

**IDENTIFICAÇÃO AUTOMÁTICA DE TAREFAS
EM PROGRAMAS ESTRUTURADOS**

PEDRO HENRIQUE RAMOS COSTA

IDENTIFICAÇÃO AUTOMÁTICA DE TAREFAS EM PROGRAMAS ESTRUTURADOS

Proposta de dissertação apresentada ao Programa de Pós-Graduação em Ciência da Computação do Instituto de Ciências Exatas da Universidade Federal de Minas Gerais como requisito parcial para a obtenção do grau de Mestre em Ciência da Computação.

ORIENTADOR: FERNANDO MAGNO QUINTÃO PEREIRA

Belo Horizonte

Julho de 2018

PEDRO HENRIQUE RAMOS COSTA

AUTOMATIC MINING OF TASKS IN STRUCTURED PROGRAMS

Dissertation proposal presented to the
Graduate Program in Computer Science of
the Federal University of Minas Gerais in
partial fulfillment of the requirements for
the degree of Master in Computer Science.

ADVISOR: FERNANDO MAGNO QUINTÃO PEREIRA

Belo Horizonte

July 2018

Ficha catalográfica elaborada pela Biblioteca do ICEx - UFMG

Costa, Pedro Henrique Ramos

C837a Automatic mining of tasks in structured programs / Pedro
Henrique Ramos Costa. — Belo Horizonte, 2018.
xxii, 62 p.: il.; 29 cm.

Dissertação (mestrado) - Universidade Federal
de Minas Gerais – Departamento de Ciência da
Computação.

Orientador: Fernando Magno Quintão Pereira

1. Computação – Teses. 2. Paralelismo. 3.
Processamento paralelo. 4. Tarefas. I. Fernando Magno
Quintão Pereira. II.
Automatic mining of tasks in structured programs.

CDU 519.6*31 (043)



UNIVERSIDADE FEDERAL DE MINAS GERAIS
INSTITUTO DE CIÊNCIAS EXATAS
PROGRAMA DE PÓS-GRADUAÇÃO EM CIÊNCIA DA COMPUTAÇÃO

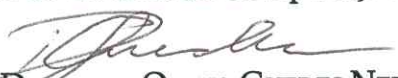
FOLHA DE APROVAÇÃO

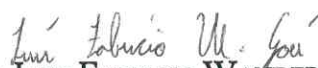
Automatic Mining of Tasks in Structured Programs

PEDRO HENRIQUE RAMOS COSTA

Dissertação defendida e aprovada pela banca examinadora constituída pelos Senhores:


PROF. FERNANDO MAGNO QUINTÃO PEREIRA - Orientador
Departamento de Ciência da Computação - UFMG


PROF. DORGIVAL OLAVO GUEDES NETO
Departamento de Ciência da Computação - UFMG


PROF. LUIS FABRICIO WANDERLEY GOES
Departamento de Ciência da Computação - PUC-MG

Belo Horizonte, 16 de julho de 2018.

To my mom, for always believing in me, and to my grandpa, for being my first teacher in life.

Acknowledgments

Firstly, I would like to thank Cassia, my mother, for always being supportive of every tough decision I have ever made in life. Thank you, mom. You are my best friend. I should also thank my grandfather Elmo, long gone in matter but forever alive in my heart and mind. Grandpa, you might never read this, but you taught me how to dream.

Both of you are my stronghold. I would also like to thank my father Alberto, for loving my mother, nourishing and cherishing her whenever he could. I thank you for caring in your own way. I thank Tita, for being my ally. You have made me feel protected and beloved.

Still on family territory, I thank Isabela, my cousin, for always being close when I needed and for making me feel accepted like I had never felt before. To my aunts Junia, Eliane, Adriana, and Thais, my godmother, I am grateful for all the emotional and financial support throughout college years. I thank Dedei, for sharing so many common interests and beliefs. I wish I could discuss about physics and philosophy with you, and share all my acquired knowledge. After all, you might be the only one in our family who would understand that. I also thank my cousin Matheus, for the late night talks; Lorena, for letting me play on her computer when I didn't have one – this certainly helped my career choice; Luciana, for being a role model; and Samuel and Alice, my sweet goddaughter, for bringing joy and color to our family.

This work wouldn't be real if it weren't for my professor and advisor Fernando Magno. Thank you for believing in me and for being the best advisor I could ever hope for. You were my Gandalf. I am forever in your debt. I would also like to thank each and every professor from the Computer Science Department, for inspiring me even under unrelenting circumstances. I thank Cesar and professor Guido, from Unicamp, for all the support in the makings of the TaskMiner.

It's also important to state that, in this quest, I was certainly accompanied by strong allies: my friends. I thank Cynthia, the girl who early awakened a fascination for computer science in me. If I'm here today, it's because you told me to keep trying.

I love you. I also thank these people, in no necessary order: Alessandra, for showing me value in the simple things around me; Natalia, for teaching me how to laugh at life; Lais, for leading me to the noble profession of teaching; Isabela, for sharing thoughts without speaking; Tamires, on practicing forgiveness, the true meaning of friendship; Rayanna, for always being so patient and for all the reinforcement; Cairo, for putting up with me as long as he could; Clara, for being my young sis and best friend on Twitter; Talita, for all the nights we've spent together; Bia, for sharing a home and many laughs; Vitor, for carrying me to the Compilers Lab and teaching me how to get to know myself; Hamilton, for never letting me forget the art and creativity that grows inside me; Mari, for being so considerate of my decisions; and Daphne, for the unconditional trust and for teaching me the meaning of perseverance.

I also thank Fabricio, Patricia and Bernardo, for being my first fellows during undergrad. Thank you guys for all the coffee breaks. I thank the Compilers Lab squad: Tarsila, Carina, Breno, Junio, Guilherme, Gleison, Andrei, Matheus, Hugo, Bruno, Leandro, Caio, Rubens, Gabriel and Yukio. I'll miss all the afternoons we spent together. Also here go my shoutouts to all the guys from the Telegram group, specially to Rafael, Jose, Joao, Matheus and Nildo. I should also thank the London Squad: Maria Clara, Marcus, Joao Vitor, Ana Beatriz, Henrique, Clara and Karina. Thank you for all the trips and the funny and difficult moments we shared. Last but not least, I thank all my friends that, despite the distance now, were close back then: Roberto, Samuel, Luiza, Ruan, Renan and all my TFLA friends and teachers.

Finally, I truly thank the committee for reading and assessing the present written work. It was made with extreme devotion and dedication, and I genuinely hope this is clear throughout the reading of this piece.

“There are no facts, only interpretations.”
(Friedrich Nietzsche)

Resumo

Esta dissertação descreve o desenvolvimento e implementação de um conjunto de análises estáticas e técnicas de geração de código para anotar programas com cláusulas OpenMP que evidenciam paralelismo de tarefas. O objetivo deste trabalho se encontra dentro do escopo de paralelização automática de código. Foram implementadas técnicas para identificação de paralelismo de tarefas em código C, e utilizou-se o sistema de anotação OpenMP para anotar o código fonte original e conferir semântica paralela a um programa originalmente escrito em um paradigma sequencial. As técnicas implementadas determinam os intervalos de memória cobertos pela região de código a ser paralelizada, limitam o número de tarefas recursivas ativas e estimam a lucratividade de tarefas candidatas. Essas ideias foram implementadas em uma ferramenta chamada TaskMiner, um compilador fonte-a-fonte capaz de inserir pragmas OpenMP em programas C/C++ sem intervenção humana. TaskMiner é construído sobre os alicerces das análises estáticas de código, e se apoia no ambiente de execução do OpenMP para desambiguar ponteiros. TaskMiner anota programas longos e complexos, e frequentemente replica os ganhos de performance obtidos através da anotação manual nesses programas. Além disso, as técnicas implantadas no TaskMiner concedem-nos um meio de descobrir oportunidades de paralelismo escondidas por muitos anos na sintaxe de benchmarks conhecidos, às vezes levando a ganhos de velocidade de até 400% em uma máquina de 12 núcleos, sem nenhum custo extra de programação.

Palavras-chave: paralelismo, OpenMP, tarefas.

Abstract

This dissertation describes the design and implementation of a suit of static analyses and code generation techniques to annotate programs with OpenMP pragmas for task parallelism. These techniques approximate the ranges covered by memory regions, bound recursive tasks and estimate the profitability of tasks. These ideas have been implemented in a tool called TaskMiner, a source-to-source compiler that inserts OpenMP pragmas into C/C++ programs without any human intervention. By building onto the static program analysis literature, and relying on OpenMP’s runtime ability to disambiguate pointers, TaskMiner is able to annotate large and convoluted programs, often replicating the performance gains of handmade annotation. Furthermore, the techniques employed in TaskMiner give us the means to discover opportunities of parallelism that remained buried in the syntax of well-known benchmarks for many years – sometimes leading to up to four-fold speedups on a 12-core machine at zero programming cost.

Palavras-chave: parallelism, OpenMP, tasks.

List of Figures

11	TaskMiner diagram. A originally sequential C program is treated as input by the TaskMiner. The result of the TaskMiner's process is the original program annotated with OpenMP task directives, able to execute in parallel in any OpenMP runtime and make full use of a many-core architecture.	3
12	Reduction example. The sum of elements is accumulated into V[0].	4
13	Sum reduction tree. Pair of values can be summed up separately and in parallel.	4
14	Reduction example annotated with an OpenMP pragma	5
15	Code for the N Queens problem.	6
16	Mergesort is a classic example of task based parallelism: each recursive call is independent and can run in parallel.	11
21	The benefits of OpenMP's annotations. Annotations appear in lines 6, 7, 9, 11.	16
31	Identifying memory regions with symbolic limits, and using runtime information to estimate the profit of tasks.	19
32	Task annotation with input memory dependencies.	19
33	Bounding the creation of recursive tasks. Example taken from [44, Fig.1]. .	21
34	Variable j must be replicated among tasks, to avoid the occurrence of data races.	22
41	Hammock region.	24
42	Memory regions annotated as <i>depend</i> clauses.	25
43	High level abstraction of the TaskMiner algorithm.	25
44	Program Dependence Graph for a given program. Solid edges represent data dependencies and dashed edges represent control dependencies.	26
45	Control Flow Graph for a given program. Instructions are logically organized in basic blocks.	27

46	Memory regions as symbolic limits.	28
47	Examples of windmills and vanes in program dependence graphs.	30
48	Estimating the cost of tasks.	32
49	Task discovery via expansion of hammock regions. COST is the overhead of creating and scheduling threads.	34
410	Example of Task Expansion.	35
411	Variables <i>j</i> and <i>i</i> are replicated among tasks, avoiding data races.	36
412	Bounding the creation of recursive tasks.	39
51	Speedup comparisons between programs annotated by TaskMiner	42
52	Benefit of task pruning. RFC : tasks created within <u>R</u> ecursive <u>F</u> unction <u>C</u> alls. NRC : interprocedural tasks created around <u>N</u> on- <u>R</u> ecursive function <u>C</u> alls. Reg : tasks involving <u>R</u> egions without function calls.	43
53	Relation between number of task regions and program size. Each benchmark is a complete C file.	45
54	Speedups (in number of times) obtained by TaskMiner when applied onto the LLVM test suite. The larger the bar, the better. LoC stands for “Lines of Code”.	45
55	Runtime of TaskMiner vs size of input programs.	46

List of Tables

Contents

Acknowledgments	ix
Resumo	xiii
Abstract	xv
List of Figures	xvii
List of Tables	xix
1 Introduction	1
1.1 Context	1
1.2 Motivation	3
1.2.1 Publications	7
1.3 Parallelism	8
1.3.1 Challenges in parallel programming	8
1.3.2 Types of parallelism	9
1.3.3 Automatic parallelization of code	11
2 The OpenMP System	13
2.1 Annotation	13
2.1.1 Directives	14
2.1.2 Clauses	14
2.2 The runtime	15
3 The Problem	17
3.1 Overview	17
3.2 Memory dependencies	18
3.3 Profitability of Tasks	20
3.4 Bounding the number of Tasks	20

3.5	Concurrency of Tasks	21
4	The TaskMiner	23
4.1	Definitions	23
4.2	The algorithm	25
4.2.1	Memory range analysis	27
4.2.2	Mapping code regions to tasks	29
4.2.3	Estimating the profitability of tasks	31
4.2.4	Task expansion	33
4.2.5	Privatization analysis	35
4.2.6	Mapping it all back into source-code	37
5	Experiments	41
5.1	Evaluation overview	41
5.2	TaskMiner’s performance	42
5.3	TaskMiner’s optimizations	43
5.4	TaskMiner’s versatility	44
5.5	TaskMiner’s scalability	46
6	Related work	47
7	Conclusion	49
	Bibliography	53

Chapter 1

Introduction

1.1 Context

It is undeniable that parallel computing is, nowadays, the most pursued alternative by developers when striving for more application performance. For years the programming community has invested in parallel paradigms to achieve greater speed and efficiency [42]. Programmers already know how to easily determine iteration independence and data locality in programs that are based on vector and matrix operations [67]. Therefore, today we have, at our disposal, many powerful tools to exploit parallelism in these sort of programs.

