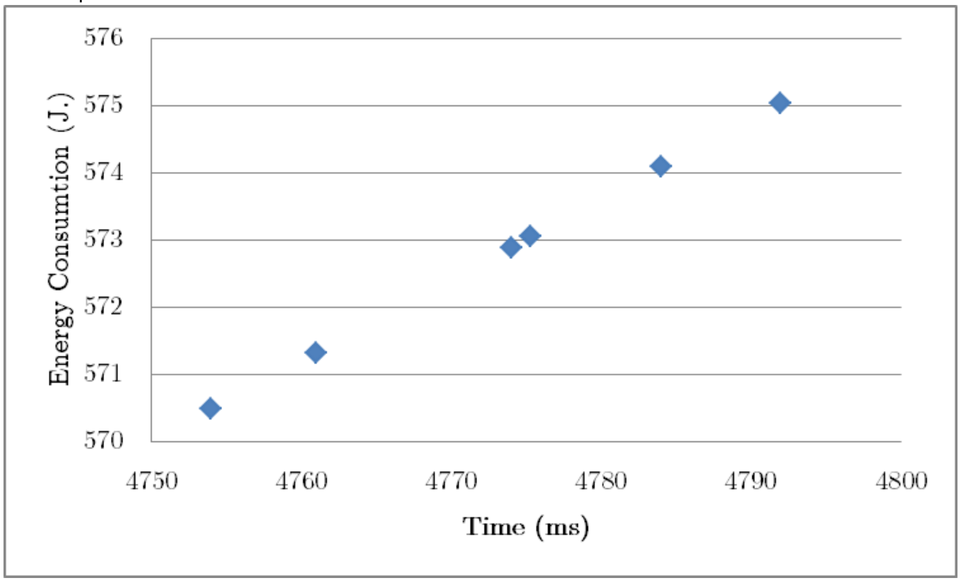
HMPP:

```
int deletedFunctionBodyNamed_g = 1;
int deletedFunctionBodyNamed_f = 1;
int main() {
  int l;
  int x = 2;
  int _p_0_f_0 = 1;
  int _return_0;
  {
    int ret_f0;
    ret_f0 = _p_0_f_0 + 1;
    _return_0 = ret_f0;
  }
  int _p_0_f_1 = 2;
  int _return_1;
  {
    int ret_f1;
    ret_f1 = _p_0_f_1 + 1;
    _return_1 = ret_f1;
  }
  int *_p_0_g_2 = &x;
  int _p_1_g_2 = 6;
  int _return_2;
  {
    int r = 2;
    int c = 1;
    *_p_0_g_2 = *_p_0_g_2 + r + 2;
    int ret = *_p_0_g_2 + _p_1_g_2
              + c + r * 2;
    int ret_g2;
    ret_g2 = ret;
    _return_2 = ret_g2;
  }
  l = _return_0 + _return_1 + _return_2;
  int *_p_0_g_3 = &x;
  int _p_1_g_3 = 2;
  int _return_3;
  {
    int r = 2;
    int c = 1;
    *_p_0_g_3 = *_p_0_g_3 + r + 2;
    int ret = *_p_0_g_3 + _p_1_g_3
              + c + r * 2;
    int ret_g3;
    ret_g3 = ret;
    _return_3 = ret_g3;
  }
  l = l * _return_3;
  return 1;
}
```

Table 3.8 | Inline transformation.

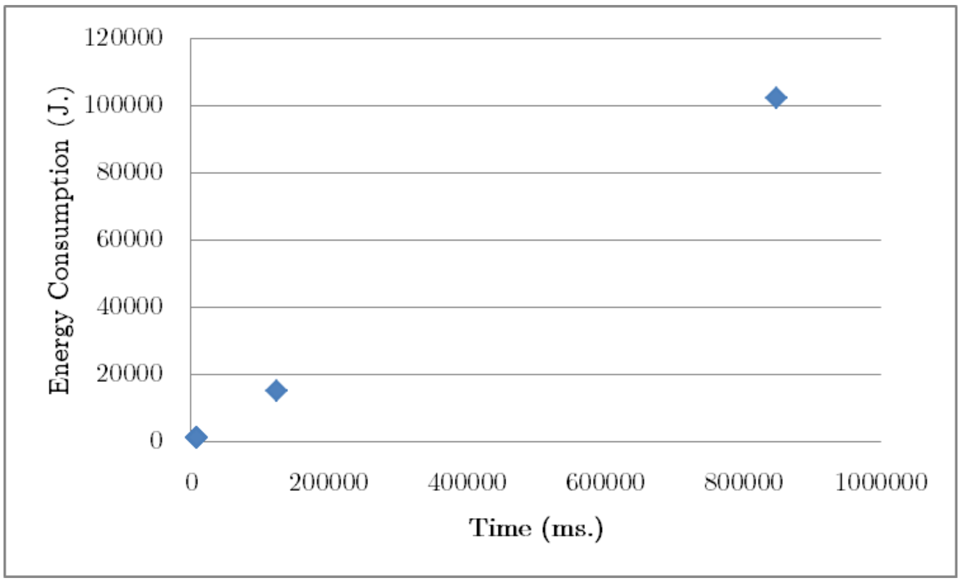


(a) | LU

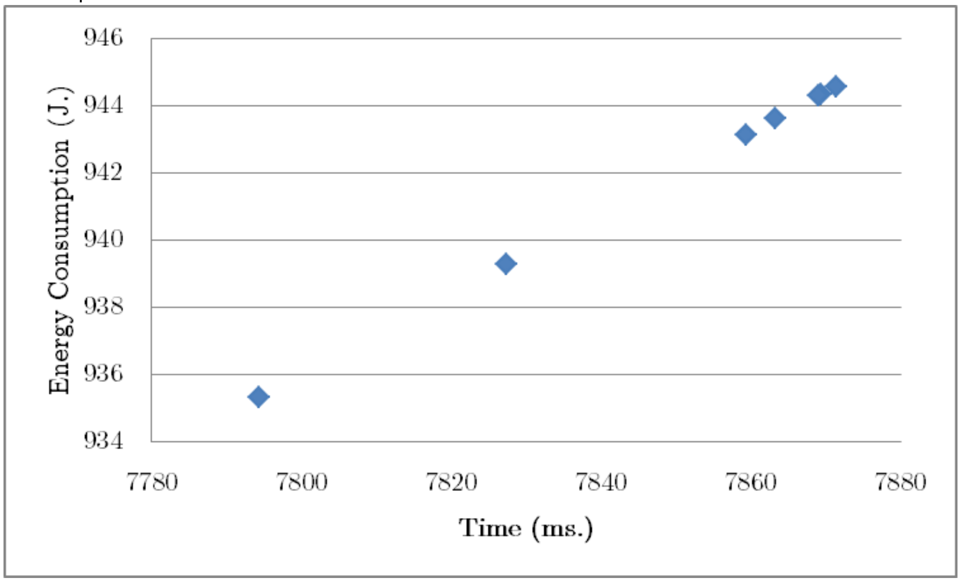


(b) | LU detail

Figure 3.20 | Energy/Time trade-off. Problem subset of the Polybench benchmark. (4/5)



(a) | GEMM



(b) | GEMM detail

Figure 3.21 | Energy/Time trade-off. Problem subset of the Polybench benchmark. (5/5)

Version/Measure	Signature	Time Expended (ms)	Energy Consumption (J)
Original (OpenMP)	0, 0, 0	59500	17428
Adv_loaddelSto...	9, 1, 0	9611	3401,55
Adv_loadRel...	11, 3, 0	10530,2	3819,2
Adv_loadRel...	11, 1, 0	10572,4	4109,9
Adv_loadRel...	10, 1, 0	10844,4	3974,2
...	...	...	...

Table 3.9 Example OMP2HMPP CSV spreadsheet output for Jacobi implementation.

OMP2MPI: Automatic MPI Code Generation from OpenMP Programs | 4

In this chapter, I develop the proposed S2S compiler tool called OMP2MPI. OMP2MPI extends the use of parallel programming on shared memory architectures to distributed memory architectures using MPI, as illustrated in Figure 1.7. The new S2S compiler tool transforms OpenMP source code into MPI source code. The tool automatically generates code that can be executed on different kinds of DM systems, such as large HPC clusters or distributed memory experimental processors such as Intel Polaris, Ambric, or experimental FPGA-based multi-soft-core platforms (e.g., [21]).

Figure 4.4 shows a simplified process flow for the OMP2MPI compiler, where an OpenMP input code is transformed into an MPI using the AST. OMP2MPI detects and transforms OpenMP blocks (focused in "*#pragma omp parallel for*"), dividing selected tasks in the MPI master and slave processes that will be distributed on the available cores. To determine which OpenMP blocks need to be transformed, OMP2MPI uses the directives proposed in [64]. This process is illustrated in the input code example shown in Table 4.1.

The OMP2MPI tool is able to use the combination of peer-to-peer communication functions (MPI_Send, MPI_Recv) and divide the code into sequential and parallel parts using MPI ranks.

With these MPI functions OMP2MPI, in contrast to [56], is able to correctly implement a MPI parallel program that overcomes the problems detected in this thesis and generate code that can be compared to an MPI hand-coded version of the original problem. OMP2MPI transforms the original code by initializing MPI

```
int main(int argc, char** argv) {
  ...
  init_array();
  ...
 #pragma omp parallel for private (iy, ix) target mpi
  for (iz = 0; iz < Cz; iz++) {
    for (iy = 0; iy < Cym; iy++) {
      for (ix = 0; ix < Cxm; ix++) {
        clf[iz][iy] = Ex[iz][iy][ix]
                      - Ex[iz][iy + 1][ix]
                      + Ey[iz][iy][ix + 1]
                      - Ey[iz][iy][ix];
        tmp[iz][iy] = (cymh[iy] / cyph[iy]);
        Hz[iz][iy][ix] = (cxmh[ix] / cxph[ix]);
        Bza[iz][iy][ix] = tmp[iz][iy];
      }
      tmp[iz][iy] = (cymh[iy] / cyph[iy])
                    * Bza[iz][iy][Cxm]
                    - (ch / cyph[iy])
                    * clf[iz][iy];
      Hz[iz][iy][Cxm] = (cxmh[Cxm] / cxph[Cxm]);
      Bza[iz][iy][Cxm] = tmp[iz][iy];
    }
  }
 double total = 0;
  for (int iter = 0; iter < Cz; iter++) {
      for (int iter2 = 0; iter2 < Cym; iter2++) {
          for (int iter3 = 0; iter3 < Cxm; iter3++) {
            total+=Bza[iter][iter2][iter3];
            }
        }
  }
  return 0;
}
```

Table 4.1 | OpenMP block source code example using the created *target* clause.

and distributing workload based on the process rank of the calling process in the communicator. The master process has rank 0 and contains all the sequential code from the original OpenMP application. It also manages the shared memory access, thus keeping all of the slaves updated, as shown in Figure 4.2. For the original OpenMP memory access, represented in Figure 4.1, all the created threads have access to shared memory.

4.1 AST Manipulation

The AST manipulation stage shown in Figure 4.4 is composed of four main steps: 1) task division, 2) context analysis, 3) loop analysis, and 4) workload distribution.

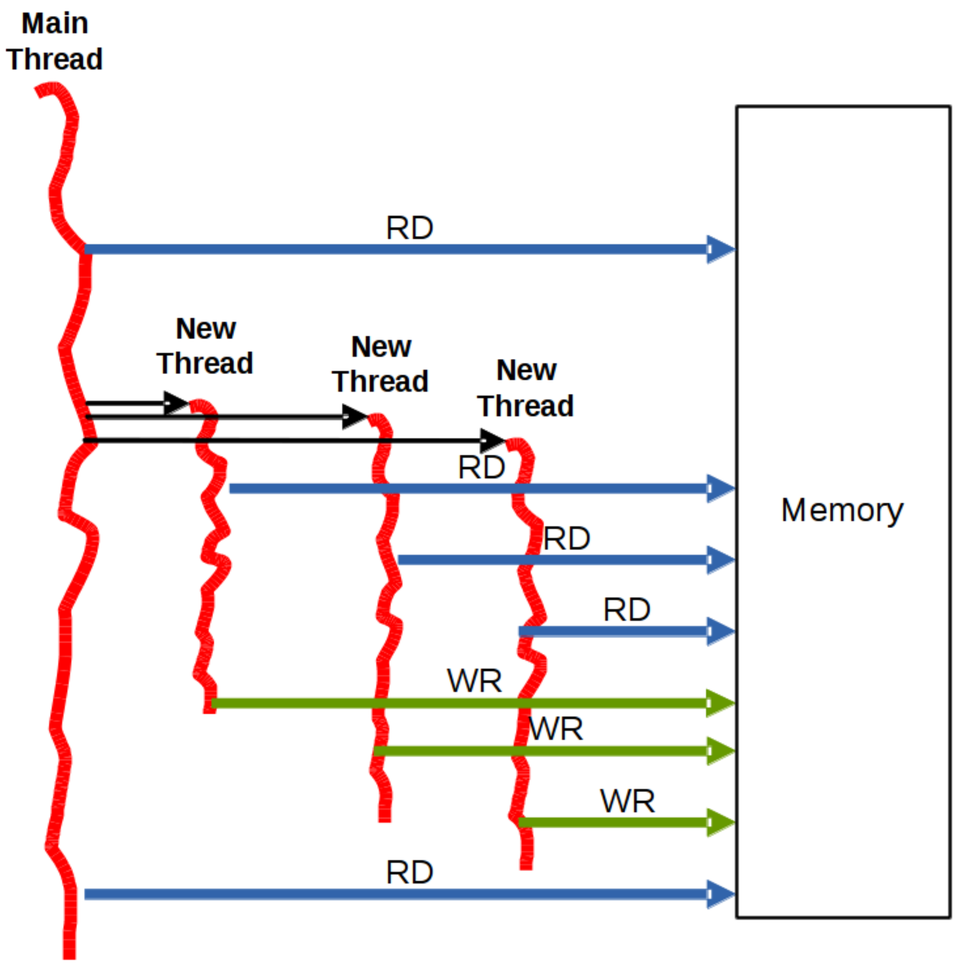


Figure 4.1 Example memory access pattern of an OpenMP application. Threads directly access the shared memory. Blue lines represent read operations and red lines represent write operations.

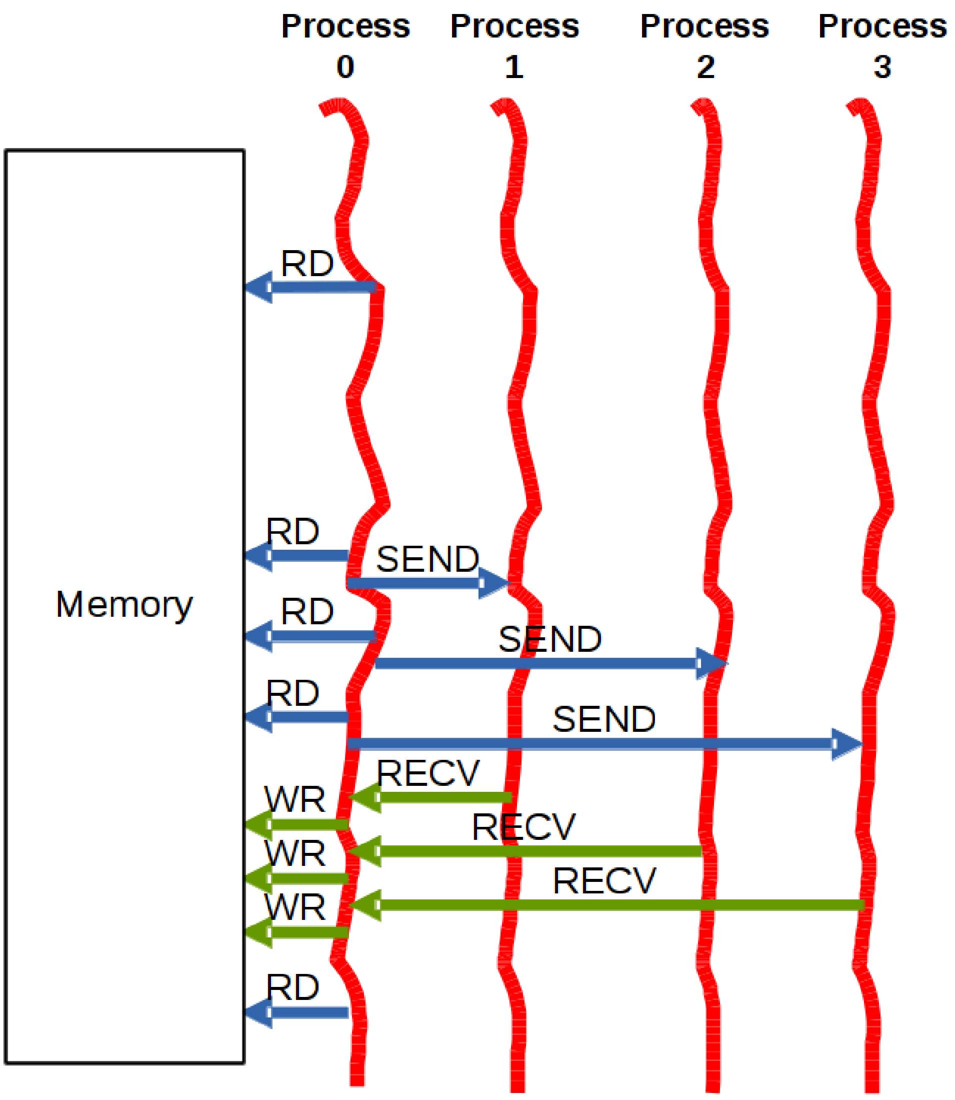
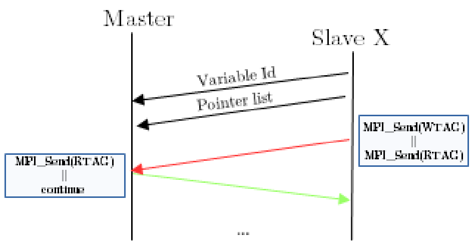
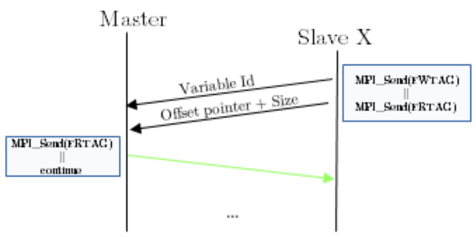


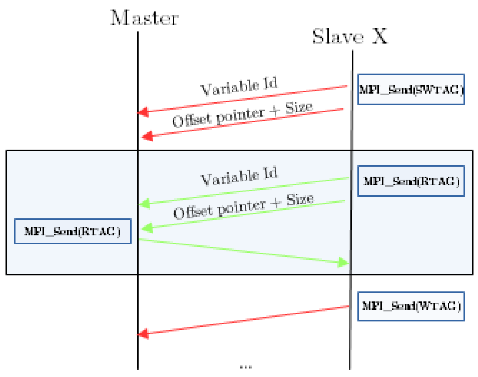
Figure 4.2 Example proposed memory access pattern for shared variables in MPI target applications. Access to shared variables is managed by the master node, and worker processes must communicate with it to access them. Blue lines represent read operations, and red lines represent write operations.



(a) Example of read/write request using the demand method. Each element is accessed individually.


Master
Slave X
Variable Id
Offset pointer + Size
MPI_Send(FWTAG) || MPI_Send(FRTAG)
MPI_Send(FRTAG) || continue
...

(b) Example of a read/write request using the group arrays method. Each element is accessed by a range defined by an offset pointer to a fixed size location.



(c) Example of a secure write request. Every element is accessed individually. The demanding process blocks the master communications to all other slaves until it finishes (atomicity operation).

Figure 4.3 Illustration of the shared memory access method. Red lines represent read operations and green lines represent write operations.

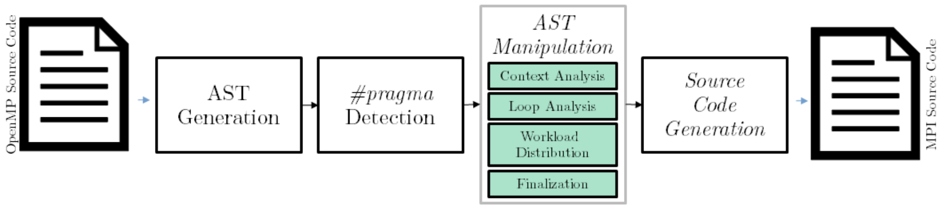


Figure 4.4 AST manipulation process performed by OMP2MPI (grey boxes) surrounded by reused Mercurium framework blocks(grey boxes).

4.1.1 Task Distribution

This stage distributes the original code calculations by implementing the MPI initialization in the communicator and distributing the workload based on the process rank of the called processes. OMP2MPI has two different alternatives for distributing tasks: full task division or finalization.

When an input source code uses full task distribution, all the processes share all the variable declarations and contain the parallelized parts of the code, but the remaining source code is simply performed in the master. This methodology allows slaves to perform other tasks while the master does not require them.

In contrast, in the finalization method (detailed in [56]), all the slaves do the same work (including initialization and previous sequential computation) on the updated values, as explained in Section 4.1.2, until the end of the last parallelized block, decreasing the number of data transfers. After the finalization of all the OpenMP-transformed blocks, it asserts the *MPI_Finalize* instructions and assigns the remaining process to the master node, avoiding unnecessary computation in the slaves.

4.1.2 Context Analysis

To transform the original code, OMP2MPI analyzes the context where the OpenMP block was originally computed and performs an accurate contextual analysis of the AST for each of the variables needed inside it. In MPI, every executed process manages its own private variables independently. The main problem when transform OpenMP to MPI is caused by the shared variables. OMP2MPI studies each shared variable used in an OpenMP block and analyzes the AST to identify when or whether they are accessed. OMP2MPI distinguishes the variables used in an OpenMP block into Input Variables (IN) variables (read inside the block and not modified), Output Variables (OUT) variables (written in the block and whose result is needed after block finalization), and Input Output Variables (INOUT) variables (variables that comply with both cases). In the example shown in Table 4.1, OMP2MPI determines that all variables read in the OpenMP block are IN variables (i.e., *Ex* and *Ey*), but just one of the written variables *Bza* is considered OUT because it is read after the block finalization.

The context analysis stage also includes the study of the OpenMP block context, for instance to detect that the OpenMP block to be transformed is placed inside a loop. In this case, OMP2MPI modifies where the initialization and task synchronization instructions are inserted. An OpenMP block located inside a loop condition modifies how the variable use (IN or OUT) is understood by the tool. In the problem illustrated in Table 4.1, OMP2MPI considers only *Bza* as an OUT because only this variable is read after block finalization. By modifying the problem and context of the OpenMP block by putting it inside a loop, all the variables that are written inside that kernel will be considered OUT if they are previously read. All the variables modified on iteration x of the loop that contains the OpenMP block require the updated value to be read on iteration $x + 1$.

4.1.3 Loop Analysis

This stage is dedicated to studying the loop that is included in the *pragma omp for* directive. Its goal is to correctly divide the inner computation statements of the for loop. OMP2MPI performs an exhaustive analysis of the *for* semantics to determine: 1) the iterated variable, 2) the variable‘s initial value, 3) the variable‘s final value 4) the decrements/increment after every iteration, and 5) the logic comparison operation. However, there are some cases in which OMP2MPI is not able to transform loops depending on specific *for* loop semantics, i.e., complex nonlinear increments of the iterator or multiple cases in the condition. OMP2MPI will generate an alert that informs the user that the selected block cannot be transformed or can only be transformed by automatic or manual loop normalization.

A loop is considered normalized if it is formed in an incremental way, from a lower to higher iterator values and if the condition to follow is a less than operator ("<"). A non-normalized loop can lead to additional errors in the following compilation stages, and OMP2MPI offers the possibility to perform this transformation automatically by modifying the loop formation. Figure 4.5 illustrates an example of the transformation process and how the original loop parts are transformed into their normalized version.

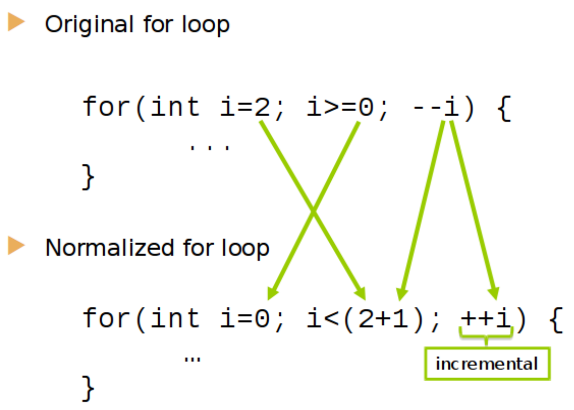
```
int main(...)
{
    int myid; MPI_Status stat; int size; int *argcVar = NULL; char ***argvVar = NULL;
    MPI_Init(argcVar, argvVar);
    MPI_Comm_size(MPI_COMM_WORLD, &size);
    MPI_Comm_rank(MPI_COMM_WORLD, &myid);
    int iz, iy, ix;
    int Cz = 10;
    int Cym = 10;
    int Cxm = 10;
    if (myid == 0) {
        init_array();
    }
    const int FTAG = 0;
    const int ATAG = 1;
    const int RTAG = 2;
    const int WTAG = 3;
    const int SWTAG = 4;
    const int FRTAG = 5;
    const int FWTAG = 6;
    double timeFinish;
    int coordVector0[3];
    int partSize = ((Cz - (0))) / (size - 1) > 0 ? ((Cz - (0))) / (size - 1) / 2 : 1;
    int offset;
    /*Master Source Code*/
    ...
    /*Slave Source Code*/
    if (myid != 0) {
        while (1) {
            MPI_Recv(&offset, 1, MPI_INT, MPI_ANY_SOURCE, MPI_ANY_TAG,
                     MPI_COMM_WORLD, &stat);
            if (stat.MPI_TAG == ATAG) {
                MPI_Recv(&partSize, 1, MPI_INT, 0, ATAG, MPI_COMM_WORLD, &stat);
                int idxForReadWriteSwitch;
                MPI_Recv(&ch, 1, MPI_DOUBLE, 0, ATAG, MPI_COMM_WORLD, &stat);
                MPI_Recv(&Ey[offset], partSize * EySizeDim1 * EySizeDim2, MPI_DOUBLE,
                         0, ATAG, MPI_COMM_WORLD, &stat);
                MPI_Recv(&Ex[offset], partSize * ExSizeDim1 * EySizeDim2, MPI_DOUBLE,
                         0, ATAG, MPI_COMM_WORLD, &stat);
                (coordVector0[0] = 0);
                (coordVector0[1] = ((Cym) - coordVector0[0]));
                (idxForReadWriteSwitch = 1);
                MPI_Send(&idxForReadWriteSwitch, 1, MPI_INT, 0, FRTAG,
                         MPI_COMM_WORLD);
                MPI_Send(&coordVector0, 2, MPI_INT, 0, FRTAG, MPI_COMM_WORLD);
                MPI_Recv(&cymh[coordVector0[0]], coordVector0[1], MPI_DOUBLE, 0,
                         FRTAG, MPI_COMM_WORLD, &stat);
                (coordVector0[0] = 0);
                (coordVector0[1] = ((0) > (Cym)) ? ((Cym) - (0)) : ((0) - (0)));
                if ((0) < (0) && coordVector0[1] > 0) {
                    (idxForReadWriteSwitch = 1);
                    MPI_Send(&idxForReadWriteSwitch, 1, MPI_INT, 0, FRTAG,
                             MPI_COMM_WORLD);
                    MPI_Send(&coordVector0, 2, MPI_INT, 0, FRTAG, MPI_COMM_WORLD);
                    MPI_Recv(&cymh[coordVector0[0]], coordVector0[1], MPI_DOUBLE,
                             0, FRTAG, MPI_COMM_WORLD, &stat);
                }
                (coordVector0[0] = (((Cym)) > (0)) ? ((Cym)) : (0));
                (coordVector0[1] = (((Cym)) > (0)) ? (Cym - ((Cym))) : (Cym - 0));
                if ((Cym) > (Cym) && coordVector0[1] > 0) {
                    (idxForReadWriteSwitch = 1);
                    MPI_Send(&idxForReadWriteSwitch, 1, MPI_INT, 0, FRTAG,
                             MPI_COMM_WORLD);
                    MPI_Send(&coordVector0, 2, MPI_INT, 0, FRTAG, MPI_COMM_WORLD);
                    MPI_Recv(&cymh[coordVector0[0]], coordVector0[1], MPI_DOUBLE, 0,
                             FRTAG, MPI_COMM_WORLD, &stat);
                }
                ...
```