Although the community has reached amazing levels of parallelism abstraction for regular programs, little is known about data locality and patterns of parallelism in irregular code [51]. Irregular code is usually based on pointer data structures such as graphs and trees, making it hard to determine dependences statically. Thus, parallelism opportunities in this sort of program can only be truly identified at runtime [67]. Nevertheless, extensive work has been done to scavenge parallelism in regular applications [51]. However, finding good parallelism opportunities in irregular code remains a grueling task.

Programming parallel applications is also, undoubtedly, a hard task, as programmers ought to take care of many aspects of the job such as identifying and keeping variable dependences correctly and assuring synchronization and scalability of threads and processes, while still avoiding race conditions. Even more, finding parallelism in programs that were not coded in a parallel paradigm is not simple, as compiler static analyses are not precise enough to identify parallel regions [52].

A way to approach this problem is to use a programming model that can capture and enforce dependencies between iterations at runtime whenever they occur.

In such model, those iterations that do not have *loop-carried* dependencies are free to run in parallel, while iterations that dynamically create loop-carried dependencies are serialized and dispatched in dependence order by a system runtime. In recent years, task-based execution models have proven themselves to be a scalable and quite flexible approach to extract regular and irregular parallelism from sequential code [68; 84; 13; 35; 76; 47; 14; 9]. In this model, the programmer uses a *task* directive to mark code regions in the program that are potential tasks and lists their corresponding dependencies. The OpenMP task scheduling model [5] is an example of such model. Although the OpenMP task construct simplifies the mechanics of dispatching and running tasks, the burden of finding code regions to be annotated with task directives still lies upon the programmer. ++ Work has been carried out to generate OpenMP annotations automatically given a set of instructions that define a region [61; 67; 89]. However, there is not a known algorithm to automatically extract those parallel regions from loops with loop-carried dependencies.

This dissertation presents a set of program analyses that merge static data structures such as the Program Dependence Graph (PDG) [36] and the Control Flow Graph (CFG) with runtime information to automatically extract task-based parallelism from irregular code. These analyses have been implemented in a tool called **TaskMiner**: a source-to-source compiler that receives a C program as input and returns the same code annotated with task-based directives that identify the parallel regions.

Figure 11 illustrates, with a diagram, the **TaskMiner**'s use in practice. The tool is used to scavenge parallelism in sequential C code. Therefore, a program written in pure C language, originally in a sequential paradigm, enters as input on the **TaskMiner**. Our tool performs a series of analyses and transformations, and the resulting output is the same program annotated with OpenMP task directives. These and other parallelism-related directives are placed at the sites pointed out by **TaskMiner**'s analyses. The tool **TaskMiner** only points out hidden task parallelism opportunities when they are present, i. e., it doesn't break correctness. Thus, it won't find anything worth parallelizing in a strictly sequential program. The output program is always semantically equivalent to the original program. The OpenMP task directives do add to the program semantics, but are completely optional: the new program can still be executed sequentially and the directives can be bypassed, should the programmer wishes to do so. Due to the coarse-grained nature of the parallelism discovered by **TaskMiner**, this dissertation claims that the use of the tool can bring great advantages when trying to improve performance of programs that are deemed sequential, but have got hidden parallelism, by enabling them to run more efficiently on many-core architectures.

Before we delve into the works of the **TaskMiner**, the author of this dissertation

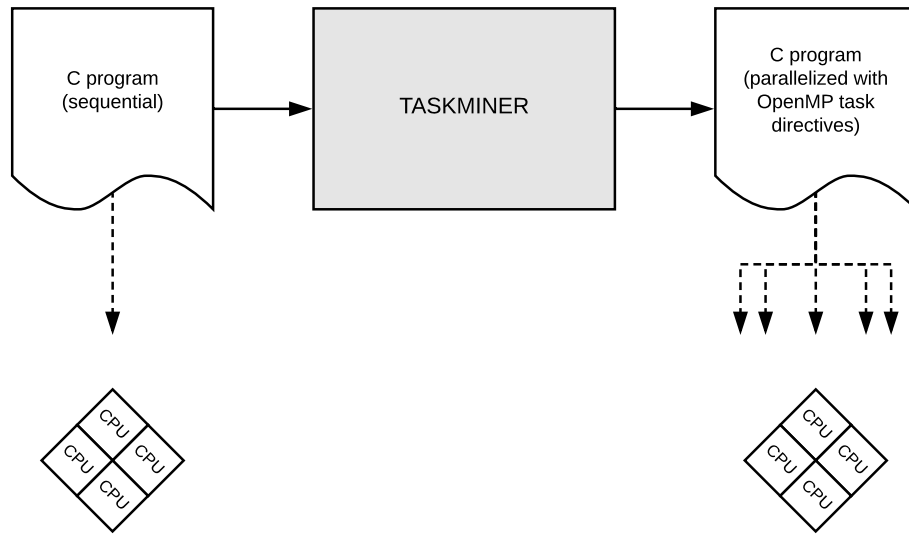


Figure 11. TaskMiner diagram. A originally sequential C program is treated as input by the TaskMiner. The result of the TaskMiner’s process is the original program annotated with OpenMP task directives, able to execute in parallel in any OpenMP runtime and make full use of a many-core architecture.

makes some subjective considerations in the next section. The speech may switch from first-person plural to first-person singular during the Section 1.2. Afterwards, the speech will remain in the first-person plural for the rest of the dissertation.

1.2 Motivation

The first time I came in contact with the topic of automatic parallelization of code I was still fighting for my undergraduate degree in Computer Science at the Federal University of Minas Gerais (UFMG). Professor Fernando, my current advisor, who was, at the time, also my Professor at the subject of *Static Code Analysis*, proposed the problem of finding *reduction patterns* in C code. The term *reduction* has its origins in functional programming. More precisely, it comes from the *fold* operation. Simply, a *reduction* mainly consists in performing repeated operations in a collection of elements — an array, for example — and storing the result of those operations into one accumulator. See the code listed in Figure 12, for example.

```

1 void reduce(int* V, int N) {
2     int i = 1;
3     for ( ; i < N; i++) {
4         V[0] += V[i];
5     }
6 }

```

Figure 12. Reduction example. The sum of elements is accumulated into $V[0]$.

In that piece of code, a loop traverses the array V and stores the sum of all elements into its first position $V[0]$. Maybe it is not so clear to see that the iterations of this loop can be parallelized. At first glance, it might appear that every iteration but the first depends on the previous, since the operator $+=$ performs a READ/WRITE operation. Hence, it is expected from a compiler to declare this loop strictly sequential, after all, there are clear dependencies in each iteration. In order to compute $V[i]$, we depend on the value of $V[i - 1]$ and so on.

However, it is known that the sum operator $(+)$ is commutative, which means that the operation is agnostic to the order of the operands. Figure 13 shows how a simple sum reduction can be parallelized if we compute different parts of the sum concurrently.

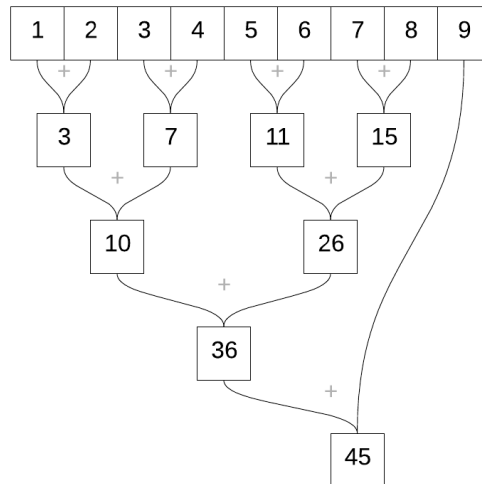


Figure 13. Sum reduction tree. Pair of values can be summed up separately and in parallel.

This type of reduction is classic and parallelizing it has, by all means, already been

solved by different approaches. One famous approach allows us to use annotation-based systems, such as OpenMP (whose behavior shall be scrutinized later in this dissertation) to point out such reductions patterns to the compiler. With human help, the compiler can then parallelize it. In Figure 14, we annotate the line in which the reduction occurs with a **pragma**. This annotation is only available when using those annotation-based systems with a compiler of choice. Evidently, the chosen compiler must know how to interpret those **pragmas**. For the example in question, we use OpenMP and compile the code with gcc-6, which supports all clauses listed in OpenMP version 4.5.

```
1 void reduce(int* V, int N) {  
2     int i = 1;  
3     #pragma omp parallel for reduction(+:V[0])  
4     for ( ; i < N; i++) {  
5         V[0] += V[i];  
6     }  
7 }
```

Figure 14. Reduction example annotated with an OpenMP **pragma**.

Therefore, the essence of finding parallelism in sequential code lies in telling the compiler that some data dependencies stated statically are not real or are very unlikely to occur during execution. Moreover, it lies in the ability to determine a more precise dependence analysis, less conservative, that allows us to find independent instructions amidst sequential code.

It's not easy to beat compilers when it comes to data dependence analyses. There are many aspects involved in figuring out more precise and less conservative dependence analyses, that range from pointer aliasing analyses to a variety of control-flow analyses. When researching though, I broadened the topic of my Masters with one single question: how can I find potentially independent instructions hidden in code that was deemed strictly dependent by the compiler?

The OpenMP system allows the use of Tasks. It basically lets us define regions of code that will be dispatched as tasks in the operating system. Thus, instead of merely looking for reduction patterns, what if we could look for coarser-grained code to be dispatched as tasks? Could we find potentially independent code and annotate it with OpenMP Task **pragmas**, and obtain a faster program? If we're able to find such hidden parallelism, how can we tell it is worth it to be dispatched as Tasks? After all, we know that parallelism may be expensive regardless the runtime, considering the cycles spent for context switching and dependence managing.

All these questions led me finally to the topic of this dissertation, that is automatically finding tasks in sequential C code. I exemplify the main motivation of this dissertation in Figure 15. In that figure, we have a function that computes the problem of the *N Queens*.

```

1 void nqueens(int n, int j, char *a, int *solutions, int depth)
2 {
3     int *csols;
4     int i;
5     if (n == j) {
6         /* good solution, count it */
7         *solutions = 1;
8         return;
9     }
10    *solutions = 0;
11    csols = alloca(n*sizeof(int));
12    memset(csols, 0, n*sizeof(int));
13    /* try each possible position for queen <j> */
14    for (i = 0; i < n; i++) {
15        /* allocate a temporary array and copy <a> into it */
16        char *b = alloca(n * sizeof(char));
17        memcpy(b, a, j * sizeof(char));
18        b[j] = (char) i;
19        if (ok(j + 1, b))
20            nqueens(n, j + 1, b, &csols[i], depth);
21    }
22    for (i = 0; i < n; i++)
23        *solutions += csols[i];
24 }

```

Figure 15. Code for the N Queens problem.

The *N Queens* problem derives from the *8 Queens Puzzle* and consists in placing n queens on an $n \times n$ chessboard in a way they're not aligned with each other either in line, column or diagonal, i.e., they're in a non-attacking position. There are solutions for every natural number but $n = 2$ and $n = 3$, as stated by Hoffman et. al [40]. Our function basically recursively computes all the possible positions for the N Queens and checks if each combination is valid. There is only one basic pruning technique, which consists in checking whether a position is valid to be placed with a queen or not, given a partial solution already computed. Other than that, it is a brute-force algorithm. So the question arises: can we parallelize it? Can we compute several combinations in parallel?

Well, if we only consider the problem in question, it's clear that we can compute the combinations in parallel. It is an inherent property of combinatorial algorithms the fact that many permutations are independent on each other. However, it won't be as easy for the compiler as it is for a human to decide which permutations can be computed in parallel and which ones cannot. Of course, this problem would greatly vary between different compilers, programming languages and implementations. We focus, however, on a raw implementation of the problem in C, using pointers as the main tool for computation. In this scenario, a C compiler might have trouble judging independence between iterations of the loop that computes the permutations.

As we are going to see, the problem of automatically finding opportunities for task parallelism is not trivial. In order to decide whether a piece of sequential code can be executed in parallel, we need to answer several questions: are the dependences stated by the compiler spurious, i.e., are they likely not to happen during execution? Can I parallelize it without compromising correctness? If it runs in parallel, can there be any data-races or other concurrency issues? Is the code coarse enough to pay off when considered the additional task runtime maintenance cost? Will parallelizing it really make a difference in the overall running time of the program in question? And, most important, how can we detect such type of irregular parallelism (task parallelism) statically, given that it usually only arises dynamically? Such questions will be scrutinized and properly addressed throughout the dissertation. Chapter 3 will define these and other questions thoroughly and Chapter 4 shall answer them.

We will, throughout this dissertation, discuss the problem of automatically detecting irregular parallelism in programs written in the C language. We will define the problem formally and then we will present an algorithmic solution implemented in a tool called **TaskMiner**. We defend the thesis that **TaskMiner** is a powerful tool capable of automatically finding task parallelism opportunities in code that was initially written using a sequential paradigm. The following sections will clarify the basis of parallelism in order to develop a solution for the aforementioned problem, as well as discuss the context of automatic parallelization of code in computer science nowadays.

1.2.1 Publications

This dissertation is the result of a two-year work on automatic parallelization of code by tasks. This work resulted in two papers with me as first author. One of them [72] was published in the proceedings of the Parallel Architectures and Compiler Techniques (PACT) Conference, certified by the Coordination for the Improvement of Higher Level Personnel (CAPES) as an A1 level conference. The other was published to the Brazilian

Symposium of Programming Languages (SBLP).

Although slightly out of scope, I have also published two other papers during my Masters. I participated in the development of a new pointer range analysis, which resulted in a paper [59] published to the proceedings of the Code Generation and Optimization (CGO) conference of 2017. In total, my Master’s studies in Computer Science yielded me 5 published papers.

1.3 Parallelism

In this section, we’ll briefly mention some concepts related to the area of parallel programming that are fundamental for the understanding of this dissertation in question. We’ll list the challenges of parallel programming, as well as the two main kinds of parallelism: data and task-based. We also discuss automatic parallelization, since it is the scope of the work in question, and then we move to the following chapters where we further explain our problem and solution.

1.3.1 Challenges in parallel programming

Since the computer industry switched to multicore and many-core architectures, parallel computing has been the primary option for developers when striving for more performance. Vectorization has become the main chosen path when parallelizing applications to run in GPU’s or CPU’s with multiple cores [42].

As we discussed in the previous section, parallel computing has been the focus of so many researches throughout Computer Science history to the point that programmers have been able to develop tools and techniques to extract an incredible level of parallelism abstraction from regular programs. Regular programs are those that commonly have high data-locality and iteration independence. They are usually programs that perform vector or dense matrix operations, stencils, graphical applications and others [51]. Still, coding in a parallel paradigm has always proven to be very challenging.

There are uncountable reasons behind the difficulties programmers often encounter when writing parallel code. The first and most well known is the unbalanced relation between memory access and arithmetic operations. On-chip resources have been growing faster than off-chip resources. On-chip has been following Moore’s law, while off-chip has relied solely in the evolution of the DRAM architecture, which has been falling behind each year due to physical and engineering limitations. The relation is, on average, 8 arithmetic instructions for each memory access instruction [42], and

this gap is expected only to increase. Therefore, designers have been keeping as much data possible on chip, extracting the most data locality and re-use as possible¹.

Another expressive obstacle when writing parallel programs revolves around the fact that, very often, the parallel version has higher complexity than the optimal and sequential version of the algorithm. Usually, designers have to rely heavily on large datasets to draw performance from parallel versions of algorithms. Parallelizing Delaunay Triangulation, an important algorithm in science and engineering, for example, had proven to be very hard when it comes to scalability [51; 67]. Computer Scientists have extensively devoted attention to parallelizing this fundamental algorithm. Unfortunately, creating unstructured meshes for vast point sets faces obstacles when working with large datasets. A recently proposed divide & conquer algorithm [64] has proven effective with large datasets, though. Also, the use of transactional memory has improved its scalability in some datasets [37; 19].

The main challenge concerning parallel programming remains being able to design a good parallel algorithm, though. The programmer must work out several data layouts, allocate memory and temporary storage, deal with pointer arithmetics, and sort out different kinds of data movement in order to draw the most of cache resources, allowing data re-use and locality. The skills of the programmer are still the biggest and most daunting challenge when it comes to designing parallel algorithms.

1.3.2 Types of parallelism

Yet, despite of all the challenges and obstacles, improving parallel programming techniques is still an incessant pursuit in the programming community. There are typically two sorts of parallelism: data parallelism and task parallelism [78; 24]. Data parallelism often arises in regular code, and as stated previously in this proposal, it revolves around dense matrix and vector operations, loops that are highly parallel, graphical operations and others that present great data-locality. Task-parallelism, on the other hand, arises in a different context, and is usually related to code with pointer-based structures such as graphs and trees.

1.3.2.1 Data parallelism

Data parallelism is the form of parallelism that arises in regular programs. Regular programs are code whose data-dependance behavior is easily determined statically,

¹That's also one of the reasons why data parallelism is easier to extract: once they are basically making use of high data locality, and the current CPU and GPU memory architectures are favoring data-locality each day, no surprise that data parallelism yields more rewardable practical results. See section 1.3.2.1.

making data parallelism easier to extract and to deal with. Loops that show iteration independence are called DOALL loops; so, naturally, data-parallelism is the dominant parallelism in DOALL loops ² [49].

Data parallelism also makes heavy use of data-locality and re-use and, as stated in Section 1.3.1, the current architectures favor this behavior. Therefore, instruction level data parallelism, specially the SIMD model (*Single Instruction, Multiple Data*) is the most explored kind of parallelism up to today. Nowadays we have many powerful tools based on integer linear programming for vectorizing and exploiting data parallelism in regular applications [51].

In data parallelism, the same computation is performed over different portions of a large dataset. That occurs with *stencil* operations, for example, in which a single computation must be applied upon a large amount of data and there is no dependence between these data, in a way that the operation can easily be done concomitantly.

To exemplify data parallelism, we come back to our first example: a *reduction*. The code in Figure 12 is a clear example of data parallelism. The same sum operation can be applied into different portions of the array in parallel. That means the same computation will be applied to different portions of the data. We can say that data parallelism arises from the distributed nature of data itself, i. e., when data has naturally no interdependence, it is highly distributable.

1.3.2.2 Task parallelism

On the other hand, task based parallelism arises in applications with structures rich in pointer operations, such as trees and graphs. These applications are deemed irregular because it's extremely hard to determine statically their data dependence due to the conservative trait of the compilers that often identify dependencies between pointers that are not going to necessarily occur during runtime.

Task parallelism can be found in different sorts of patterns, the most classic being the recursive one. On Figure 16 we can see one of those patterns. Each recursive call inside the method *mergesort* is independent, i.e., both can run in parallel. In contrast with instruction level parallelism, task based parallelism *coarser*, i. e., it is parallelism applied to a large set of instructions. For that reason, it is also called *coarsened-grained parallelism*, once it relies on the distributed nature of large chunks of independent code.

Loops that show heavy iteration dependence are called DOACROSS loops [49]³. Parallelizing a DOACROSS loop consists in finding task based parallelism, and it's

²A DOALL loop is also called a regular loop.

³A DOACROSS loop is also called an irregular loop.

```
1 void merge_sort(int* vec, int size, int* temp) {  
2     if (size == 1)  
3         return;  
4     merge_sort(vec, size/2, temp);  
5     merge_sort(vec + (size/2), size - (size/2), temp);  
6     merge(vec, size, temp);  
7 }
```

Figure 16. Mergesort is a classic example of task based parallelism: each recursive call is independent and can run in parallel.

proven to be a difficult task [58]. Researchers have been tackling this problem of finding parallelism in irregular loops for a long time [49; 25; 85]. Generally, we'll observe that the programs written in a Divide & Conquer paradigm are more likely to be strong candidates for task parallelism.

1.3.3 Automatic parallelization of code

Although parallelism can yield great performance gains, the burden of finding the opportunities to parallelize a program still lies on the programmer. The programmer must analyze the code, adapt the structures and methods to fit a parallel paradigm and deal with data movements and memory storage to extract the most of data reuse and locality. This task is certainly not trivial, and it's clearly error-prone [8]. Most of current models in parallel programming utilize *threads* which execute over a shared memory environment. However, it is notoriously difficult to write programs with multiple *threads* [71].

Given the difficulties in writing parallel programs and switching already sequentially written programs to a parallel paradigm, automatic parallelization of code is a very popular topic in computer science [42; 26; 61]. However, most of the works have focused on automatically finding data parallelism. Few has been done to automatically extract task based parallelism.

We developed an algorithm capable of finding task parallelism in irregular applications automatically. We present it as the tool **TaskMiner**. **TaskMiner** scavenges sequentially written C code for potential tasks, assess every aspect of each task such as concurrency, workload and data dependences, and then, finally, annotates the source code with **OpenMP** directives that surround these tasks. These directives will be compiled by a compiler that understands these **OpenMP**'s directives and will produce parallel code, improving performance of sequential programs without any human intervention.

Before we move on to discussing the problem in a more detailed perspective, we shall briefly discuss the OpenMP system and why it was our weapon of choice to point out task parallelism. The reader must understand the main benefits of using a runtime environment such as OpenMP's, for such advantages have guided most of the design decisions that led to the development of **TaskMiner**.

Chapter 2

The OpenMP System

The main contribution of this work is a suite of techniques to automatically scavenge task parallelism opportunities in C code. This task, however, involves several stages. For example, once we found such hidden parallelism in a given program, we would still need a tool to evidence it. It's not a contribution of this Dissertation an architecture or new runtime system to exploit task parallelism. Instead, we chose to rely on the OpenMP Runtime System. The OpenMP System is an annotation system which allows us to attach parallel semantics to a sequential fragment of code without changing the original syntax of the program. Since our focus is to identify tasks, the OpenMP System fits our needs for its practicality and simplicity. In this chapter, we will discuss the advantages involving the use of the OpenMP system.

2.1 Annotation

Annotation systems have risen to a place of prominence as a simple and effective means to write parallel programs. Examples of such systems include OpenMP [46], OpenACC [79], OpenHMPP [3], OpenMPC [55], OpenSs [60], Cilk++ [56] and OmpSs [17; 30]. Annotations work as a meta-language: they let developers grant parallel semantics to syntax originally written to execute sequentially. Combined with modern hardware accelerators such as GPUs and FPGAs, they have led to substantial performance gains [11; 62; 70]. Nevertheless, although convenient, the use of annotations is not straightforward, and still lacks supporting tools that help programmers to check if annotations are correct and/or effective.

As already stated previously in this dissertation, our tool of choice to exploit task parallelism in programs is OpenMP 4.5. We describe below which annotations we used, and explain, informally, their semantics. Full overviews of the syntax and semantics

of these annotations are publicly available¹; hence, we will not dive into their details. We will focus only on the list of directives and clauses that are effectively used by our TaskMiner compiler.

2.1.1 Directives

Directives work as **pragmas** inside the code. TaskMiner’s algorithm makes use of 4 main directives: **parallel**, **single**, **task** and **taskwait**. These directives work as an extension to the language, in a sense that they add new semantics to the language.