Table 4.2 Automatically generated MPI source code with expanded slave code. OMP2MPI automatically groups accesses into read/write requests with conditional range updates. (1/2)

```
...
for (int iz = offset; iz < offset + partSize; ++iz) {
                    double cymh_iy; double cyph_iy; double cxmh_ix; double cxph_ix;
                    double cxmh_Cxm; double cxph_Cxm;
                    for (iy = 0; iy < Cym; iy++) {
                        for (ix = 0; ix < Cxm; ix++) {
                            clf[iz][iy] = Ex[iz][iy][ix] − Ex[iz][iy + 1][ix]
                                        + Ey[iz][iy][ix + 1] − Ey[iz][iy][ix];
                            (tmp[iz][iy] = (cymh[iy] / cyph[iy]));
                            (Hz[iz][iy][ix] = (cxmh[ix] / cxph[ix]));
                            (Bza[iz][iy][ix] = tmp[iz][iy]);
                        }
                        (tmp[iz][iy] = (cymh[iy] / cyph[iy]) ∗ Bza[iz][iy][Cxm]
                                    − (ch / cyph[iy]) ∗ clf[iz][iy]);
                        if (Cxm > (Cxm) || Cxm < (0)) {
                            (idxForReadWriteSwitch = 2);
                            (coordVector0[0] = Cxm);
                            MPI_Send(&idxForReadWriteSwitch, 1, MPI_INT, 0, RTAG,
                                    MPI_COMM_WORLD);
                            MPI_Send(&coordVector0, 1, MPI_INT, 0, RTAG,
                                    MPI_COMM_WORLD);
                            MPI_Recv(&cxmh_Cxm, 1, MPI_DOUBLE, 0,
                                    RTAG, MPI_COMM_WORLD, &stat);
                        } else {
                            (cxmh_Cxm = cxmh[Cxm]);
                        }
                        if (Cxm > (Cxm) || Cxm < (0)) {
                            (idxForReadWriteSwitch = 6);
                            (coordVector0[0] = Cxm);
                            MPI_Send(&idxForReadWriteSwitch, 1, MPI_INT, 0, RTAG,
                                    MPI_COMM_WORLD);
                            MPI_Send(&coordVector0, 1, MPI_INT, 0, RTAG,
                                    MPI_COMM_WORLD);
                            MPI_Recv(&cxph_Cxm, 1, MPI_DOUBLE, 0, RTAG,
                                    MPI_COMM_WORLD, &stat);
                        } else {
                            (cxph_Cxm = cxph[Cxm]);
                        }
                        (Hz[iz][iy][Cxm] = (cxmh_Cxm / cxph_Cxm));
                        (Bza[iz][iy][Cxm] = tmp[iz][iy]);
                    }


            }
                MPI_Send(&partSize, 1, MPI_INT, 0, ATAG, MPI_COMM_WORLD);
                MPI_Send(&offset, 1, MPI_INT, 0, ATAG, MPI_COMM_WORLD);
                MPI_Send(&Bza[offset], partSize ∗ BzaSizeDim1 ∗ BzaSizeDim2,
                         MPI_DOUBLE, 0, ATAG, MPI_COMM_WORLD);
            } else if (stat.MPI_TAG == FTAG) {
                    break;
    } } }
    MPI_Finalize();
    double total = 0;
    if (myid == 0) {
        for (int iter = 0; iter < Cz; iter++) {
            for (int iter2 = 0; iter2 < Cym; iter2++) {
                for (int iter3 = 0; iter3 < Cxm; iter3++) {
                    total += Bza[iter][iter2][iter3];
                } } }
        printf("total=%f\n", total);
    }
    return 0;
}
```

Table 4.3 Automatically generated MPI source code with expanded slave code. OMP2MPI automatically groups accesses into read/write requests with conditional range updates. (2/2)



```
for(int i=2; i>=0; --i) {
        ...
}
```

```
for(int i=0; i<(2+1); ++i) {
        ...
}
```

Figure 4.5 | Example of the loop normalization process.

4.1.4 Workload Distribution

When the context and the proper loop semantics are understood, OMP2MPI distributes the OpenMP block calculation to work with the master/slave MPI model according to a producer/consumer paradigm. OMP2MPI has already handled all the variables at the context analysis stage. Figures 4.9a and 4.9b show how OMP2MPI divides the computation for each of the OpenMP blocks. The iterations of the OpenMP block are divided differently depending on whether the original OpenMP block contains a static or guided *schedule* clause in its pragma clauses. OMP2MPI divides the iterations of the *for* loop between all the available slaves, as described in Equation 4.1, where S_i is the number of iterations for slave i, $maxV$ it she maximum value of the iterator, $minV$ is the minimum value of the iterator, sN is the number of slaves available, and dF is the division factor that could be determined by the command line in OMP2MPI execution. The value of S_i can also be fixed using the OMP2MPI command line to one size for all slaves (if it is possible). Figure 4.9b shows the distribution model for an OpenMP dynamic block, and Figure 4.9a shows how an OpenMP static or guided block is distributed among the master and slaves.

$$S_i = \frac{(maxV - minV)}{sN}/dF. \tag{4.1}$$

By using static division, the outer loop is scheduled in a round-robin fashion using MPI_Recv from specific ranks. This could lead to an unbalanced load. However, this kind of division is needed because OMP2MPI is thought to be faithful to the

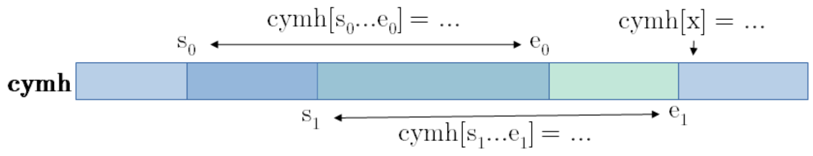


Figure 4.6 | Index accesses knowledge at compilation time.

original OpenMP code, which could have this directive. In contrast, for dynamic division, the outer loop is scheduled dynamically using ANY_SOURCE MPI_Recv, which gives more efficient results.

In previous versions of OMP2MPI [56], the workload distribution used (shown in Figure 4.7a) does not properly deal with all the possible shared variables that could be found in an OpenMP block. To overcome these problems, I issue back to the formal structure shown in Figure 4.3a. Each individual read/write access is managed through the master node by a read/write request from the slave (on demand). This is solved by the producer/consumer paradigm allowing the master to perform these requests in the transformed source code. This is described in Figure 4.3a using Read Tag (RTAG) and Write Tag (WTAG). In a similar way, OMP2MPI includes the possibility of concurrent accesses to these variables using Secure Write Tag (SWTAG), as is illustrated in Figure 4.3c. Table 4.6 shows an example of the transformation based on these principles and the requests that the source generated for a slave (using the input code illustrated in Table 4.1). Nevertheless, because every memory access implies a slave-to-master request, the communication time significantly increases, as shown in Figure 4.7b.

To decrease the communication time, OMP2MPI analyzes the context information extracted for every variable in order to group, when possible, all read or write accesses into a single variable and transfer them using just one request. One of the OMP2MPI methodologies to group these transfers consists of detecting of the divided iterator, linearly in the first dimensional access pointer in a read/write operation of a variable (i.e., *clf* or *Ex* in line 9 of Table4.1) using MPI_Send and MPI_Recv functions transfers after the starting iterator value. Its maximum value in that division is just the portion of the IN/OUT variables that must be read or have been modified. They need to be counted from the offset to the actual maximum iterator, as illustrated in Figures 4.9a and 4.9b. These variables are received on the slave before the start of the iteration division computation, and are returned once the modified variables value is updated. Line 96 of Table 4.3 shows the instruction that sends the modified array back to the master using the process illustrated in Figure 4.3b. In all the other shared variable cases, OMP2MPI operates using an on-demand method (i.e., *cymh* in line 13 of Table4.1). Figure 4.7c illustrates this method. Communications are significantly decreased with respect to the previous case.

To decrease the number of transfers, OMP2MPI also deals with array variables that are accessed inside a loop nested inside a divided loop and that have a different iterator, especially if it is used as first dimension value conforming to a range of accesses (i.e., *cymh*. To do this, it uses Full Read Tag (FRTAG) or Full Write Tag (FWTAG), as shown in lines 37 to 58 in Table 4.2. With such a grouping, the number of transfers is further decreased, as shown in Figure 4.7d, but there are still some unnecessary transfers that could be deleted.

Each read/write operation that operates a variable on a given range, even in the same expression, is fully transferred using the whole range of variable values. A simple representation of accesses to array *cymh* shown in Figure4.6 shows that OMP2MPI is not able to determine if the value of S_0 is less than, greater than, or the same as S_1 (nor any other values). Applying this example to the problem in Table 4.2, it loads all the values every time an access to its range is detected. This generates redundant accesses to the same variable. For this reason, OMP2MPI uses conditional ranges to define the accessed range that will be determined at execution time. This methodology considerably reduces the number of transfers, as shown in Figures 4.7e and 4.7f. Moreover, in this version, individual accesses to a variable with indexes that are not iterators on a range (i.e., *cymh[x]* in Figure 4.6) will not be transformed as an individual request if the result at execution time indicates that the accessed member is already loaded before the conditional range. This last example can be seen in lines 79 to 87 in Table 4.2.

After that, an optimal and automatically generated version that maximizes the transfer groupings (decreasing their number considerably) is then obtained. Furthermore, OMP2MPI offers to the user the option of using two kinds of work distribution (detailed in Subsection 4.1.1), the finalization method illustrated in Figure 4.7f or full distribution method illustrated in Figure 4.7e. Using the finalization method, users decrease the amount of initialization of data on slaves because all the slaves have the updated values after executing all the same code; however, the full division method allows slaves to be dedicated to other tasks when the master does not require their work.

Finally, a special case of shared variables is the reduced variable, as specified by the OpenMP reduction clause. In this case, OMP2MPI supports a few sets of reduction operations. When a reduction operation is detected, OMP2MPI will determine the starting value of the reduced variable depending on the reduction operation. OMP2MPI uses a starting value of 0 for "+" and "-" operations and 1 for "*" and "/". It also accumulates the received results computed at the slaves in the resulting variable during the reduce operation.

```
int main(...) {
  /*Init MPI*/
  ...
  int coordVector0[maxNumOfDimensionsInStudiedVar];
  int partSize = ((iteratorMaxValue - (iteratorInitValue))) / (size - 1)
                  > iteratorInitValue ?
                  ((iteratorMaxValue - (iteratorInitValue))) / (size - 1)
                  / divisionFactor : 1;
  int offset;
  if (myid == 0) {
      ...
      for (int to = 1; to < size; ++to) {
          if ((followIN + partSize) < iteratorMaxValue) {
              MPI_Send(&followIN, 1, MPI_INT, to, ATAG, MPI_COMM_WORLD);
              MPI_Send(&partSize, 1, MPI_INT, to, ATAG, MPI_COMM_WORLD);
              /*Send in variables*/
          } else if ((iteratorMaxValue - followIN) > 0) {
              (partSize = maxIteratorValue - followIN);
              MPI_Send(&followIN, 1, MPI_INT, to, ATAG, MPI_COMM_WORLD);
              MPI_Send(&partSize, 1, MPI_INT, to, ATAG, MPI_COMM_WORLD);
              /*Send in variables*/
          } else {
              MPI_Send(&offset, 1, MPI_INT, to, FTAG, MPI_COMM_WORLD);
              killed++;
          }
          followIN += partSize;
      }
      while (killed != size - 1) {
          MPI_Recv(&partSize, 1, MPI_INT, MPI_ANY_SOURCE, MPI_ANY_TAG,
                   MPI_COMM_WORLD, &stat);
          int source = stat.MPI_SOURCE;
          if (stat.MPI_TAG == RTAG) {
              switch (partSize) {
                case 0 :
                  MPI_Recv(&coordVector0, 1, MPI_INT, source, RTAG,
                           MPI_COMM_WORLD, &stat);
                  MPI_Send(&var1[coordVector0[0]], 1, MPI_DOUBLE, source, RTAG,
                           MPI_COMM_WORLD);
                  break;
                case 1 :
                    ...
              }
          } else if (stat.MPI_TAG == WTAG) {
              switch (partSize) {
                case 0 :
                  MPI_Recv(&coordVector0, 3, MPI_INT, source, WTAG,
                           MPI_COMM_WORLD, &stat);
                  MPI_Recv(&var2[coordVector0[0]][coordVector0[1]][coordVector0[2]],
                           1, MPI_DOUBLE, source, WTAG, MPI_COMM_WORLD, &stat);
                  break;
              }
          } else if (stat.MPI_TAG == SWTAG) {
              switch (partSize) {
                case 0 :
                  do {
                      MPI_Recv(&partSize, 1, MPI_INT, source, MPI_ANY_TAG,
                               MPI_COMM_WORLD, &stat);
                      if (stat.MPI_TAG == RTAG) {
                          switch (partSize) {
                              case 0 :
                                  MPI_Recv(&coordVector0, 1, MPI_INT, source, RTAG,
                                           MPI_COMM_WORLD, &stat);
                                  MPI_Send(&var1[coordVector0[0]], 1, MPI_DOUBLE,
                                          source, RTAG, MPI_COMM_WORLD);
                                  break;
                              case 1 :
                                  ...
                          }
                      }
                  } while (stat.MPI_TAG != WTAG);
                  MPI_Recv(&coordVector0, 3, MPI_INT, source, WTAG,
                           MPI_COMM_WORLD, &stat);
                  MPI_Recv(&var2[coordVector0[0]][coordVector0[1]][coordVector0[2]],
                          1, MPI_DOUBLE, source, WTAG, MPI_COMM_WORLD, &stat);
                  break;
                case 1:
                  ...
              }
          ...//Continue on (2/2)
```