- **parallel** (*clauses*): This directive acts as a starting point for the parallel paradigm. It forms a team of threads which will execute the marked program region in parallel.
- **single** (*clauses*): This directive specifies that a program region must be executed by a single thread in the team. It only has meaning if it’s used inside the scope of a **parallel** defined region.
- **task** (*clauses*): Its scope points out code that should be dispatched as a Task.
- **taskwait**: It defines a synchronization point where threads must wait for the completion of child tasks.

2.1.2 Clauses

Some directives allow the use of clauses. Clauses work as parameters for the directives. For example, the **if** clause will define a condition for the execution of the associated directive.

- **default**([**shared**/**private**]): This clause indicates that variables in scope are either shared or replicated among tasks.
- **firstprivate**(*v*): It indicates that *v* must be replicated among tasks, and initialized with its value at the point where the annotation is executed. This clause will show to be very important when we solve the problem of concurrency between tasks in section 4.2.5.

¹For a quick overview, we refer the reader to the leaflet “Summary of OpenMP 4.0 C/C++ Syntax”, which is made available by the OpenMP group.

- **untied**: If a task has the **untied** modifier, then its associated code can be executed by more than one thread. That means threads might alternate execution due to preemption and load balancing, since the code is not tied to a single thread.
- **depend**(*in/out/inout*): It determines if data is read (*in*), written (*out*) or both within a task region. This clause effectively sets dependencies among tasks. Once we perform memory analysis to determine which ranges of memory each Task accesses, then this clause will be required if we want to keep the parallel version of the program correct.
- **final**(*condition*): This clause defines the conditions under which a task is allowed to create threads. We use it to enforce that only tasks that are above a certain workload threshold will be dispatched.

2.2 The runtime

The main advantage of using **OpenMP** annotations to split a program into parallel tasks is probably the ability of leaving the job of handling dependencies to a runtime. The **OpenMP** runtime maintains a task dependence graph which dynamically exposes more parallelism opportunities than those resulting from a conservative compilation-time analysis. In particular, the runtime lets us circumvent the shortcomings that pointer aliasing impose on the automatic parallelization of code. Aliasing, which is defined as the possibility of two pointers dereferencing to the same memory region, either prevents parallelism altogether, or forces compilers to resort to complex runtime checks to ensure its correctness [2; 73]. The **OpenMP** runtime shields us from this problem, because it already checks for dependencies among tasks during execution, and dispatches them in a correct order [53].

However, dependence tracking can be a bit complex. For example, should the runtime system treat dependencies across memory ranges, this tracking can involve checks of large ranges of addresses. It can also include memory labelling and renaming, if the runtime system is able to remove false dependencies. Regardless of its capacity, which we must state, is not the same across every **OpenMP** implementation, the runtime system will represent dependencies using a task dependence graph (TDG) [32]. In this directed acyclic graph, nodes denote tasks and edges represent dependences between them. Tasks are dispatched for execution according to a dynamic topological ordering of this graph [69].

OpenMP's runtime support allows the parallelization of irregular applications, such as programs that traverse data-structures formed by a mesh of pointers. In such

```

1 struct LINE {char* line; size_t size;}
2 int str_contains_pattern(char* str, char* pattern);
3 char* copy_str(char* str);
4 void book_filter(struct LINE** bk_in, char* pattern,
5     char** bk_out, int size) {
6     #pragma omp parallel
7     #pragma omp single
8     for (int i = 0; i < size; i++) {
9         #pragma omp task depend(in: line, pattern)
10        if (str_contains_pattern(bk_in[i]->line, pattern)) {
11            #pragma omp task depend(in: line)
12            bk_out[i] = copy_str(bk_in[i]->line);
13        }
14    }
15 }

```

Figure 21. The benefits of OpenMP’s annotations. Annotations appear in lines 6, 7, 9, 11.

programs, control constructs like *if* statements make the execution of some statements dependent on the program’s input. The runtime can capture such dependencies, in contrast to static analysis tools.

As an example, Figure 21 shows an application that finds patterns in lines of a book. The book is given as an array of pointers. Each pointer leads to a string representing a potentially different line. Our parallelization of this program consists in firing up a task to process each line. Figure 21 has been annotated by **TaskMiner**. The benefits of the OpenMP runtime environment can be seen in the picture. Neither the runtime checks of Alves [2] or Rus [77], nor Whaley’s context and flow sensitive alias analysis would be able to ensure the correctness of the automatic parallelization of this program. The OpenMP execution environment ensures the correct scheduling of the tasks created at lines 9 and 11, by reassuring that the annotated dependencies *line* and *pattern* are respected at runtime.

Chapter 3

The Problem

In this chapter we will examine, with more details, the problem we address in this dissertation.

3.1 Overview

As presented in Chapter 1, thus far, most of technologies concerning automatic parallelization of code through annotation systems explore data-parallelism. Such fact is unfortunate, since much of the power of these current annotation systems lies in their ability to create tasks [6]. The power to run different routines simultaneously on independent data brings annotation systems closer to irregular programs such as those that process graphs and worklists [67]. The essential purpose of this work was to address this omission.

This dissertation describes **TaskMiner**, a source-to-source compiler that exposes task parallelism in C/C++ programs. To fulfill this goal, **TaskMiner** solves different challenges. First, it determines symbolic bounds to the memory blocks accessed within programs (Section 4.2.1). Second, it finds program regions, e.g., loops or functions, that can be effectively mapped onto tasks (Section 4.2.2). Third, it extracts parameters from the code to estimate when it is profitable to create tasks. These parameters feed conditional checks, which, at runtime, enable or disable the creation of tasks (Section 4.2.3) and limit their recursion depth (Section 4.2.6). Fourth, **TaskMiner** determines which program variables need to be privatized in new tasks, or shared among them (Section 4.2.5). Finally, it maps all this information back into source code, producing readable annotations (Section 4.2.6).

We defend the allegation that automatic task annotations are effective and useful. Our techniques enhance the productivity of developers, because they save them the

time to annotate programs. In Chapter 5, we evaluate our tool and show that we have been able to achieve almost four-fold speedups onto standard benchmarks (not-used in parallel programming), with zero programming cost. **TaskMiner** receives as input C code, and produces, as output, a C program annotated with human-readable OpenMP task directives. We are currently able to annotate non-trivial programs, involving every sort of composite type available in the C language, e.g., arrays, structs, unions and pointers to these aggregates. Some of these programs, such as those taken from the Kastor [88], Bots [33] or the LLVM test suites, are large and complex. Yet, our automatically annotated programs not only approximate the execution times of the parallel versions of those benchmarks, but are in general faster than their sequential — unannotated — versions, as we report in the evaluation (Chapter 5).

3.2 Memory dependencies

One of the greatest challenges in annotating C code with Task **pragmas** is probably the fact that, in C, we’re always dealing with pointers. And as the reader might know, pointers don’t keep track of where they went or where they go; they simply point to a range of addresses during the program’s execution. Should we want to parallelize a sequential code filled with pointers operations (arrays, structs, matrices), we ought to be certain of all possible addresses that can be pointed to by every pointer inside the parallel region. Otherwise, we could end up with a parallel version of the program that is incorrect, because it ignores or disrespects certain memory dependencies that are necessary for the program to execute correctly. That type of analysis has a name: pointer range analysis, or memory range analysis. It determines the entire range of addresses that a given pointer might point to during execution.

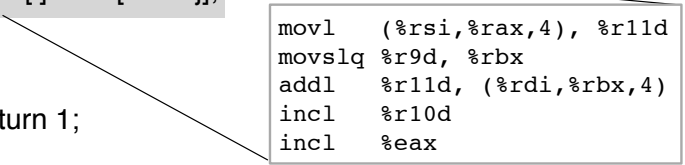
As seen in Chapter 2, the OpenMP runtime liberates us from the burden of having to track dependencies between pointers statically. However, we still need to make sure we annotate the program correctly. That means for each Task we need to determine, with precision, the ranges of addresses that are written, read or both. This is indispensable if we want to obtain an annotated version that maintains correctness.

See Figure 31, for example. The code in it illustrates our first challenge when trying to annotate a program with Tasks directives. That program receives an $M \times N$ matrix V , in linearized format, and produces a vector U , so that $U[i]$ contains the sum of all the elements in line i of matrix V . For reasons to be considered later in Section 4.2.1, our static analysis determines that each iteration of the outermost loops could be made into a task. So let’s say we ran our algorithm (to be described later) and now we want

```

1  int foo(int* U, int* V, int N, int M) {
2      int i, j;
3      #pragma omp parallel
4      #pragma omp single
5      for(i = 0; i < N; i++) {
6          #pragma omp task depend(in: V[i*N:i*N+M]) \
7                                     if (5 * M < WORK_CUTOFF)
8          for (j = 0; j < M; j++) {
9              U[i] += V[i*N + j];
10         }
11     }
12     return 1;
13 }

```



```

movl    (%rsi,%rax,4), %r11d
movslq  %r9d, %rbx
addl    %r11d, (%rdi,%rbx,4)
incl    %r10d
incl    %eax

```

Figure 31. Identifying memory regions with symbolic limits, and using runtime information to estimate the profit of tasks.

to mark the innermost loop as a task.

Thus, tasks will comprise of the innermost loop, and will traverse the memory region between addresses $\&V + i * N$ and $\&V + i * N + M$. The identification of such ranges involves the use of a symbolic algebra, which we have borrowed from the compiler-related literature, as we explain in Section 4.2.1.

If we are to annotate the program correctly, we will place the directive `task` accompanied by clauses `depend` above the innermost loop, stating the input and output dependencies of this task. That means the regions in memory that this task writes to or reads from. Figure 32 isolates the annotation on Figure 31.

```

1 #pragma omp task depend(in: V[i*N : i*N + M])

```

Figure 32. Task annotation with input memory dependencies.

TaskMiner’s algorithm runs a symbolic range analysis to find precise memory ranges, as described further in Section 4.2.1.

Figure 31 also introduces the second challenge that we tackle, which will be discussed in the next section: how do we statically estimate a Task’s profitability?

3.3 Profitability of Tasks

The creation of tasks involves a heavy runtime cost due to allocation, scheduling and real-time management of the dependence graph, as described in Section 2.2. Ideally, this cost should be paid only for tasks that perform an amount of work sufficiently large to pay for their management. Being an interesting program property, on Rice’s sense [74], the amount of work performed by a task cannot be discovered statically. As we show in Section 4.2.3, we can try to approximate this quantity, using, to this end, program symbols, which are replaced with actual values at runtime.

For instance, in Figure 31, we know that the body of the innermost loop is formed by five instructions. Thus, we approximate the amount of work performed by a task with the expression $5 * M$. We use the runtime value of M to determine, during the execution of the program, if we create a task or not. Such test is carried out by the guard at line 7 of the figure, which is part of OpenMP’s syntax. Also, we provide a reliable estimate on the *workload cutoff* from which a task can be safely spawned without producing performance overhead. This *cutoff* considers factors such as number of available cores and runtime information on the task dispatch cost in terms of machine instructions. This is further explained in Section 4.2.3.

3.4 Bounding the number of Tasks

We shall introduce the next major challenge faced by us when mining for tasks automatically by quoting Duran *et al.*: “*In task parallel languages, an important factor for achieving a good performance is the use of a cut-off technique to reduce the number of tasks created*” [31]. This observation is particularly true in the context of recursive, fine-grained tasks, as we analyze later in Section 4.2.6. Bottom line, we don’t want too many small tasks in the pool, but we don’t want a few too large tasks either. We should strive for the maximum number of tasks that exceed a minimum workload threshold, otherwise we won’t have gains when running the program in parallel. In fact, if we have too many small tasks we might end up with a slower program, given that the runtime has a cost related to the maintenance of the Task Dependence Graph.

Figure 33 provides an example to this issue, which often appears in recursive patterns. As stated previously in this dissertation, some Divide & Conquer recursive programs are excellent candidates for automatic parallelization by tasks. However, it might be common to a recursive program to have few coarse-grained recursive calls at the start of execution and multiple extremely fine-grained calls at the end of the recursion tree. In Figure 33 we see a classic brute-force *Fibonacci* algorithm. We

```

1  static int taskminer_depth_cutoff;
2  long long fib(int n) {
3      taskminer_depth_cutoff++;
4      long long x, y;
5      if (n < 2) return n;
6      #pragma omp task untied default(shared) \
7          if(taskminer_depth_cutoff < DEPTH_CUTOFF)
8      x = fib(n - 1);
9      #pragma omp task untied default(shared) \
10         if(taskminer_depth_cutoff < DEPTH_CUTOFF)
11      y = fib(n - 2);
12      #pragma omp taskwait
13      taskminer_depth_cutoff--;
14      return x + y;
15  }

```

Figure 33. Bounding the creation of recursive tasks. Example taken from [44, Fig.1].

understand that this is clearly not the optimal implementation for the problem, but our focus should be on the granularity of the tasks within the execution of the program. If we determine each recursive call as a Task, we might end up with too many active tasks in the pool after a few recursive hops.

We deal with this problem in two ways: first, we assess the Task’s profitability, as stated in Section 3.3 and explained in Section 4.2.3. Second, to place a limit on the number of recursive tasks simultaneously in flight, we simply associate the invocation of recursive functions annotated with task pragmas with a counter — `taskminer_depth_cutoff` in Figure 33. The guard in line 7 ensures that we never exceed `DEPTH_CUTOFF`, a predetermined threshold. This example, together with Figure 31, allows us to emphasize that the code generation algorithms presented in this work are parameterized by constants such as `DEPTH_CUTOFF`, or `WORK_CUTOFF` in Figure 31. Although these have default estimates, we provide them as parameters to be set at will.

3.5 Concurrency of Tasks

The two previous sections describe problems related to the performance of annotated programs: if left unsolved, we shall have correct, although inefficient programs. Section 3.2 and Section 3.5, in turn, describe problems related to correctness. If we don’t solve

```

1 void sum_range(int* V, int N, int L, int* A) {
2     int i = 0;
3     #pragma omp parallel
4     #pragma omp single
5     while (i < N) {
6         int j = V[i];
7         A[i] = 0;
8         #pragma omp task default(shared) firstprivate(i, j)
9         for (; j < L; j++) { A[i] += V[j]; }
10        i++;
11    }
12 }

```

Figure 34. Variable j must be replicated among tasks, to avoid the occurrence of data races.

them, we end up with a parallel version of the program that is incorrect.

When determining a sequentially written code region as a Task to be executed in parallel, the maintenance of correctness asks for the identification of variables that must be replicated among threads. This process of replication is called *privatization*. As an example, variable j in Figure 34 must be privatized. In the absence of such action, that variable will be shared among all the tasks created at line 5 of Figure 34, and then we will face a data-concurrency issue. Every task will read and write at j . If that happens, the program will clearly be incorrect, since there should be a j for every Task instead of only one shared between all. Specifically in this example, because j is written to within these tasks, race conditions would ensue. Section 4.2.5 explains how we distinguish private from shared variables.

Next Chapter will finally address TaskMiner’s algorithm as well as our solution to all of the problems listed in Chapter 3.

Chapter 4

The TaskMiner

In this Chapter, we finally address the TaskMiner algorithm in detail. This algorithm mixes information from different compiler structures such as the Control Flow Graph (CFG) and the Program Dependence Graph (PDG) to statically infer task parallelism opportunities from code written in C. Before we go into the algorithm, we present these and other concepts that are fundamental for the understanding of TaskMiner’s mechanics. We do so in the order we believe to be the clearest for the algorithm’s construction.

4.1 Definitions

If we are to state the main purpose of TaskMiner, it surely is to identify Tasks. So the main goal of our work is to identify Tasks in *structured* programs. In the context of this work, a structured program is a program that can be partitioned in *hammock* regions, a concept introduced by Ferrante *et al.* [36] in the middle 1980’s. For completeness purposes, we re-state what are *hammock* regions in Definition 4.1.1.

Definition 4.1.1 (Hammock Region [36]) *A Control Flow Graph G is a directed graph with a starting node s , and an exit node x . A hammock region G' is a subgraph of G with a node h that dominates¹ the other nodes in G' . Additionally, there exists a node $w \in G$, $w \notin G'$, such that w post-dominates every node in G' . In this definition, h is the entry point, and w is the exit point of G' .*

Figure 41 illustrates the notion of a *hammock* region. This concept is fundamental to the concept of a Task, to be formalized in definition 4.1.2.

¹A node $n_1 \in G$ dominates another node $n_2 \in G$ if every path from S to n_2 goes across n_1 . Inversely, n_1 post-dominates n_2 if every path from n_2 to x must go across n_1 .

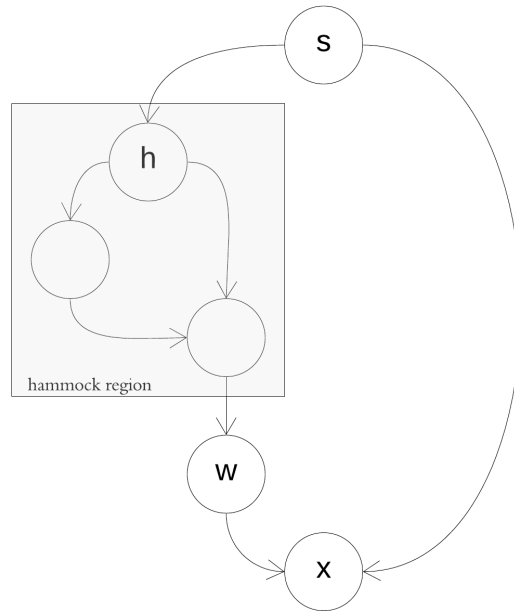


Figure 41. Hammock region.

Definition 4.1.2 (Task) *Given a program P , a task T is a tuple (G', M_i, M_o) formed by a hammock region G' , plus a set M_i of memory regions representing data that T reads, and a set M_o of memory regions representing data that T writes.*

Definition 4.1.2 uses the concept of *memory region*. Syntactically, memory regions are described by program variables and/or pointers plus ranges of dereferenceable off-sets. Given two tasks: $T_1 = (G_1, M_{i1}, M_{o1})$ and $T_2 = (G_2, M_{i2}, M_{o2})$, if $M_{o1} \cap M_{i2} \neq \emptyset$, then we say that T_2 *depends* on T_1 . If a program P is partitioned into a set of n tasks, then this partition is said to be *correct* if it abides by Definition 4.1.3.

Definition 4.1.3 (Correctness) *A set T of n tasks is a correct parallelization of a program P if:*

- *T does not contain cyclic dependence relations;*
- *the execution of the tasks in T in any ordering determined by dependence relations leads to the same results as the sequential execution of P .*

Example 4.1.1 (Memory Regions) *Below we have two tasks, $T_{foo} = (\text{foo}, \{v[i-1]\}, \{v[i]\})$ and $T_{bar} = (\text{bar}, \{v[i]\}, \{\})$. Dependencies are identified*

via the *depend* clause. In Figure 42, each hammock region is formed by one function call.

```

1 #pragma omp task depend(in: V[i-1]) depend(out: V[i])
2 V[i] = foo(&V[i-1], i);
3 #pragma omp task depend(in: V[i])
4 bar(&V[i], i);

```

Figure 42. Memory regions annotated as *depend* clauses.

The TaskMiner algorithm will use these and other definitions to find Tasks in structured programs.

4.2 The algorithm

Figure 43 provides a high-level view of a source-to-source compiler that incorporates the techniques discussed in this dissertation.

```

1 void TaskMiner(Program, CostModel)
2 {
3   CFG = ControlFlowGraph(Program);
4   PDG = ProgramDependenceGraph(Program);
5   Ranges = SymbolicRangeAnalysis(CFG);
6   Vanes = FindVanes(PDG);
7   Tasks = []
8   for (v in Vanes)
9   {
10    Region = v.Expand();
11    Tasks.append(Region);
12  }
13  Privs = PrivatizationAnalysis(Tasks, PDG);
14  Costs = ProfitabilityAnalysis(Tasks, Ranges, CostModel);
15  Program.Annotate(Tasks, Costs, Ranges, Privs);
16 }

```

Figure 43. High level abstraction of the TaskMiner algorithm.

The many parts of this algorithm will be explained in this section. This pseudo-code uses several concepts well-known in the compilers literature, such as control flow graphs [50] and dependence graphs [36]. In the rest of this section we explain how we

have combined this previous knowledge to delimit and annotate tasks in structured programs. Our presentation focuses on the new elements that we had to add onto known techniques, in order to adapt them to our purposes.

In a simplified way, the TaskMiner algorithm starts by generating two important structures in compilers: the Control Flow Graph [50] and the Program Dependence Graph [36].

Definition 4.2.1 (Program Dependence Graph) *A Program Dependence Graph (PDG) in computer science is a representation, using graph notation that makes data dependencies and control dependencies explicit. These dependencies are used during dependence analysis in optimizing compilers to make transformations so that multiple cores are used, and parallelism is improved. Figure 44 illustrates a PDG for a given program.*

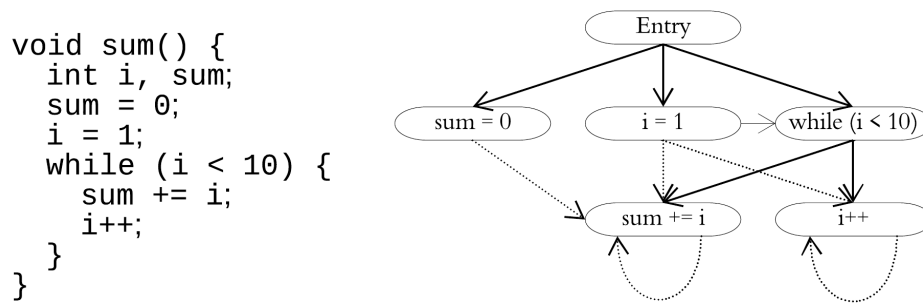


Figure 44. Program Dependence Graph for a given program. Solid edges represent data dependencies and dashed edges represent control dependencies.

Definition 4.2.2 (Control Flow Graph) *In a control flow graph each node in the graph represents a basic block, i.e. a straight-line piece of code without any jumps or jump targets; jump targets start a block, and jumps end a block. Directed edges are used to represent jumps in the control flow. There are, in most presentations, two specially designated blocks: the entry block, through which control enters into the flow graph, and the exit block, through which all control flow leaves. Figure 45 illustrates a CFG for a given program.*

Moving on, TaskMiner performs a symbolic range analysis to determine the Memory Regions accessed by every pointer inside the region being analyzed by the algorithm.

```

void sum() {
    int i, sum;
    sum = 0;
    i = 1;
    while (i < 10) {
        sum += i;
        i++;
    }
}

```

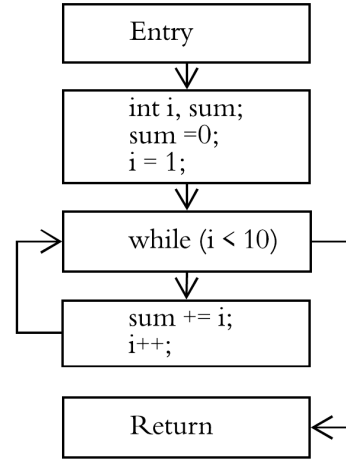


Figure 45. Control Flow Graph for a given program. Instructions are logically organized in basic blocks.

This process will be further explained in Section 4.2.1. Then, **TaskMiner** scavenges the PDG looking for structures called **Vanes**. This structure is the key insight behind finding **Tasks** statically and shall be introduced in Section 4.2.2. These **vanes** will, eventually, become **Tasks** if they fit some requirements. First, we shall perform a profitability analysis to check if a **Vane** can become a **Task** without loss of performance. This is explained with details in Section 4.2.3. Then, for each found **Vane**, we execute an expansion algorithm, to be explained in Section 4.2.4. This algorithm will try to expand **Task** regions to encompass larger regions in order to fit the **CostModel**. However, this expansion has limits, once it cannot reduce the parallelism that would be obtained should that **Vane** be identified as a **Task**. Later on, with the **Tasks** in hand, the algorithm will execute a privatization analysis to find out which values are to be shared between **Tasks** and which shall be privatized and replicated among **Tasks**. We visit this procedure with details in Section 4.2.5. Finally, it maps it all back into source code again in form of OpenMP **pragmas** directives. Last but not least, this step might be the trickiest for it has to deal with scope issues and other engineering details when mapping back information from an intermediate representation of the program to source-code. We explain how this mapping is done in Section 4.2.6.