Table 4.4 | Automatically generated MPI source code with expanded master code. (1/2)

```
...
} else if (stat.MPI_TAG == FRTAG) {
                switch (partSize) {
                  case 0 :
                    MPI_Recv(&coordVector0, 2, MPI_INT, source, FRTAG,
                             MPI_COMM_WORLD, &stat);
                    MPI_Send(&var1[coordVector0[0]], coordVector0[1], MPI_DOUBLE,
                             source, FRTAG, MPI_COMM_WORLD);
                    break;
                  case 1 :
                    ...
                }
            } else if (stat.MPI_TAG == FWTAG) {
                switch (partSize) {
                  case 0 :
                    MPI_Recv(&coordVector0, 2, MPI_INT, source, FWTAG,
                             MPI_COMM_WORLD, &stat);
                    MPI_Recv(&var2[coordVector0[0]],
                             coordVector0[1] * (sizeCoord1) * (sizeCoord2),
                             MPI_DOUBLE, source, FWTAG, MPI_COMM_WORLD, &stat);
                    break;
                }
            } else if (stat.MPI_TAG == ATAG) {
                MPI_Recv(&offset, 1, MPI_INT, source, ATAG, MPI_COMM_WORLD, &stat);
                /*Recieve out vars*/
                MPI_Recv(&reductionVar, 1, MPI_DOUBLE, source, MPI_ANY_TAG,
                         MPI_COMM_WORLD, &stat);
                reducedVar += reductionVar;
                /*Follow next iterations*/
} } } ... }
```

Table 4.5 | Automatically generated MPI source code with expanded master code. (2/2)

These kinds of requests and the reduction performed by the master are shown in Tables 4.4 and 4.5.

4.2 Results

A subset of the Polybench benchmark was compiled using OMP2MPI. The generated versions were executed on 64 CPUs E7-4800 with 2.40 GHz (Bullion quadri module) and compiled with bullxmpi (MPI 2.1 compatible). Bullxmpi has been enhanced by Bull with many new features such as effective abnormal pattern detection, network-aware collective operations, and multi-path network fail-over to increase reliability, resilience, and boost the performance of parallel MPI applications. The codes resulting from the execution of OMP2MPI was compared with the original OpenMP results as well as with a sequential version of the same problem. Figures 4.11, 4.12, 4.13, 4.14, 4.15, and 4.16 compare the speedup for the selected problems. These figures show that OMP2MPI produces a good transformation of the original OpenMP code and, in most cases, the generated code demonstrates better scalability than the original, with a linear speedup increase that is related to the number of processors used for execution. It is remarkable that the results

```
...
for (int iz = offset; iz < offset + partSize; ++iz)
{
     double clf__iz__iy;
     double Ex__iz__iy__ix;
     double Ex__iz__iyPlus1__ix;
     double Ey__iz__iy__ixPlus1;
     double Ey__iz__iy__ix;
     ...
     for (iy = 0; iy < Cym; iy++) {
          for (ix = 0; ix < Cxm; ix++) {
               (idxForReadWriteSwitch = 8);
               (coordVector0[0] = iz);
               (coordVector0[1] = iy);
               (coordVector0[2] = ix);
               MPI_Send(&idxForReadWriteSwitch, 1, MPI_INT,
                         0, RTAG, MPI_COMM_WORLD);
               MPI_Send(&coordVector0, 3, MPI_INT, 0, RTAG,
                         MPI_COMM_WORLD);
               MPI_Recv(&Ex__iz__iy__ix, 1, MPI_DOUBLE, 0,
                         RTAG, MPI_COMM_WORLD, &stat);
               (idxForReadWriteSwitch = 8);
               (coordVector0[0] = iz);
               (coordVector0[1] = iy + 1);
               (coordVector0[2] = ix);
               MPI_Send(&idxForReadWriteSwitch, 1, MPI_INT,
                         0, RTAG, MPI_COMM_WORLD);
               MPI_Send(&coordVector0, 3, MPI_INT, 0, RTAG,
                         MPI_COMM_WORLD);
               MPI_Recv(&Ex__iz__iyPlus1__ix, 1, MPI_DOUBLE,
                         0, RTAG, MPI_COMM_WORLD, &stat);
               /*Other reads on expression*/
               ...
               (clf__iz__iy = Ex__iz__iy__ix - Ex__iz__iyPlus1__ix
                         + Ey__iz__iy__ixPlus1 - Ey__iz__iy__ix);
               (idxForReadWriteSwitch = 1);
               MPI_Send(&idxForReadWriteSwitch, 1, MPI_INT,
                         0, WTAG, MPI_COMM_WORLD);
               (coordVector0[0] = iz);
               (coordVector0[1] = iy);
               MPI_Send(&coordVector0, 2, MPI_INT, 0, WTAG,
                         MPI_COMM_WORLD);
               MPI_Send(&clf__iz__iy, 1, MPI_DOUBLE, 0, WTAG,
                         MPI_COMM_WORLD);
               ...
} ... } } ...
```

Table 4.6 Automatically generated MPI source code with expanded slave code. Each access to a variable implies a request to the master process.

are obtained in a shared memory architecture node, which opens up the possibility of executing both using the same number of processors; nonetheless, the most important aspect of these results is that they open the possibility of using MPI in our original OpenMP-implemented source code, allowing us to execute the transformed codes in many more processors because of the capability of MPI to go outside the node.

Figures 4.11, 4.12, 4.13, 4.14, 4.15, and 4.16 show the OpenMP speedup (red lines) and MPI speedup (yellow lines). Green triangles and yellow rhombuses represent the execution results of the same input problem transformed by OMP2MPI for different numbers of iterations assigned to each slave execution. Green rhombuses indicate that the default size (Equation 4.1) was divided by 2, and yellow triangles indicate that the default size was divided by 4. Figures 4.11, 4.12, 4.13, 4.14, 4.15, and 4.16 show that the modification of the number of iterations assigned to each slave execution can modify the global performance because it affects the load balance. Figure 4.10 compares the speedup of the doitgen and bigc problems, which shows the importance of tuning the size value.

4.3 Concluding Remarks

OMP2MPI is a tool proposed to facilitate the portability of OpenMP source code to MPI. I showed how it effectively and automatically translates OpenMP source code to MPI because it is able to go outside the node, thus allowing the program to exploit non-shared-memory architectures such as clusters or NoC-based MPSoC.

This automatic task is very useful because the programmer can keep working with the OpenMP model, which is easily readable, and simply compile it over the OMP2MPI compiler to obtain the advantages of the MPI model offer (e.g., speedup and scalability). The readability of the generated code is acceptable, so further optimizations can be done by experts to improve performance results. The experimental results obtained on the Polyhedral benchmark in Figures 4.11, 4.12, 4.13, 4.14, 4.15, and 4.16 are promising, especially when considering it is an effortless version. They also produce better scalability than the original OpenMP code. The speedup figures for 64 cores are higher than 60× in some cases, and also higher than the original OpenMP code. These results show again that, as mentioned in the introduction, OpenMP does not always perform better than MPI in shared memory systems.

OMP2MPI considerably reduces the number of transfers. Nevertheless, as illustrated in Figures 4.11c, 4.12a, 4.12c, 4.14b, and 4.15a there are still some cases in which a high number of transfers are produced, especially when the on-demand method is mostly used on the generated transformation. Future versions of OMP2MPI will have to find a way to decrease the number of such transfers, e.g., by creating local variables on slaves that include all the individual accesses.

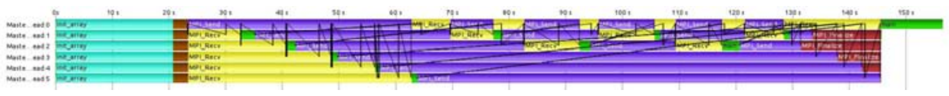
OMP2MPI produces highly tunable results, thus allowing users easily modify the generated versions by changing the sizes of the divided blocks. This decreases the possibility of unbalanced loads, as demonstrated in Figures 4.10a and 4.10b, where greater speedups were obtained by dividing the size of blocks differently.

The current version of the tool has some limitations with respect to standard vectors and dynamic arrays. These could not be easily parsed to determine their size at compilation time, causing some errors. OMP2MPI states that it is not able to transform correctly during compilation. Nevertheless, the access to OMP2MPI source code could still help expert users solve such errors or to improve aspects that could be found by future analysis. After the defence of this thesis I would like to open source the completeness of the source code in[65].

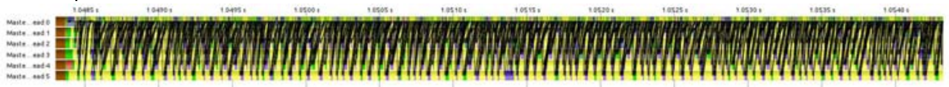
Future improvements will improve transference by detecting elements that are used in a detected range, as explained in 4.1.4, by transferring only the operational indexes in a range, i.e., in the range

```
for(int i=0;i<N;i=i+2) {
A[i][...
}
```

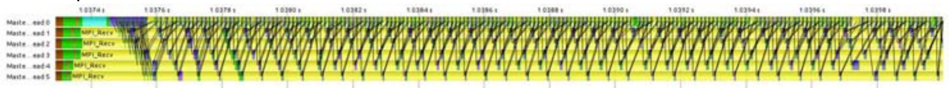
and an inner variable A that is accessed in the first dimension by variable i. OMP2MPI will only transfer the accessed values according to the step of the defined range. This work can be done automatically by analyzing the data access pattern to understand which parts of the variable are read or written after a loop with memory traces inserted is executed.



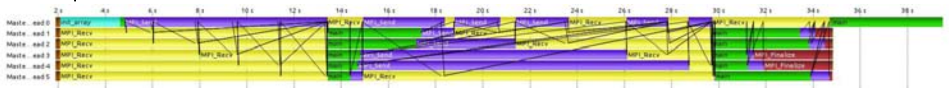
(a) Method implemented in [56]



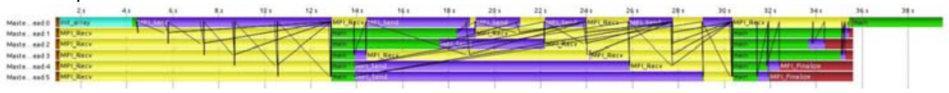
(b) Each read/write operation is transformed into a read-/write request to the master.



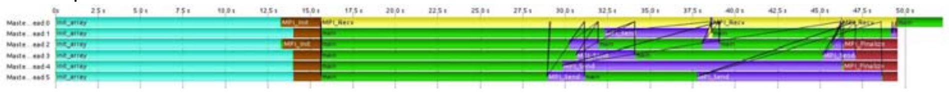
(c) On Start/End arrays or matrices that use the principal iterator as the first dimension access index inside a block are transferred in a group according to the operation range. The rest are treated as illustrated in the previous figure.



(d) All the arrays or matrices that use an iterator as the index of the first dimension, defining a range of required values, are transferred as a group.

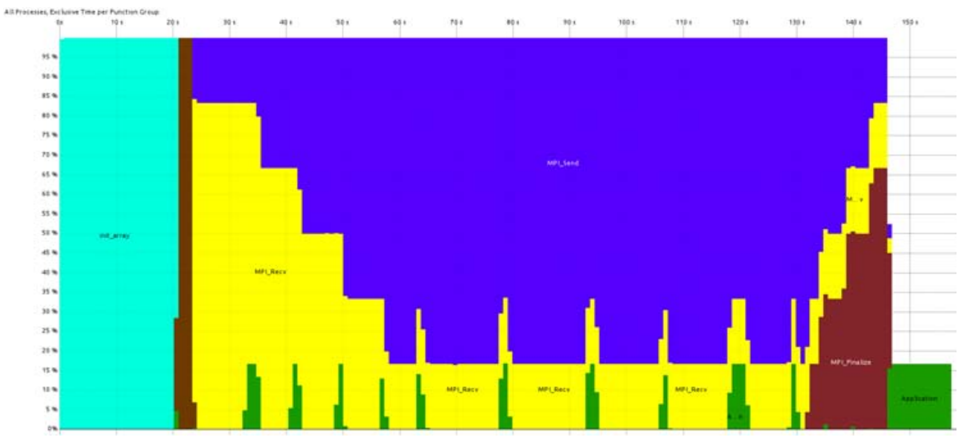


(e) All the arrays or matrices that use, as the first dimension index, an iterator that defines a range are transferred as a group. This case uses conditional ranges during execution, as explained in Section 4.1.4. The task is divided using the finalization method in Section 4.1.1.

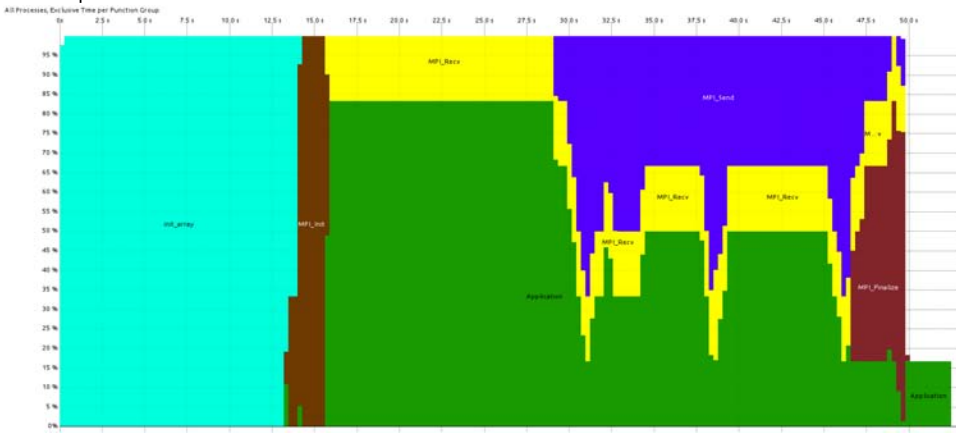


(f) All the arrays or matrices that use, as the first dimension index, an iterator that define a range are transferred as a group. This case uses conditional ranges at execution, as explained in Section 4.1.4. The task is divided using the full division method in Section 4.1.1.