4.2.1 Memory range analysis

To produce the annotations that create a task $T = (G, M_i, M_o)$, we need to determine the memory regions M_i that T reads, and the memory regions M_o that it writes. Memory regions consist of pointers, plus ranges of addresses that these pointers can

dereference. Example 4.2.1 illustrates this notion. To determine precise bounds for memory regions, we resort to an old ally of compiler writers: the *Symbolic Range Analysis*, a concept that Definition 4.2.3 formalizes.

Example 4.2.1 (Memory Region) *The statement $U[i] += V[i*N + j]$, at line 9 of Figure 31 contains two memory accesses: $U[i]$ and $V[i*N + j]$. The first covers the memory region $[\&U, \&U + (M - 1) \times \text{sizeof}(\text{int})]$. The other covers the region $[\&V + N \times \text{sizeof}(\text{int}), \&V + ((i \times M + j) - 1) \times \text{sizeof}(\text{int})]$. There is no code in Figure 46 that depends on the loop in lines 8-10. Thus, a task annotation must only account for the input dependence, e.g., the access on V . That is why the **depend** clause in line 6 contains a reference to this region.*

```

1  int foo(int* U, int* V, int N, int M) {
2    int i, j;
3    #pragma omp parallel
4    #pragma omp single
5    for(i = 0; i < N; i++) {
6      #pragma omp task depend(in: V[i*N:i*N+M]) \
7        if (5 * M < WORK_CUTOFF)
8      for (j = 0; j < M; j++) {
9        U[i] += V[i*N + j];
10     }
11   }
12   return 1;
13 }

```

```

movl  (%rsi,%rax,4), %r11d
movslq %r9d, %rbx
addl  %r11d, (%rdi,%rbx,4)
incl  %r10d
incl  %eax

```

Figure 46. Memory regions as symbolic limits.

Definition 4.2.3 (Symbolic Range Analysis) *This is a form of abstract interpretation that associates an integer variable v with a Symbolic Interval $R(v) = [l, u]$, where l and u are Symbolic Expressions. A symbolic expression E is defined by the grammar below, where s is a program symbol:*

$$\begin{aligned}
 E ::= & \ z \mid s \mid E + E \mid E \times E \mid \min(E, E) \mid \\
 & \max(E, E) \mid -\infty \mid +\infty
 \end{aligned}$$

The program symbols mentioned in Definition 4.2.3 are names that cannot be reconstructed as functions of other names. Examples include global variables, function arguments, and values returned by external functions. There are several implementations of symbolic range analysis available. We have adopted the one in the DawnCC compiler [62], which, itself, reuses work from Blume *et al.* [12]. The only extension

that we have added into DawnCC’s implementation was the ability to handle C-like **structs**. Therefore, we shall not provide further details about this part of our work. To understand the rest of this dissertation, it suffices to know that this implementation is sufficiently solid to handle the entire C99 language, always terminates, and runs in time linear on the size of the program’s dependence graph. Thus, in the worst case, it is quadratic on the number of program variables. If a program contains an array access $V[E]$, and we have that $R(E) = [l, u]$, then the memory region covered by this access is, at least, $[\&V + l, \&V + u]$.

To clarify this observation, let us take this example: Symbolic range analysis, when applied to Figure 46, give us $R(i) = [0, N - 1]$, and $R(j) = [0, M - 1]$. The memory access $V[i*N + j]$ at line 9, when combined with this information, yields the symbolic region that appears in the task annotation at line 6 of Figure 46.

After the memory range analysis is done, TaskMiner moves on to the critical section of its algorithm: finding potential Tasks.

4.2.2 Mapping code regions to tasks

A *task candidate* is a set of program statements that can run in parallel with the rest of the program. To identify task candidates, we rely on the program’s *Dependence Graph*. We have stated the definition for a Program Dependence Graph (PDG) in Section 4.2, but we shall redefine it again in this Section in a less verbose and more symbolic way. Definition 4.2.4 gives us a Program Dependence Graph definition with a symbolic approach.

Definition 4.2.4 (Program Dependence Graph [36]) *Given a program P , its PDG contains one vertex for each statement $s \in P$. There exists an edge from s_1 to s_2 if the latter depends on the former.*

Statement s_2 is data-dependent on s_1 if it reads data that s_1 writes.

It is control dependent on s_1 if s_1 determines the outcome of a branch, and depending on this outcome, we might execute s_2 or not.

Task candidates are vanes in windmills. *Windmills* are a family of graphs. To the best of our knowledge, the term was coined by Rideau *et al.* [75] to describe structural relations between register copies. Our windmills exist as subgraphs of the program’s dependence graph. We adopt a slightly more general definition than Rideaus’; however, the metaphor that gave origin to the name: the shape of the graphs, is still unmistakable:

Definition 4.2.5 (Windmill [75]) A windmill is a graph $G_w = G_c \cup G_{v1} \cup \dots \cup G_{vn}$ formed by a strongly connected component G_c (its center), plus n components, not necessarily strong, G_{vi} , the vanes, such that:

- for any topological ordering of G_w , and nodes $n_c \in G_c$, $n_v \in G_{vi}$, we have n_c ahead of n_v ;
- for any i and j , $1 \leq i < j \leq n$, G_{vi} and G_{vj} do not share vertices (thus, sharing edges is also impossible).

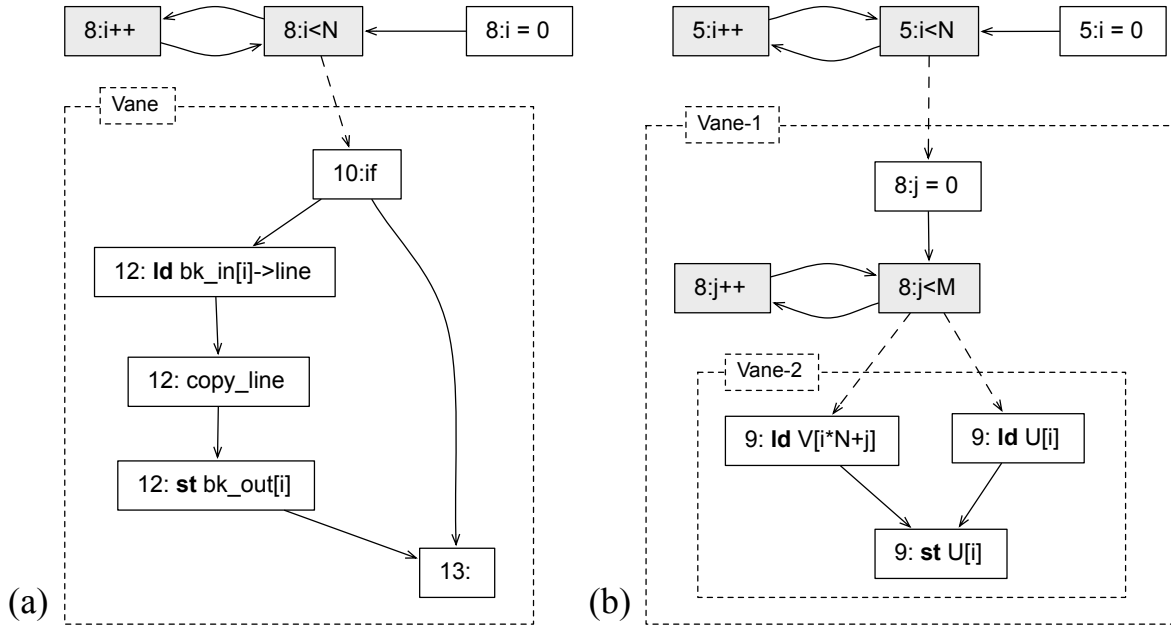


Figure 47. Examples of windmills and vanes in program dependence graphs.

Figure 47 shows two program dependence graphs, highlighting windmills and their vanes. Nodes that form centers of windmills are colored in gray. Data-dependence edges are solid; control-dependence edges are dashed. Vanes of windmills appear within dotted boxes. Figure 47 (a) shows the PDG for Figure 21; and (b) shows the PDG for Figure 31.

The structure of windmills naturally leads to task candidates. Vanes correspond to program parts that are likely to be executed several times, because they sprout from a loop – the center of the windmill. Two different executions of the same vane can run in parallel. This is a property of the PDG, and one of the original motivations behind its design.

These two different executions can be seen as two instances of the same structure (the vane), coming out of the same windmill center. A topological ordering of the dependence graph does not impose any order among these two hypothetical replicas of the same structure.

Windmills give us structural properties to identify tasks. We can find them via a depth-first search traversal of the program's dependence graph. Function `FindVanes()`, in Figure 43 implements this search. However, not every windmill leads us to profitable tasks. Furthermore, some windmills cannot be annotated, due to the inability of range analysis to find bounds to the memory they access. We deal with these shortcomings in the next two sections.

4.2.3 Estimating the profitability of tasks

The benefit of creating tasks depends on two factors: runtime cost, and parallelism. This tension leads to a sweet spot in the size of tasks, where size is measured as the amount of code they execute. The smaller the tasks are, the more parallel they tend to be, as a consequence of less dependencies.

However, if tasks are too small, then the performance gained by the extra parallelism might not compensate the cost of managing them. On the other hand, if tasks are too big, then we might not obtain enough parallelism, because individual tasks still execute sequentially. `TaskMiner` marks as tasks *the first maximal analyzable vane above the runtime cost*.

To imbue this last statement with meaning, in what follows, we define more formally the notions of task size. We leave the concept of maximal analyzable vane for Section 4.2.4.

4.2.3.1 Estimating the Size of Tasks

The size of a task is given by its workload, i.e., the number of instructions that such task executes. The same program region might lead to the creation of tasks with different sizes. Therefore, the actual size of a task is only known after it finishes execution. However, to judge if the creation of a task is worth its cost, we must be able to approximate its size statically. With this goal, we define the notion of *Static Workload Estimate*:

Definition 4.2.6 (Static Workload Estimate (SWE)) *Let $G = G_1 \cup G_2 \cup \dots \cup G_n$ be a partition of a program's control flow graph G into n disjoint hammock regions. We*

define the static workload estimate ($W(G)$) as a non-negative real number, such that $W(G) = W(G_1) + W(G_2) + \dots + W(G_n)$.

Definition 4.2.6 is too unconstrained: a function $W(G) = 0$ for any G satisfies it. However, we want W to approximate the dynamic behavior of programs. The literature contains heuristics to implement W . Perhaps, the best-known among these heuristics is the static profiler of Wu and Larus [90].

We adopted a different approach: we reuse the symbolic range analysis of Section 4.2.1 to augment static data with dynamic information. In other words, we use symbolic range analysis to construct expressions that represent the number of iterations of loops. A similar approach has already been used to decide when to migrate virtual memory pages in NUMA architectures [66]. OpenMP 4.5 contains syntax to enable the conditional creation of tasks – see the clauses listed in Chapter 2. Our symbolic analysis lets us build predicates for such conditionals.

The condition in line 7 of Fig. 46 determines that tasks are created if $5 \times M < \text{WORK_CUTOFF}$. M is a program symbol, i.e., a variable passed as argument to function `foo`. The expression $5 \times M$ is an approximation for the size of the task that comprises the loop at lines 8-10. The constant `WORK_CUTOFF` represents the cost of creating a thread in the OpenMP runtime, and it is determined empirically.