Figure 4.7 Profile analysis for automatically generated MPI versions of the problem shown in Table 4.1 with 10 rows and columns in Figures 4.7b and 4.7c and 300 in the rest. Yellow, dark blue, red, and blue text indicate MPI_Recv functions, MPI_Send functions, MPI_Init and MPI_Finalize functions, and variable initialization, respectively. Some figures show a portion of the whole process for clarity.

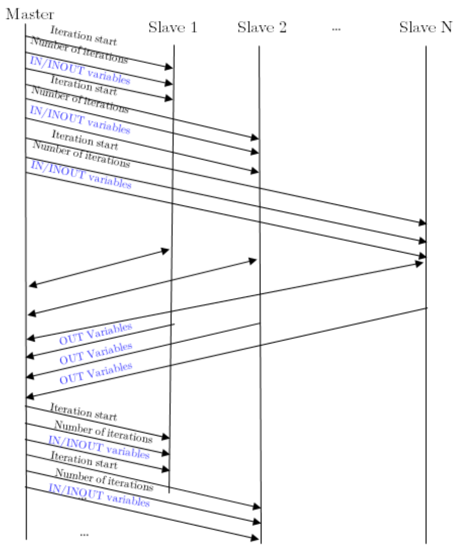


(a) | Execution profile of the method implemented in [56].

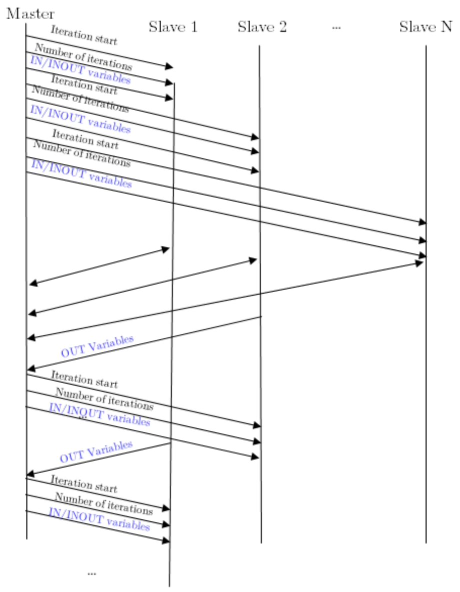


(b) | Execution profile using the methodology in which all the arrays or matrices that use, as the first dimension index, an iterator that define a range are transferred as a group. This case uses conditional ranges at execution, as explained in Section 4.1.4. The task is divided using the full division method in Section 4.1.1.

Figure 4.8 | Profile analysis for automatically generated MPI versions of the problem shown in Table 4.1 with 300 rows and columns. Yellow, dark blue, red, and blue text indicates MPI_Recv functions, MPI_Send Functions, MPI_Init and MPI_Finalize functions, and init_array, respectively.

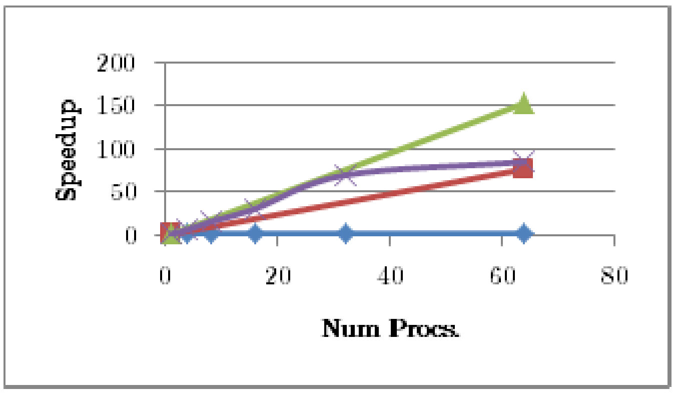


(a) Static workload distribution. The work is sent in an orderly manner depending on the rank of the slaves. All slaves must finish before continuing with the next piece of the workload. Optional communications depending on the selected process are indicated in blue and data exchanged during slave computation is indicated in green.

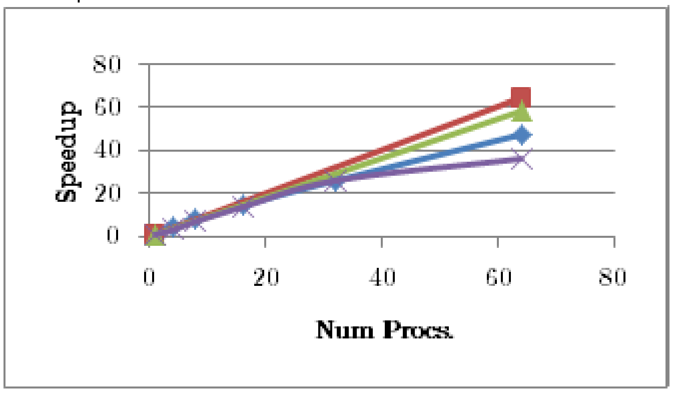


(b) Dynamic workload distribution. The work is divided by dynamically responding to the slave that answers with the range of iterations and variables needed to perform the computation. Optional communications depending on the selected process are indicated in blue and data exchanged during slave computation is indicated in green.

Figure 4.9 | Workload distribution

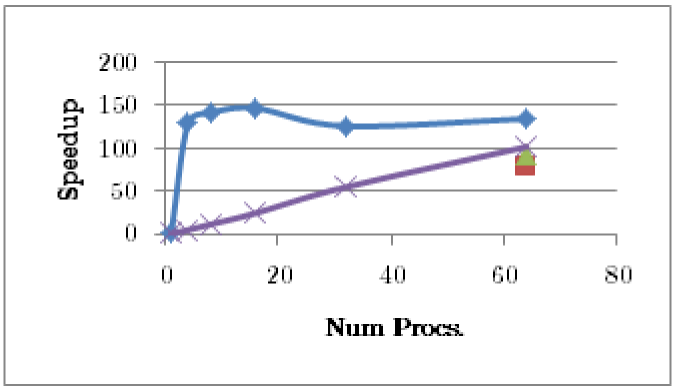


(a) | Bigc

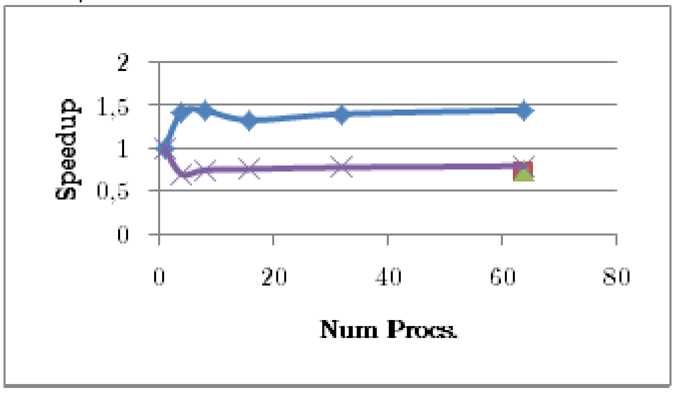


(b) | Doitgen

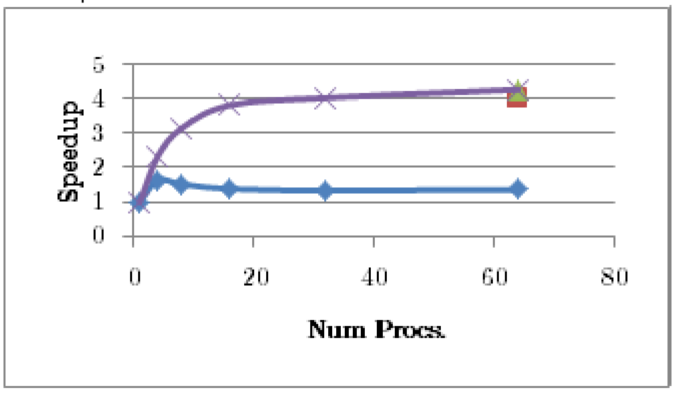
Figure 4.10 | Speedup obtained for the sequential test problems using 16, 32, and 64 processors. Purple lines represent OpenMP speedup, and blue lines show MPI speedup. The green and red lines represents the same problem when dividing the slave block size by 2 and 4, respectively.



(a) | 2mm

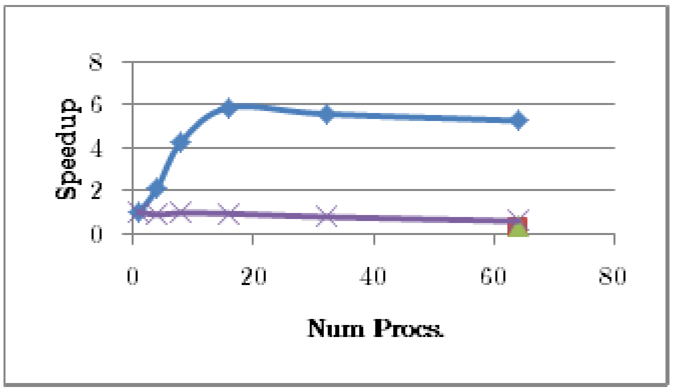


(b) | Adi

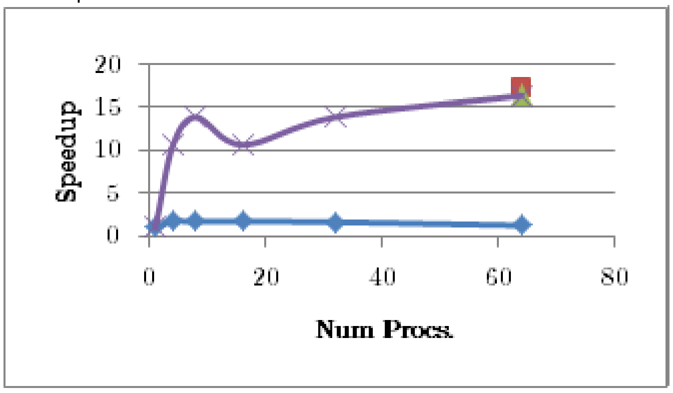


(c) | Convolution

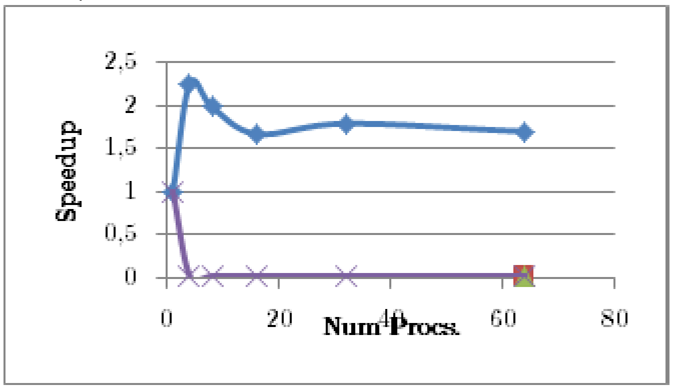
Figure 4.11 | Speedup obtained for the sequential test problems using 16, 32, and 64 processors. Purple lines represent OpenMP speedup, and blue lines show MPI speedup. Green and red lines show the same problem executed using the maximum number of available processors but different slave block sizes. (1/6)



(a) | Fdtd-2d

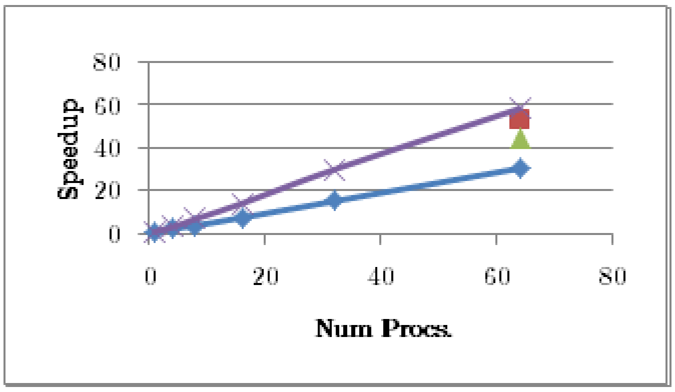


(b) | Fdtd-apml

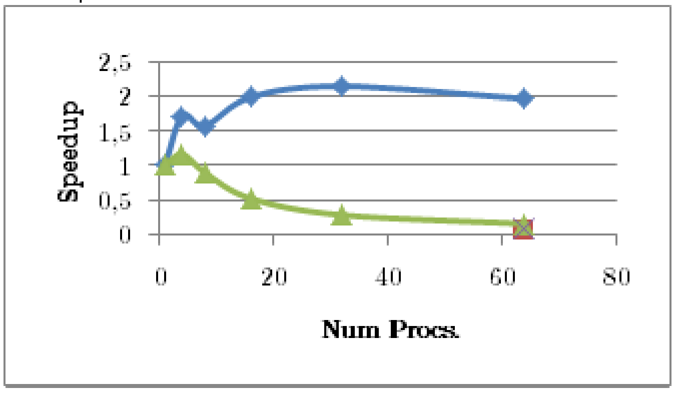


(c) | Gauss filter

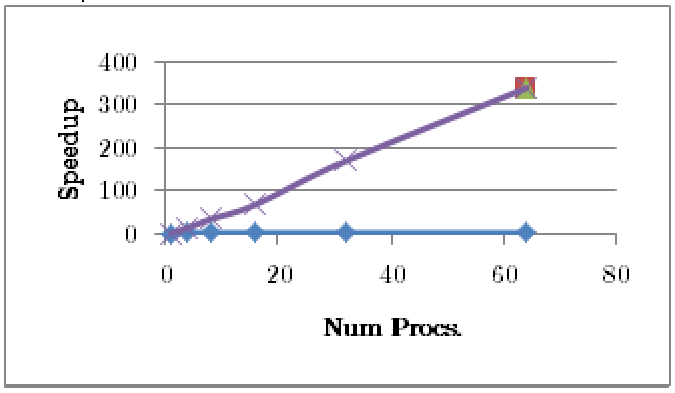
Figure 4.12 | Speedup obtained for the sequential test problems using 16, 32, and 64 processors. Purple lines represent OpenMP speedup, and blue lines show MPI speedup. Green and red lines show the same problem executed using the maximum number of available processors but different slave block sizes. (2/6)



(a) | GEMM

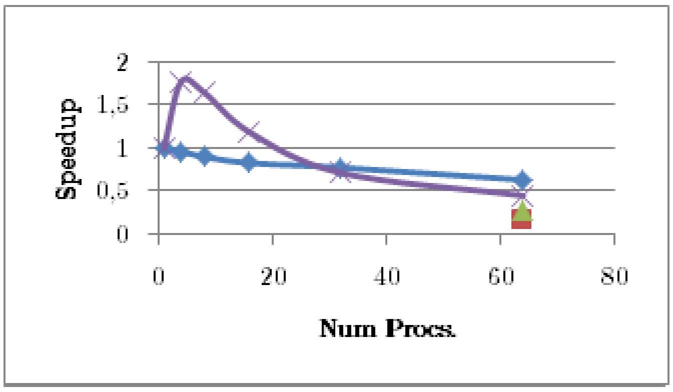


(b) | Gemver

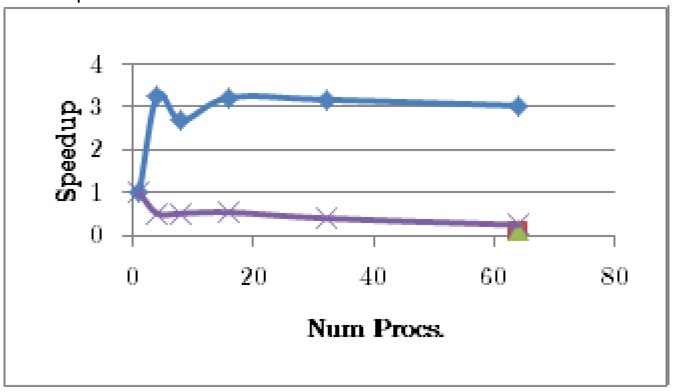


(c) | Gesummv

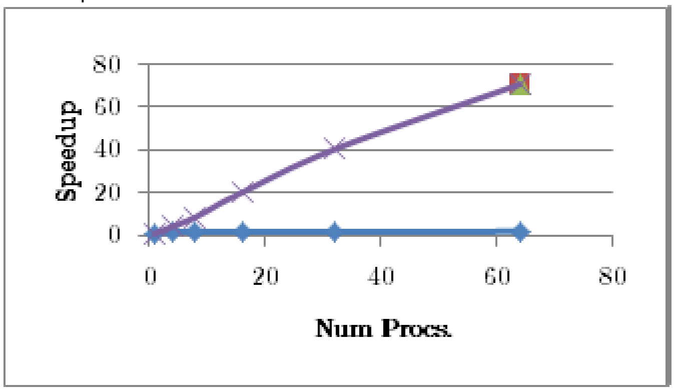
Figure 4.13 | Speedup obtained for the sequential test problems using 16, 32, and 64 processors. Purple lines represent OpenMP speedup, and blue lines show MPI speedup. Green and red lines show the same problem executed using the maximum number of available processors but different slave block sizes. (3/6)



(a) | Jacobi 1d

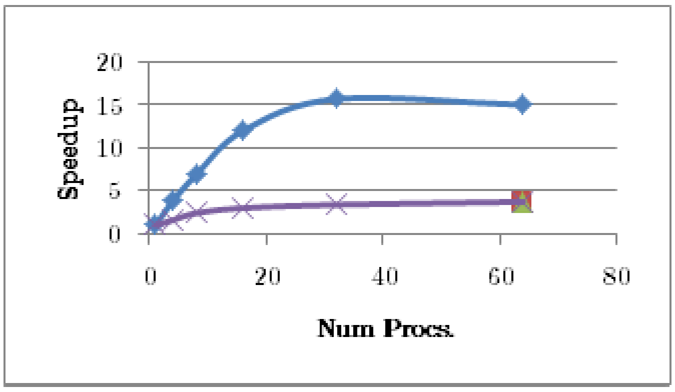


(b) | Jacobi 2d

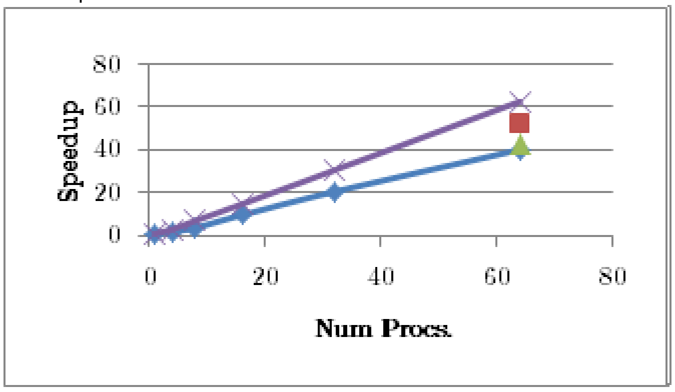


(c) | Mvt

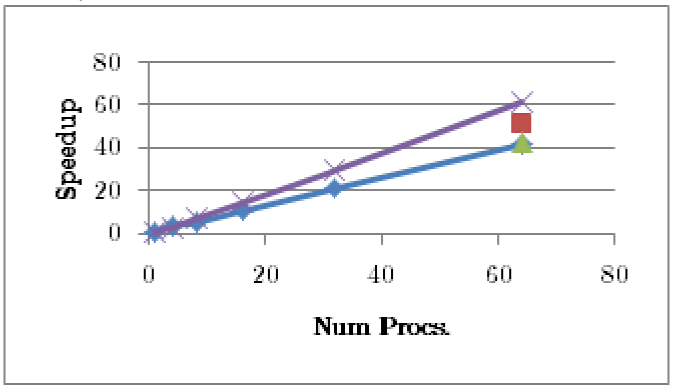
Figure 4.14 | Speedup obtained for the sequential test problems using 16, 32, and 64 processors. Purple lines represent OpenMP speedup, and blue lines show MPI speedup. Green and red lines show the same problem executed using the maximum number of available processors but different slave block sizes. (4/6)



(a) | Seidel

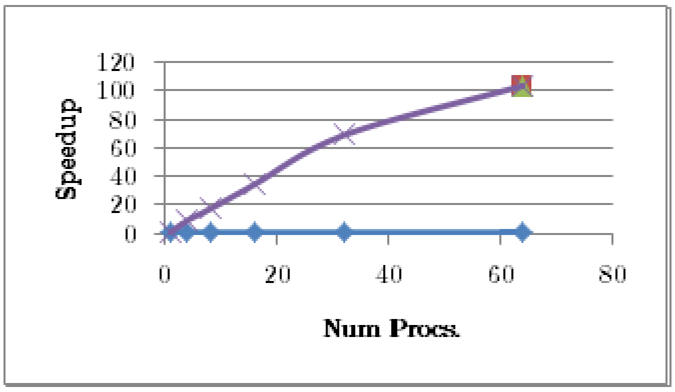


(b) | Syr2k



(c) | Syrk

Figure 4.15 | Speedup obtained for the sequential test problems using 16, 32, and 64 processors. Purple lines represent OpenMP speedup, and blue lines show MPI speedup. Green and red lines show the same problem executed using the maximum number of available processors but different slave block sizes. (5/6)



(a) | Trisolv

Figure 4.16 | Speedup obtained for the sequential test problems using 16, 32, and 64 processors. Purple lines represent OpenMP speedup, and blue lines show MPI speedup. Green and red lines show the same problem executed using the maximum number of available processors but different slaves block sizes . (6/6)

Integrated Modular System 5

To explore the possibilities of the full system work-flow, I first test the use of combined computing using a combination of MPI and OpenMP blocks and generating all the possible versions that can be extracted from the problem, as illustrated in Table 2.4. These versions were parallelized using the Access Pattern tool presented in Chapter 2 and the work-flow explained in Figure 4.3b using OMP2MPI. Figure 4.3b also includes the results of the OpenMP+MPI hybrid model. The hybrid model was constructed by adding a "*#pragma omp parallel for*" inside the slave part of work. This kind of hybrid parallelization could be beneficial when utilizing the high optimization of the shared memory model on each node. As a result, the generated codes are able to take advantage of the MPI capabilities by exploiting the cluster jumping between nodes. Further, performance could be incremented by access to shared memory without memory transfers between nodes. An example is shown in Table 5.1.

In this particular problem, the proposed system could not obtain great scalability using MPI (either by combining OpenMP+MPI or using the hybrid model) because of the time expended on data transference and synchronization, as illustrated in Figure 5.1 on a 10 *times* 10 problem size. Nevertheless, it is useful to understand how difficult it would be for a non-expert user to arrive to that conclusion in contrast to the ease of use of the Access Pattern Visualization and OMP2MPI tools.

Taking as an example the problem illustrated in Figure 4.11a and comparing these results with the results of a hybrid model, it is clear that the proposed tools allows many solutions that can improve the performance of sequential source code to be explored. Figure 5.3 compares the results obtained by the hybrid model for the 2MM problem with those of previous experiments. These results are obtained by

```
if (myid != 0) {
while (1) {
    MPI_Recv(&offset, 1, MPI_INT, MPI_ANY_SOURCE,
                MPI_ANY_TAG, MPI_COMM_WORLD, &stat);
    if (stat.MPI_TAG == ATAG) {
        MPI_Recv(&partSize, 1, MPI_INT, 0, MPI_ANY_TAG,
                MPI_COMM_WORLD, &stat);
        MPI_Recv(&sum[offset], partSize, MPI_DOUBLE,
                0, MPI_ANY_TAG, MPI_COMM_WORLD, &stat);
                #pragma omp parallel for
        for (int i = offset; i < offset + partSize;++i) {
                double x = (i + 0.5) * step;
                sum[i] = 4.0 / (1.0 + x * x);
        }
        MPI_Send(&offset, 1, MPI_INT,
                        0, 0, MPI_COMM_WORLD);
        MPI_Send(&partSize, 1, MPI_INT,
                        0, 0, MPI_COMM_WORLD);
        MPI_Send(&sum[offset], partSize, MPI_DOUBLE,
                        0, 0, MPI_COMM_WORLD);
    } else if (stat.MPI_TAG == FTAG) {
            break;
    }
}
}
```

Table 5.1 | Slave source code. Hybrid MPI + OpenMP

adding a created new OpenMP directive *hybrid* (still in the testing phase) that could be added near the *check* clause, as explained in Chapter 4.

It is not now possible to test the combination of OpenMP+MPI+HMPP, as it was at the start of this thesis three years ago. On June 27, 2014, CAPS Enterprise, who managed the HMPP compiler and its license distribution closed due to economic difficulties [66]. This company maintained the licenses on the NOVA cluster until that date. After that, it was impossible to compile the generated code even when correctly generated, as shown in Tables 5.2 and 5.3. These tables illustrate a possible transformation generated using the TRMM problem from the Polybench benchmark, previously parallelized as explained in Chapter 2 and shown in Table 2.4. In this source, three loops are defined that could be parallelized by the use of OpenMP parallel for blocks. Using the tools presented on this thesis, I demonstrate a correct transformation by combining the three programming languages. The first block is kept as OpenMP, the second block is transformed into MPI using the OMP2MPI tool (the following Figure 5.2 indicates that 16 processors could work well), and, finally, the third block is generated using OMP2HMPP.

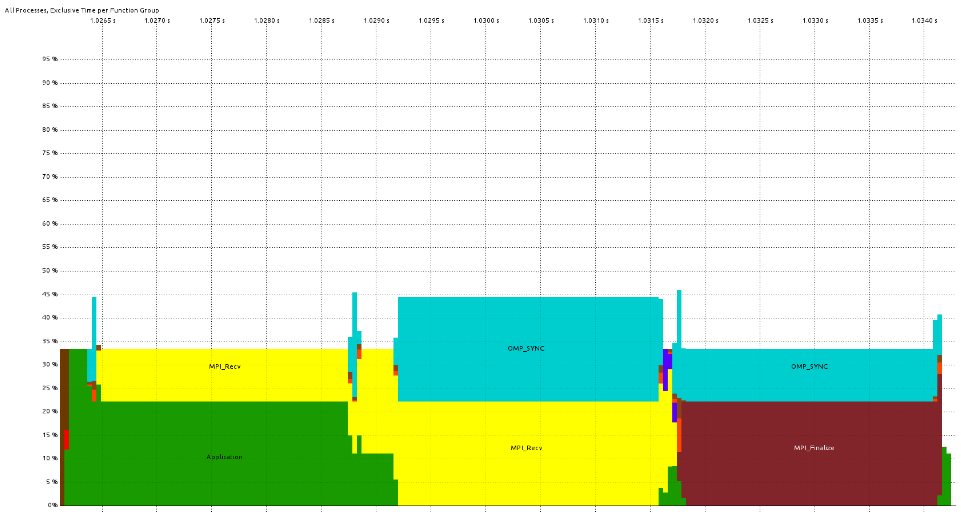


Figure 5.1 Speedup obtained for the sequential test problems using 16, 32, and 64 processors. The OpenMP blocks of the TRMM problem are solved using an OpenMP and MPI combination or hybrid model.