[INSTR]	$W(\text{instr}) = \text{given by the spec.}$
[SEQ]	$\frac{W(S_1) = w_1 \quad W(S_2) = w_2}{W(S_1; S_2;) = w_1 + w_2}$
[BRANCH]	$\frac{W(S_1) = w_1 \quad W(S_2) = w_2 \quad W(S_3) = w_3}{W(\text{if}(S_1) S_2; \text{else } S_3;) = w_1 + \max(w_2, w_3)}$
[LOOPINF]	$\frac{W(S_1) = w_1 \quad W(S_2) = w_2 \quad \text{Iter}(S_1) = \infty}{W(\text{while}(S_1) S_2;) = 10 \times (w_1 + w_2)}$
[LOOPEXP]	$\frac{W(S_1) = w_1 \quad W(S_2) = w_2 \quad \text{Iter}(S_1) = E}{W(\text{while}(S_1) S_2;) = E \times (w_1 + w_2)}$

Figure 48. Estimating the cost of tasks.

The declarative rules in Figure 48 sketch the heuristics that we use to compute the SWE for a hammock region S . Rule `LOOPINF` applies on loops whose number of iterations we cannot bound via some symbolic expression computed statically. We

assume that loops execute 10 times, following previous statistics [90]. Rule `LOOPEXP` represents loops that we can analyze using our Symbolic Range Analysis. The auxiliary function $Iter(S)$ returns a symbolic expression that represents the range of values covered by the code S that controls the number of iterations of the loop. In Figure 43, these rules are implemented by function `profitabilityAnalysis`.

4.2.3.2 The Cost Model

A static workload estimate is only meaningful in the context of a *cost model*. A cost model is a collection of parameters that determine the impact of the underlying computer architecture onto the creation and management of tasks. The literature contains examples of techniques used to build cost models, be it analytically [7], or empirically [70].

The automatic construction of a cost model is not the goal of `TaskMiner`. Instead, the reader shall notice that Figure 43 receives a cost model as a parameter. For the experiments carried out in this dissertation, to be presented in Chapter 5, we have determined a small collection of constants related to the creation of threads in our target architecture. As an example, the value of `WORK_CUTOFF`, seen in Example 4.2.1, in our setting, is roughly 500 cycles.

The next step of the algorithm is the `Task` expansion.

4.2.4 Task expansion

As we stated previously in this dissertation, `TaskMiner` treads on the tenuous limit between the number of tasks (high parallelism) and the size of tasks (profitable tasks). If we annotate a small region as a `Task`, it might not pay off its cost in the runtime, leading to a slower program. On the other hand, if we expand the small tasks into larger ones, we might end up with less tasks, less parallelism. After all, the largest possible `Task` is the entire program. If we found the entire program as a `Task`, then there's no parallelism whatsoever.

The method of expanding small tasks into larger ones, bearing external factors to find the best size for a given `Task` is called `Task Expansion`. The process is iterative, which means that a certain `Task` will be expanded until a certain condition is met.

Thus, `Task` expansion consists in finding the smallest task that is large enough to pay for the thread creation cost. As we saw in Chapter 2, the maintenance of the `Task Dependence Graph` in the Runtime has a cost. Hence, we don't want to dispatch `Tasks` that are too small. Figure 49 shows the algorithm that performs this activity. We start

```

1 void Expand(Vane, COST)
2 {
3   CanExpand = True;
4   while (CanExpand and SWE(Vane) < COST)
5   {
6     ParentRegion = Vane.GetParent();
7     Vane.BasicBlocks.Add(ParentRegion);
8     Vane.ResolveDependencies();
9     CanExpand = CheckExpansion(Vane);
10  }
11 }

```

Figure 49. Task discovery via expansion of hammock regions. **COST** is the overhead of creating and scheduling threads.

this algorithm by setting **REGION** to be a hammock graph H that corresponds to a vane within a windmill W .

The hammock decomposition of a structured program forms a tree [36], such that vertices in this tree represent hammock graphs. H_1 is a child of H_2 in this tree if two conditions apply: (i) $H_1 \subset H_2$, and (ii) for any other node H_x , if $H_1 \subset H_x$ then $H_2 \subset H_x$. In this context, we call H_2 the *parent* of H_1 .

The routine **CheckExpansion()** sets the boolean **CanExpand** in Figure 49. This function is true as long as the following conditions apply:

- (i) **VANE** has the structural properties of a windmill's vane, i.e., it is contained within a windmill W ;
- (ii) we can analyze the memory regions within **VANE** using the symbolic range analysis of Section 4.2.1;
- (iii) **VANE** does not depend on another vane inside the same windmill W .

Figure 410 illustrates Task Expansion. In (a) we can see an example of doubly nested loop that shall be analyzed by the **TaskMiner**; in (b), the dependence graph of the program; and in (c) the decomposition of this loop into windmills and vanes. D is the set of variables the vane depends on.

The program in Figure 410 (a), a slight variation of the code seen in Figure 31, contains two windmills. The first consists of the **for** loop at line 1. The second, nested in the first, is the **for** loop at line 2. The vanes that sprout from these windmills are highlighted in the program dependence graph in Figure 410 (b). The figure shows that we have three task candidates, each formed by a different vane. To find the actual tasks, we apply function **expand** (Fig. 49) onto the innermost vanes: **Vane-2** and **Vane-3**. The

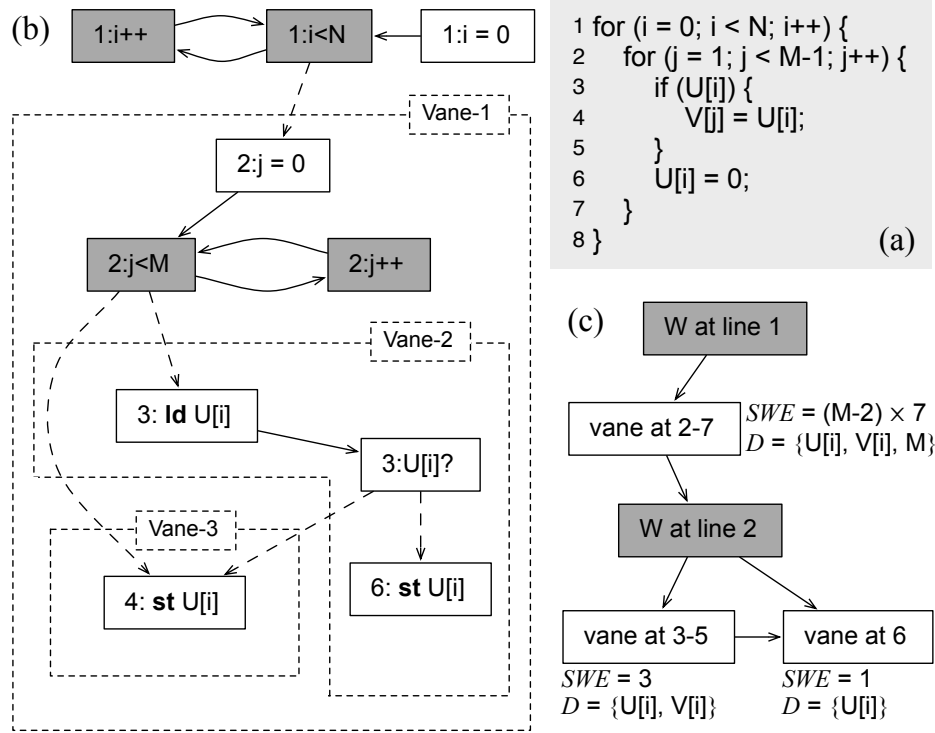


Figure 410. Example of Task Expansion.

latter cannot be expanded, as it depends on the former. On the other hand, **Vane-2** can be expanded to include **Vane-3**, as this expansion does not create new dependences between vanes in the same windmill.

We expand tasks up to the **COST** threshold, which is determined by the cost model seen in Section 4.2.3. If the profitability of a task is smaller than this constant, then parallelism might not pay off the cost of managing threads. However, if we expand tasks too much, then we might lose parallelism, as code in a task runs sequentially. Thus, determining **COST** is essential to the performance of our optimization. In this work, we have set this parameter empirically for the runtime environment to be described in Chapter 5.

In the next Section, we shall see how **TaskMiner** determines which values should be shared and which should be replicated among tasks, to work around race conditions.

4.2.5 Privatization analysis

TaskMiner bestows a parallel semantics on code that has been originally conceived to run sequentially. This semantic gap might lead to race conditions. Race conditions happen when annotations cause two or more threads to update values that, in the

sequential program, are denoted by the same name. To avoid such situations, the insertion of annotations requires us to replicate some values, making them private to each task. We have named such replication *privatization*.

```

1 void sum_range(int* V, int N, int L, int* A) {
2     int i = 0;
3     #pragma omp parallel
4     #pragma omp single
5     while (i < N) {
6         int j = V[i];
7         A[i] = 0;
8         #pragma omp task default(shared) firstprivate(i, j)
9         for (; j < L; j++) { A[i] += V[j]; }
10        i++;
11    }
12 }
```

Figure 411. Variables j and i are replicated among tasks, avoiding data races.

Variables i and j in Figure 411 need to be replicated among the tasks that the annotation at line 8 creates. The need to replicate j is more apparent: this variable is updated in the body of the task pragma, at line 9. In the absence of replication, we have a race condition. Variable i must also be replicated. Each iteration of the loop at line 9 reads a different value of i . Thus, each task should receive a different value in this variable. In the absence of replication, all the threads would read the same value; hence, parallelization would change the semantics of the program.

Definition 4.2.7 (The Private Requirement) *If a variable name has scalar type and contains at least one def-use chain that enters the frontier of a task, then this name is said to bear the private requirement. The frontier of a task is determined by the `expand` function seen in Figure 49.*

Variable i must be privatized because it is defined at lines 2 and 10 of Figure 411, and is used at line 9 – the body of a new task. Moreover, this variable has a scalar type, e.g., `int`. Similarly, variable j is defined at line 6, and is used at line 9. This property — the existence of def-use chains entering the task region — leads to the insertion of the `firstprivate` clause at line 8 of Figure 411.

According to Definition 4.2.7, only names representing scalar values are privatized. These names have *value semantics*, i.e., they are “passed by copy” if used as

actual arguments of functions. Memory regions, that is to say, regions whose access happens through pointers, are not privatized. These regions are, instead, marked as dependences of the task, through the `depend` clause. Memory regions are discovered via the symbolic region analysis of Section 4.2.1. In addition to pointers, we do not privatize values defined within a task region, and used outside it. That is to say: we privatize incoming def-use chains, but do not privatize outgoing chains. Values having this “outgoing chain property” are shared by default. We indicate this semantics via the `default(shared)` pragmas, as seen in line 8 of Figure 411.

Lastly, it is up to `TaskMiner` to finish the job and annotate those newly found Tasks with OpenMP directives in the source-code.

4.2.6 Mapping it all back into source-code

The analyses described in Sections 4.2.1-4.2.5 have been implemented at the level of the LLVM intermediate representation. We chose to implement those techniques at that level to benefit from the support of several data-flow analyses already in place in the LLVM infra-structure. In particular, this decision let us reuse LLVM’s scalar evolution [38, p.18], from which we derive our symbolic analysis.

Additionally, we could reuse LLVM’s program dependence graph, and its suite of powerful alias analyses. This toolbox gave us the necessary means to disambiguate pointers, albeit conservatively, and estimate safe limits to memory regions.

However, our annotations are still inserted in C programs. Having them in a high-level programming language has two main advantages: they are human-readable, and thus can be debugged, and they are compatible with compilers other than `clang`, such as `gcc`. Thus, a major part of this work was the engineering effort to map information from LLVM’s intermediate representation back into C.

4.2.6.1 The Scope Tree

To recover high-level information from the low-level IR, we have designed a supporting data-structure henceforth named the *scope tree*. The scope tree maps hammock regions into C constructions, such as `while` and `if-then-else` blocks. Each node of this graph represents a hammock region, augmented with meta-information, such as the program part that it represents. We keep track of these program parts via debugging information, which is appended into the LLVM IR via the `-g` flag passed to the `clang` compiler. We have an edge from node s_1 to node s_2 if region s_2 is nested within region s_1 . In the absence of control-flow optimizations such as loop unrolling or dead-code elimination, each hammock region corresponds to some structured code block (code

region delimitable by braces). Thus, we can find line numbers for each of the regions that we have marked as tasks after the expansion seen in Section 4.2.4.