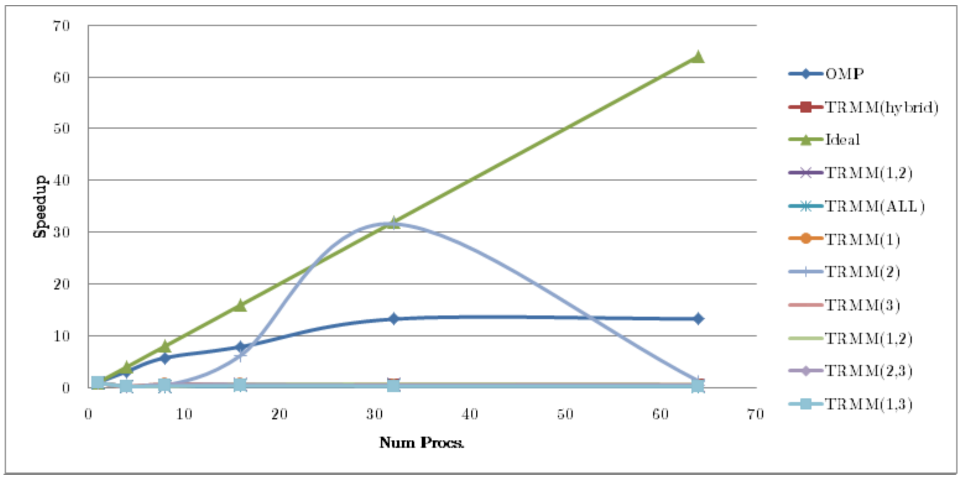


Figure 5.2 Speedup obtained for the sequential test problems using 16, 32, and 64 processors. The OpenMP blocks of the TRMM problem are solved using an OpenMP and MPI combination or hybrid model.

```
int main(...) {
    ...
    for (i = 1; i < n; i++) {
#pragma omp parallel for private(k)
      for (j = 0; j < i; j++) {
          for (k = 0; k < i; k++) {
              B[i][j] += alpha * A[i][k] * B[j][k];
      } }
      (partSize = (((i + 1 - (i))) / (size - 1)) > 0 ? ((((i + 1 - (i)))
                  / (size - 1)) / 1) : 1);
      if (myid == 0) {
          ...//Master Source Code as Explained on Chapter 4
      }
      if (myid != 0) {
          while (1) {
            MPI_Recv(&offset, 1, MPI_INT, MPI_ANY_SOURCE, MPI_ANY_TAG,
                      MPI_COMM_WORLD, &stat);
            if (stat.MPI_TAG == ATAG) {
                MPI_Recv(&partSize, 1, MPI_INT, 0, ATAG, MPI_COMM_WORLD, &stat);
                int idxForReadWriteSwitch;
                MPI_Recv(&i, 1, MPI_INT, 0, ATAG, MPI_COMM_WORLD, &stat);
                MPI_Recv(&alpha, 1, MPI_DOUBLE, 0, ATAG, MPI_COMM_WORLD, &stat);
                MPI_Recv(&B[offset], partSize * (sizeDimension1), MPI_DOUBLE, 0,
                         ATAG, MPI_COMM_WORLD, &stat);
                (coordVector0[0] = i);
                (coordVector0[1] = ((i + 1) - coordVector0[0]));
                (idxForReadWriteSwitch = 3);
                MPI_Send(&idxForReadWriteSwitch, 1, MPI_INT, 0, FRTAG,
                          MPI_COMM_WORLD);
                MPI_Send(&coordVector0, 2, MPI_INT, 0, FRTAG, MPI_COMM_WORLD);
                MPI_Recv(&A[coordVector0[0]], coordVector0[1] * (sizeDimension1),
                          MPI_DOUBLE, 0, FRTAG, MPI_COMM_WORLD, &stat);
                (coordVector0[0] = i);
                (coordVector0[1] = (offset > (i + 1)) ?
                                    ((i + 1) - (i)) : (offset - (i)));
                if ((i) < offset && coordVector0[1] > 0) {
                    (idxForReadWriteSwitch = 2);
                    MPI_Send(&idxForReadWriteSwitch, 1, MPI_INT, 0, FRTAG,
                              MPI_COMM_WORLD);
                    MPI_Send(&coordVector0, 2, MPI_INT, 0, FRTAG,
                              MPI_COMM_WORLD);
                    MPI_Recv(&B[coordVector0[0]], coordVector0[1] * (sizeDimension1),
                              MPI_DOUBLE, 0, FRTAG, MPI_COMM_WORLD, &stat);
                }
                (coordVector0[0] = ((offset + partSize) > (i)) ?
                                    (offset + partSize) : (i));
                (coordVector0[1] = ((offset + partSize) > (i)) ?
                                    (i + 1 - (offset + partSize)) : (i + 1 - i));
                if ((i + 1) > offset + partSize && coordVector0[1] > 0) {
                    (idxForReadWriteSwitch = 2);
                    MPI_Send(&idxForReadWriteSwitch, 1, MPI_INT, 0, FRTAG,
                              MPI_COMM_WORLD);
                    MPI_Send(&coordVector0, 2, MPI_INT, 0, FRTAG,
                              MPI_COMM_WORLD);
                    MPI_Recv(&B[coordVector0[0]], coordVector0[1] * (sizeDimension1),
                              MPI_DOUBLE, 0, FRTAG, MPI_COMM_WORLD, &stat);
                }
                for (int j = offset; j < offset + partSize; ++j) {
                    for (k = 0; k < i; k++) {
                        (B[i][j] += alpha * A[i][k] * B[j][k]);
                    }
                }
                ...
```

Table 5.2 | Automatically generated OpenMP + MPI source code with HMPP kernels using the proposed tools. (1/2)

```
              ...
              (idxForReadWriteSwitch = 0);
              (coordVector0[0] = i < offset);
              (coordVector0[1] = (offset > i + 1) ?
                                  (i + 1 - i) : (offset - i));
              if ((i) <= offset && coordVector0[1] > 0) {
                  MPI_Send(&idxForReadWriteSwitch, 1, MPI_INT, 0, FWTAG,
                            MPI_COMM_WORLD);
                  MPI_Send(&coordVector0, 2, MPI_INT, 0, FWTAG,
                            MPI_COMM_WORLD);
                  MPI_Send(&B[coordVector0[0]], coordVector0[1] * (sizeDimension1),
                            MPI_DOUBLE, 0, FWTAG, MPI_COMM_WORLD);
              }
              (idxForReadWriteSwitch = 0);
              (coordVector0[0] = ((offset + partSize) > (i)) ?
                                  (offset + partSize) : (i));
              (coordVector0[1] = ((offset + partSize) > (i)) ?
                                  (i + 1 - (offset + partSize)) : (i + 1 - i));
              if ((i + 1) >= offset + partSize && coordVector0[1] > 0) {
                  MPI_Send(&idxForReadWriteSwitch, 1, MPI_INT, 0, FWTAG,
                            MPI_COMM_WORLD);
                  MPI_Send(&coordVector0, 2, MPI_INT, 0, FWTAG,
                            MPI_COMM_WORLD);
                  MPI_Send(&B[coordVector0[0]], coordVector0[1] * (sizeDimension1),
                            MPI_DOUBLE, 0, FWTAG, MPI_COMM_WORLD);
              }
              MPI_Send(&partSize, 1, MPI_INT, 0, ATAG, MPI_COMM_WORLD);
              MPI_Send(&offset, 1, MPI_INT, 0, ATAG, MPI_COMM_WORLD);
              MPI_Send(&B[offset], partSize * (sizeDimension1), MPI_DOUBLE, 0, ATAG,
                        MPI_COMM_WORLD);
          } else if (stat.MPI_TAG == FTAG) {
                  break;

    } } }
#pragma hmpp <group0_1> _instr_for1_ol_10_main advancedload, args[B]
#pragma hmpp <group0_1> _instr_for1_ol_10_main advancedload, args[A]
#pragma hmpp <group0_1> _instr_for1_ol_10_main callsite
        _instr_for1_ol_10_main(j, i, n, k, B, alpha, A);
#pragma hmpp <group0_1> _instr_for1_ol_10_main delegatedstore, args[B]
    }
    if (myid == 0) { ... }
    return 0;
}

#pragma hmpp <group0_1> _instr_for1_ol_10_main codelet, args[A].io=in
                                                       , args[B].io=inout
void _instr_for1_ol_10_main(int j, int i, int n, int k, double B[10][10],
                            double alpha, double A[10][10]) {
    for (j = i + 1; j < n; j++) {
        for (k = 0; k < i; k++) {
            B[i][j] += alpha * A[i][k] * B[j][k];
        }
    }
}
```

Table 5.3 | Automatically generated OpenMP + MPI source code with HMPP kernels using the proposed tools. (2/2)

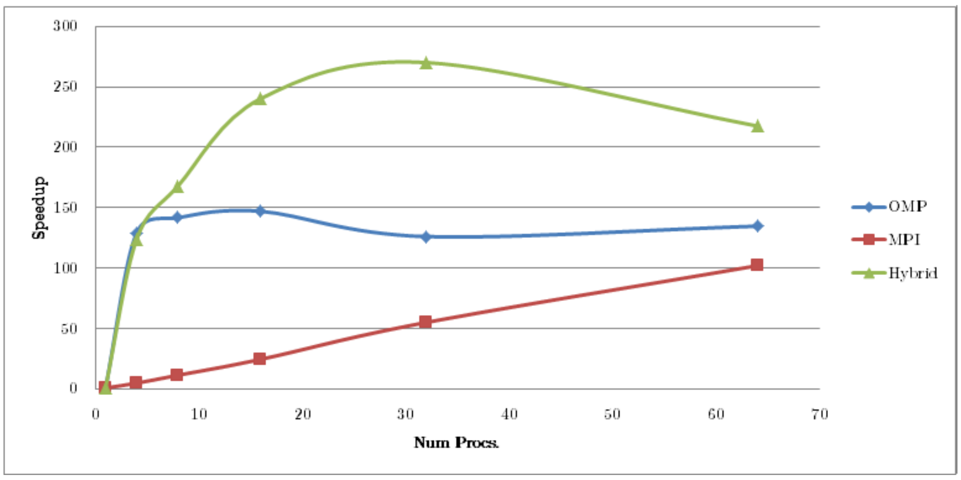


Figure 5.3 Speedup obtained for the sequential test problems and the hybrid model of the same problem using 16, 32, and 64 processors.

Conclusions | 6

The proposed tools were shown to be able to combine different programming models to automatically generate solutions for a variety of target hardware from parallel programming on shared memory architectures by the use of OpenMP, distributed memory architectures using MPI to HC combining the use of CPUs and GPGPUs and obtain correct solutions that could be orchestrated distributing computing tasks and loads on accelerator and HC platforms in an easy and effective way.

The proposed modular is translated into the implementation of a set of tools that facilitates the portability of a sequential source code to OpenMP, MPI and HMPP, showing how it effectively automatically translates input code. The generated codes allows to combine these as explained in Chapter 5. The presented tools allow to parallelize in an easy way sequential codes into OpenMP and to keep this code easily and readable to later transform to work on MPI or HMPP. S2S compilers allow to avoid learning of HMPP directives or MPI functions just compiling over one of compilers to take the advantages of the output models offer (speedup, scalability, etc.), and more important, obtains a good performance analysis that allows a smart selection of the best version according to its requirements. The readability of the code generated is acceptable so that allows further optimization by an expert intending to improve performance results.

I analyzed with our tools the version of Polybench benchmark implemented on [67], where I detected some codes not well parallelized using the tool presented in Chapter 2 as the TRMM problem presented on that section.

The results of OMP2MPI tool of the correctly parallelized problems in Chapter 4 where I detected that the speedup figures for 64 cores in some of the cases are higher than 60× compared to the sequential version, and also higher than the original OpenMP code. These results show again, as mentioned in the introduction, that OpenMP does not always perform better than MPI in shared memory systems.

And the results of a sub-set of problems of these problems using the tool OMP2HMPP. In that experiment I obtain an average speedup of 31× and an average increase of energy efficiency of 5.86×, comparing the generated version with the best results coming from OpenMP version. OMP2HMPP tool produces solution that rarely differ from the best HMPP hand-coded version. I notice that a CUDA hand-coded version CUDA obtains an speedup near 1.7× compared with the version the best speedup of OMP2HMPP.

The set of presented tools could be combined as presented on Chapter 5, even that the results doesn't give a considerable speedup improvement in the selected case when I tested combination of OpenMP and MPI programming model combining the use of OpenMP and MPI. However, this able to explore without effort the possible combinations of that two languages.

The bankruptcy of CAPS enterprises makes it impossible to combine GPGPUs with OpenMP or MPI, nonetheless future improvements are considered to make that transformation possible. This would preserve the modularity of our system, so that it is more adaptable to changes in a domain that is not yet stable.

6.1 Open Research

As mentioned above, one of the main problems with the presented tools is with the OMP2HMPP tool, in which the selected target language of our S2S compiler has been halted because of the economic difficulties of CAPS Enterprise. For that reason, future changes for that tool may use two possible target programming languages.

On one hand, OpenACC or CUDA is a possible language target. The first one was developed to be a programming standard for parallel computing developed by Cray, CAPS, Nvidia, and PGI. It was designed to simplify the parallel programming of heterogeneous CPU/GPU systems and merge into the OpenMP specification to create a common specification that extends OpenMP to accelerator support in a future release. Considering that OpenMP will remain the starting point of the modified tool, this is one of the easiest ways to improve it. OpenACC directives are quite similar to HMPP directives, which are used to annotate C/C++ source code to identify the areas that should be accelerated using compiler directives and additional functions. Hence, this task will not take significant effort.

On the other hand, because of this versatility, I plan to consider OpenCL because it explores HC using a framework that allows programs to be written that are executed across heterogeneous platforms consisting of CPUs, GPUs, digital signal processors, FPGAs, and other processors. Hence, OpenCL specifies a language (based on C99) for programming these devices and Application Programming Interfaces (API)s to control the platform and execute programs on the compute devices. OpenCL provides parallel computing using task-based and data-based parallelism.

For distributed memory computing, I would like to offer a new possible solution using MCAPI [20]. This solution, in comparison to MPI, focuses purely on embedded communications, allowing MCAPI to support various quality of service, where connected channels may exploit underlying embedded hardware. MCAPI provides a standardized API for communication and synchronization between closely distributed (multiple cores on a chip and/or chips on a board) embedded systems. This will enable better HC solutions to be explored.

Because it is designed as a modular system, in future experiments, I plan to combine the generated solutions, exploring all the implemented tools, always with the objective of automatically and efficiently orchestrating the distribution of computing tasks and loads on accelerator and heterogeneous computing platforms.

References

[1] J. L. Hennessy and D. A. Patterson. *Computer architecture: a quantitative approach.* Elsevier, 2011.

[2] I. PRESENT. *Cramming more components onto integrated circuits.* Readings in computer architecture, page 56, 2000.

[3] OpenMP. *The openmp api*, March 1997. URL `http://openmp.org/wp/`.

[4] S. Williams, A. Waterman and D. Patterson. *Roofline: an insightful visual performance model for multicore architectures.* Communications of the ACM, 52(4):65–76, 2009.

[5] G. E. Moore. *Lithography and the future of moore's law.* In *SPIE's 1995 Symposium on Microlithography*, pages 2–17. International Society for Optics and Photonics, 1995.

[6] G. Krawezik. *Performance comparison of mpi and three openmp programming styles on shared memory multiprocessors.* In *Proceedings of the fifteenth annual ACM symposium on Parallel algorithms and architectures*, pages 118–127. ACM, 2003.

[7] D. Buono, T. De Matteis, G. Mencagli and M. Vanneschi. *Optimizing message-passing on multicore architectures using hardware multi-threading.* 22nd Euromicro International Conference on Parallel, Distributed and Network-Based Processing (PDP), pages 262–270, 2014.

[8] MPI. *Ibm developerworks*, 2018. URL `https://www.mpi-forum.org/docs/mpi-1.3/mpi-report-1.3-2008-05-30.pdf`.

[9] C. Willard, A. Snell, L. Sergervall and M. Feldman. *Hpc user site census: Systems*, July 2013. URL `http://www.intersect360.com/industry/downloadsummary.php?id=92`.

[10] D. Castells-Rufas and J. Carrabina. *128-core many-soft-core processor with mpi support.* In *Jornadas de ComputaciÃşn Reconfigurable y Aplicaciones (JCRA).* 2015.

[11] NVIDIA. *Cuda sdk*, March 2007. URL `https://developer.nvidia.com/cuda-downloads`.

[12] CAPS. *Openhmpp directives*, March 2007. URL `http://www.caps-entreprise.com/openhmpp-directives/`.

[13] R. Dolbeau, S. Bihan and F. Bodin. *Hmpp: A hybrid multi-core parallel programming environment.* In *Workshop on General Purpose Processing on Graphics Processing Units (GPGPU 2007).* 2007.

[14] Rapid. *home page*, March 2009. URL `http://www.rapidmind.net/`.

[15] OpenACC Working Group. *The OpenACC Application Programming Interface, Version 1.0*, November 2011. URL `http://www.openacc-standard.org/`.

[16] PeakStream. *home page*, March 2006. URL `http://www.peakstream.com/`.

[17] A. CTM. *Technical reference manual*, March 2006. URL `http://ati.amd.com/companyinfo/researcher/documents/ATI_CTM_Guide.pdf`.

[18] AMD. *Amd developer*, August 2014. URL `http://developer.amd.com/resources/heterogeneous-computing/what-is-heterogeneous-system-architecture-hsa/`.

[19] M. P. Forum. *Mpi: A message-passing interface standard.* Technical report, Knoxville, TN, USA, 1994.

[20] T. M. Association. *Multicore communications api*, March 2014. URL `http://www.multicore-association.org/workgroup/mcapi.php`.

[21] E. Fernandez-Alonso, D. Castells-Rufas, S. Risueño, J. Carrabina and J. Joven. *A noc-based multi-{soft} core with 16 cores.* In *Electronics, Circuits, and Systems (ICECS), 2010 17th IEEE International Conference on*, pages 259–262. IEEE, 2010.

[22] C. Lattner and V. Adve. *Llvm: A compilation framework for lifelong program analysis & transformation.* In *Code Generation and Optimization, 2004. CGO 2004. International Symposium on*, pages 75–86. IEEE, 2004.

[23] C. Ancourt, F. Coelho, B. Creusillet, F. Irigoin, P. Jouvelot and R. Keryell. *Pips: A framework for building interprocedural compilers, parallelizers and optimizers.* Technical report, Technical Report A/289, Centre de Recherche en Informatique, Ecole des Mines de Paris, 1996.

[24] C. Dave, H. Bae, S.-J. Min, S. Lee, R. Eigenmann and S. Midkiff. *Cetus: A source-to-source compiler infrastructure for multicores.* Computer, 42(12):36–42, 2009.

[25] D. Quinlan. *Rose: Compiler support for object-oriented frameworks.* Parallel Processing Letters, 10(02n03):215–226, 2000.

[26] J. Balart, A. Duran, M. Gonzàlez, X. Martorell, E. Ayguadé and J. Labarta. *Nanos mercurium: a research compiler for openmp.* In *Proceedings of the European Workshop on OpenMP*, volume 8. 2004.

[27] E. D. Willink. *Meta-compilation for c+.* 2001.

[28] Nanos. *Mercurium*, March 2004. URL `https://pm.bsc.es/projects/mcxx`.

[29] Y. Qian. *Automatic parallelization tools.* In *Proceedings of the World Congress on Engineering and Computer Science*, volume 1. 2012.

[30] D. A. Padua, R. Eigenmann, J. Hoeflinger, P. Petersen, P. Tu, S. Weatherford and K. Faigin. *Polaris: A new-generation parallelizing compiler for mpps.* In *In CSRD Rept. No. 1306. Univ. of Illinois at Urbana-Champaign.* Citeseer, 1993.

[31] H.-S. Kim, Y.-H. Yoon, S.-O. Na and D.-S. Han. *Icu-pfc: an automatic parallelizing compiler.* In *High Performance Computing in the Asia-Pacific Region, 2000. Proceedings. The Fourth International Conference/Exhibition on*, volume 1, pages 243–246. IEEE, 2000.

[32] R. Allen and K. Kennedy. *Automatic translation of fortran programs to vector form.* ACM Transactions on Programming Languages and Systems (TOPLAS), 9(4):491–542, 1987.

[33] M. Amini, B. Creusillet, S. Even, R. Keryell, O. Goubier, S. Guelton, J. O. McMahon, F.-X. Pasquier, G. Péan, P. Villalon et al. *Par4all: From convex array regions to heterogeneous computing.* In *IMPACT 2012: Second International Workshop on Polyhedral Compilation Techniques HiPEAC 2012.* 2012.

[34] U. Bondhugula, A. Hartono, J. Ramanujam and P. Sadayappan. *Pluto: A practical and fully automatic polyhedral program optimization system.* In *Proceedings of the ACM SIGPLAN 2008 Conference on Programming Language Design and Implementation (PLDI 08), Tucson, AZ (June 2008).* Citeseer, 2008.

[35] E. Kallel, Y. Aoudni and M. Abid. *Openmp automatic parallelization tools: An empirical comparative evaluation.* International Journal of Computer Science Issues (IJCSI), 10(4), 2013.

[36] IBM. *Ibm developerworks*, August 2013. URL `https://goo.gl/s8555T`.

[37] Intel. *Intel parallel studio xe 2015*, August 2015. URL `https://software.intel.com/en-us/intel-parallel-studio-xe`.

[38] O. Brewer, J. Dongarra and D. Sorensen. *Tools to aid in the analysis of memory access patterns for fortran programs.* Parallel Computing, 9(1):25–35, 1988.

[39] V. Subotic, A. Campos, A. Velasco, E. Ayguade, J. Labarta and M. Valero. *Tareador: The unbearable lightness of exploring parallelism.* In *Tools for High Performance Computing 2014*, pages 55–79. Springer, 2015.

[40] M. Ishihara, H. Honda, T. Yuba and M. Sato. *Interactive parallelizing assistance tool for openmp: ipat/omp.* In *Proc. Fifth European Workshop on OpenMP (EWOMP âĂŸ03)*, pages 21–29. 2003.

[41] M. Corina. *Quantitative analysis and visualization of memory access patterns.* Ph.D. thesis, TU Delft, Delft University of Technology, 2010.

[42] C. Bastoul. *Code generation in the polyhedral model is easier than you think.* In *Proceedings of the 13th International Conference on Parallel Architectures and Compilation Techniques*, pages 7–16. IEEE Computer Society, 2004.

[43] M. Sato, S. Satoh, K. Kusano and Y. Tanaka. *Design of openmp compiler for an smp cluster.* In *Proc. of the 1st European Workshop on OpenMP*, pages 32–39. 1999.

[44] A. Basumallik, S.-J. Min and R. Eigenmann. *Towards openmp execution on software distributed shared memory systems.* In *High Performance Computing*, pages 457–468. Springer, 2002.

[45] V. Schuster and D. Miles. *Distributed openmp, extensions to openmp for smp clusters.* In *Proc. of the Workshop on OpenMP Applications and Tools (WOMPAT 2000).* 2000.

[46] J. P. Hoeflinger. *Extending openmp to clusters.* White Paper, Intel Corporation, 2006.

[47] Y.-S. Kee, J.-S. Kim and S. Ha. *Parade: An openmp programming environment for smp cluster systems.* In *Proceedings of the 2003 ACM/IEEE conference on Supercomputing*, page 6. ACM, 2003.

[48] R. Eigenmann, A. Basumallik, S.-J. Min, J. Hoeflinger, R. H. Kuhn, D. Padua and J. Zhu. *Is openmp for grids?* In *Parallel and Distributed Processing Symposium, International*, volume 2, pages 0171b–0171b. IEEE Computer Society, 2002.

[49] A. Duran, J. M. Perez, E. Ayguadé, R. M. Badia and J. Labarta. *Extending the openmp tasking model to allow dependent tasks.* In *OpenMP in a New Era of Parallelism*, pages 111–122. Springer, 2008.

[50] B. R. Gaster. *Streams: Emerging from a shared memory model.* In *OpenMP in a New Era of Parallelism*, pages 134–145. Springer, 2008.

[51] A. Pop, S. Pop, H. Jagasia, J. Sjödin and P. H. Kelly. *Improving gnu compiler collection infrastructure for streamization.* Proceedings of the 2008 GCC Developers Summit, pages 77–86, 2008.

[52] A. Basumallik and R. Eigenmann. *Towards automatic translation of openmp to mpi.* In *Proceedings of the 19th annual international conference on Supercomputing*, pages 189–198. ACM, 2005.

[53] D. Millot, A. Muller, C. Parrot and F. Silber-Chaussumier. *Step: a distributed openmp for coarse-grain parallelism tool.* In *OpenMP in a New Era of Parallelism*, pages 83–99. Springer, 2008.

[54] Khronos OpenCL Working Group. *The OpenCL Specification, version 1.0.29*, 8 December 2008. URL `http://khronos.org/registry/cl/specs/opencl-1.0.29.pdf`.

[55] A. Saà-Garriga, D. Castells-Rufas and J. Carrabina. *Omp2hmpp: Compiler framework for energy-performance trade-off analysis of automatically generated codes.* International Journal of Computer Science Issues (IJCSI), 12(2):9, 2015.

[56] A. Saà-Garriga, D. Castells-Rufas and J. Carrabina. *Omp2mpi: Automatic mpi code generation from openmp programs.* High Performance Energy Efficient Embedded Systems (HIP3ES), 2015.

[57] S. Lee, S.-J. Min and R. Eigenmann. *Openmp to gpgpu: a compiler framework for automatic translation and optimization.* ACM Sigplan Notices, 44(4):101–110, 2009.

[58] T. D. Han and T. S. Abdelrahman. *hi cuda: a high-level directive-based language for gpu programming.* In *Proceedings of 2nd Workshop on General Purpose Processing on Graphics Processing Units*, pages 52–61. ACM, 2009.

[59] H.-F. Li, T.-Y. Liang and J.-L. Jiang. *An openmp compiler for hybrid cpu/gpu computing architecture.* In *Intelligent Networking and Collaborative Systems (INCoS), 2011 Third International Conference on*, pages 209–216. IEEE, 2011.

[60] E. Ayguadé, R. M. Badia, P. Bellens, D. Cabrera, A. Duran, R. Ferrer, M. Gonzàlez, F. Igual, D. Jiménez-González, J. Labarta et al. *Extending openmp to survive the heterogeneous multi-core era.* International Journal of Parallel Programming, 38(5-6):440–459, 2010.

[61] P. Cantiello, B. Di Martino and F. Piccolo. *Unimodular loop transformations with source-to-source translation for gpus.* In *Algorithms and Architectures for Parallel Processing*, pages 186–195. Springer, 2013.

[62] D. Castells-Rufas. *Scalable Parallel Architectures on Reconfigurable Platforms.* Ph.D. thesis, Microelectronics and Electronic Systems Department, Universitat Autònoma de Barcelona, 2015.

[63] PolyBench. *The polyhedral benchmark suite*, March 1997. URL `http://web.cse.ohio-state.edu/~pouchet/software/polybench/`.

[64] E. Ayguade, R. M. Badia, D. Cabrera, A. Duran, M. Gonzalez, F. Igual, D. Jimenez, J. Labarta, X. Martorell, R. Mayo et al. *A proposal to extend the openmp tasking model for heterogeneous architectures.* In *Evolving OpenMP in an Age of Extreme Parallelism*, pages 154–167. Springer, 2009.

[65] A. Saà-Garriga. *Github repository: Automatic parallelization*, 2014. URL `https://github.com/sdruix/AutomaticParallelization/`.

[66] CAPS. *Caps entreprise bankrupt*, March 2014. URL `http://www.produits-c-s.fr/Nous-ne-commercialisons-plus-les-produits-CAPS_a143.html`.

[67] S. Grauer-Gray, L. Xu, R. Searles, S. Ayalasomayajula and J. Cavazos. *Autotuning a high-level language targeted to gpu codes.* In *Innovative Parallel Computing (InPar), 2012*, pages 1–10. IEEE, 2012.