4.2.6.2 Simplification

Before we annotate programs, we proceed to simplify these annotations. Simplification happens via a system of rewriting rules, which explores identities. Typical identities include, for instance, $x + 0 = x$, $\max(x, x) = x$ and $c \times \max(x, y) = \max(c \times x, c \times y)$. Identities are applied iteratively, until a fixed-point is reached. Because we do not support commutativity, a fixed-point is guaranteed to be always reached.

Notice that the source code that we produce is not totally equivalent to the original program augmented with annotations. To be able to insert annotations, we format the original code. This operation involves, for instance, breaking lines containing multiple statements, and inserting delimiting braces within every block in the program, even one-liners.

This said, it is still possible that a task, after expansion, maps to a single line that contains multiple statements, such as nested function calls. In this case, we do not annotate the target program. Chapter 5 provides data about TaskMiner’s capacity to annotate real-world programs.

4.2.6.3 Bounding Recursive Tasks

During the evaluation of the TaskMiner, we observed that recursive programs experienced performance losses due to the excessive creation of tasks. To avoid this kind of slowdown, we currently give users the possibility to bound the number of threads ever in flight, via a command line option, e.g., `./Taskminer -r 12` will limit the number of tasks to 12. We implement this feature directly at the source code level, as part of the final annotation of code. The solution is simple, yet, as the reader shall perceive in Chapter 5, it brings non-negligible benefits to recursive benchmarks. Task bounding is implemented via a global variable, statically linked in the program. This variable is `taskminer_depth_cutoff` in Figure 412. We insert code to increment it at the beginning of recursive functions that are invoked within task regions, and we insert code to decrement it at each return point of said functions.

Figure 412 illustrates the strategy that we use to limit the number of tasks in flight due to recursive function invocation. The parameter `DEPTH_CUTOFF` is determined by TaskMiner’s users.

There are more involved ways to bound the number of tasks. We believe that the state-of-the-art in the field today is the work of Iwasaki and Taura [44; 45]. These

```

1  static int taskminer_depth_cutoff;
2  long long fib(int n) {
3      taskminer_depth_cutoff++;
4      long long x, y;
5      if (n < 2) return n;
6      #pragma omp task untied default(shared) \
7          if(taskminer_depth_cutoff < DEPTH_CUTOFF)
8      x = fib(n - 1);
9      #pragma omp task untied default(shared) \
10         if(taskminer_depth_cutoff < DEPTH_CUTOFF)
11         y = fib(n - 2);
12     #pragma omp taskwait
13     taskminer_depth_cutoff--;
14     return x + y;
15 }

```

Figure 412. Bounding the creation of recursive tasks.

authors propose different techniques to limit the creation of tasks, be it through the replication of code, be it through the estimation of work, given function inputs. Pragma for the conditional creation of tasks, such as the one seen in lines 7 and 10 of Figure 412, lets us obtain much of the benefit of code versioning, as proposed by Iwasaki and Taura.

However, we found work estimation of recursive functions a task too difficult to accomplish in general. Quoting Iwasaki [44, p.355]: “*there are tasks which essentially do not have simple termination conditions (e.g., tasks traversing pointer-based trees)*”. Thus, although simple, our recursion counters are general enough, handling even such tasks that are hard to bound symbolically. Nevertheless, we still would like to explore further ways to cut-off extremely fine-grained tasks in the future.

Chapter 5

Experiments

In this Chapter we present the results for the evaluation that was carried out over the `TaskMiner` tool. We have implemented the techniques described in this dissertation in LLVM 3.9 [54]. All our experiments were performed in a 64-bits 12-core Intel(R) Xeon(R) CPU E5-2620 at 2.00GHz, with 32K of L1 cache, 256K of L2 cache, 15M of L3 cache and 15Gb of main memory. We use OpenMP 4.5, from November of 2015.

5.1 Evaluation overview

We assess `TaskMiner`'s performance under four different aspects. The main research questions involved in the `TaskMiner`'s evaluation are as follows:

- [Performance]: how do our automatically annotated programs compare against their sequential counterparts, or against manually annotated versions?
- [Optimizations]: what is the impact of the cost model (Section 4.2.3) and recursion bounding (Section 4.2.6) to the programs that we annotate?
- [Versatility]: how effective is `TaskMiner` in finding opportunities to annotate general benchmarks?
- [Scalability]: what is the runtime complexity of our implementation of `TaskMiner`?

We address each of these aspects in the sections to come.

5.2 TaskMiner’s performance

Figure 51 compares the runtime of programs produced by TaskMiner against either the original program or versions of these programs annotated manually. Automatic annotations compared favorably against manual interventions, and led to great performance gains. The Y-axis shows the speedup of either manual intervention or the TaskMiner upon the original programs, in number of times. Small numbers in boxes show speedup achieved by TaskMiner. In this experiment we use the benchmarks BSC-Bots [33] (fft to jacobi in Fig. 51) and Swan [63] (dfs to private). Both these benchmarks come with sequential and parallel (manually annotated) versions. Baseline and annotated sources are compiled with gcc-6 -O3.

The programs annotated by TaskMiner were faster than their sequential counterparts in 13 of the 16 samples. Most of the programs in these benchmarks were classic Divide and Conquer algorithms, such as Strassen’s matrix multiplication, knapsack and fft. In 6 of them, the automatically annotated programs were close and sometimes even slightly above the manually annotated versions. In three examples where TaskMiner fell behind the original samples, e.g., fib, floorplan and bellmandford (bell-ford), it produced code faster than the manually annotated competitor.

Visual inspection of bellmanford shows that the version parallelized manually is dispatching a large number of small tasks. Our cost model can prune some of these small tasks, but not enough to win on the sequential version. We emphasize that the

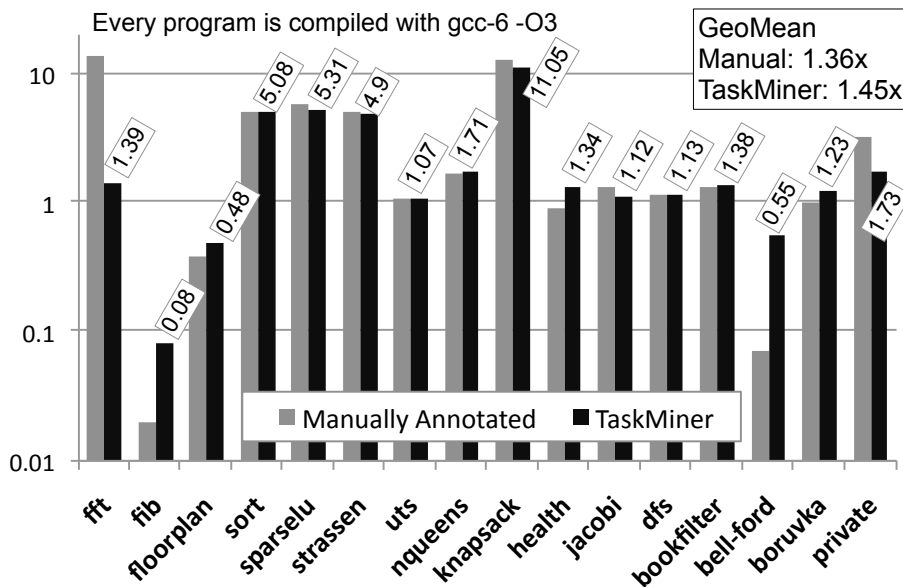


Figure 51. Speedup comparisons between programs annotated by TaskMiner.

human-annotated versions of our benchmarks have not been tested originally in our architecture. For instance, **BSC-Bots** was evaluated on an SGI Altix 4700 with 128 processors [33]. Thus, this difference in hardware might be accountable for the slowdown in some of the manually annotated samples. Nevertheless, this experiment demonstrates that *TaskMiner can deliver non-negligible speedups comparable with sequential and manually annotated programs.*

5.3 TaskMiner’s optimizations

TaskMiner deploys two ways to optimize task placement. Both these techniques are based on the idea of “task pruning”: we avoid creating tasks if we deem them unprofitable. The first technique prunes tasks judged unprofitable by the cost-model of Section 4.2.3; the second prunes tasks that are too deep in the recursion stack, as described in Section 4.2.6.

Figure 52 illustrates the benefits of these two techniques. We show the gains — in terms of speedup over the baseline — of the three benchmarks that benefit the most from each type of task pruning, among those seen earlier in Figure 51.

In three benchmarks, **private**, **boruvka** and **bellman-ford**, the cost model avoids creating tasks around loops that initialize data-structures. For instance, the following loop, in line 75 of **bellmanford.c**, would be parallelized by TaskMiner, were it not for the

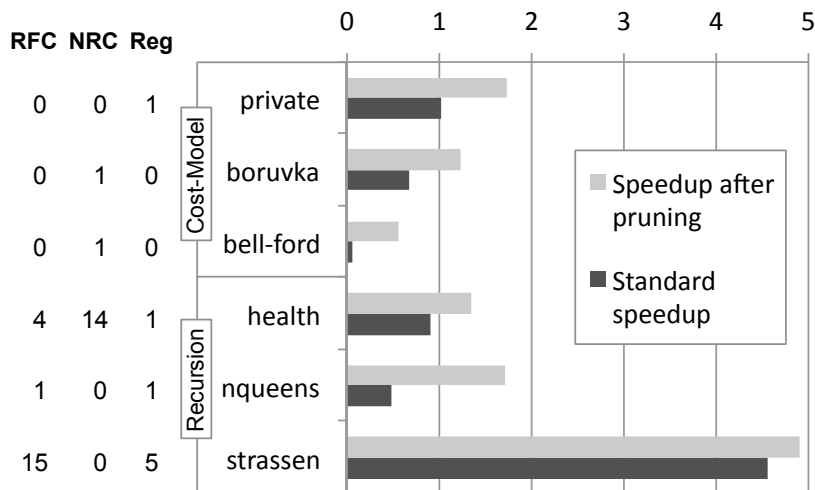


Figure 52. Benefit of task pruning. **RFC**: tasks created within Recursive Function Calls. **NRC**: interprocedural tasks created around Non-Recursive function Calls. **Reg**: tasks involving Regions without function calls.

cost-model marking it as unprofitable:

```
for (long unsigned i = 0; i < N; i++)
  for (long unsigned j = 0; j < N; j++)
    *(G + i * N + j) = rand();
```

Similarly, recursion bounding (Sec. 4.2.6), although simple, is effective in eliminating the excessive number of tasks. Figure 52 shows the effect of this optimization upon three BSC-Bots benchmarks: **health**, **nqueens** and **strassen**. These programs were designed to illustrate the parallelization of divide-and-conquer algorithms [33], and contain a large number of recursive calls (see **RFC**) in Figure 52. Pruning at higher levels of the recursion tree lets them deliver non-negligible speedups onto the baseline programs. Were pruning absent, then we would observe slowdowns in **health** and **nqueens**. This experiment shows that *our optimizations are effective to improve the quality of the code that we generate*.

5.4 TaskMiner’s versatility

The benchmarks used in Section 5.2 have been coded to demonstrate the power of parallel systems; hence, they have been written in a way that simplifies the discovery of parallelism in programs. In this section, we address the following question: “can TaskMiner find parallelism in general programs?”

To answer the proposed question, we have applied TaskMiner onto the 219 C programs available in the LLVM test suite, and have compared the annotated version against the sequential version compiled. TaskMiner annotates a benchmark if:

- (i) it can find symbolic bounds to every memory access used in a vane; and
- (ii) the vane is large enough to pay for the cost of creating threads.

Under these constraints, we have discovered tasks in 63 benchmarks. Figure 53 relates the number of tasks and the number of instructions in the 30 largest benchmarks that we have used. To avoid counting multiple C files for the same benchmark, Figure 53 contains only benchmarks that consists of a single file (present in LLVM’s **SingleSource** folder). As this experiment demonstrates, TaskMiner *can annotate a non-trivial number of real-world benchmarks*.

Most of the benchmarks that have been automatically annotated, i. e., 27, did not show speedups, because the regions marked as tasks were too small to influence the program’s runtime. We have observed slowdowns in 17 benchmarks. In this case,

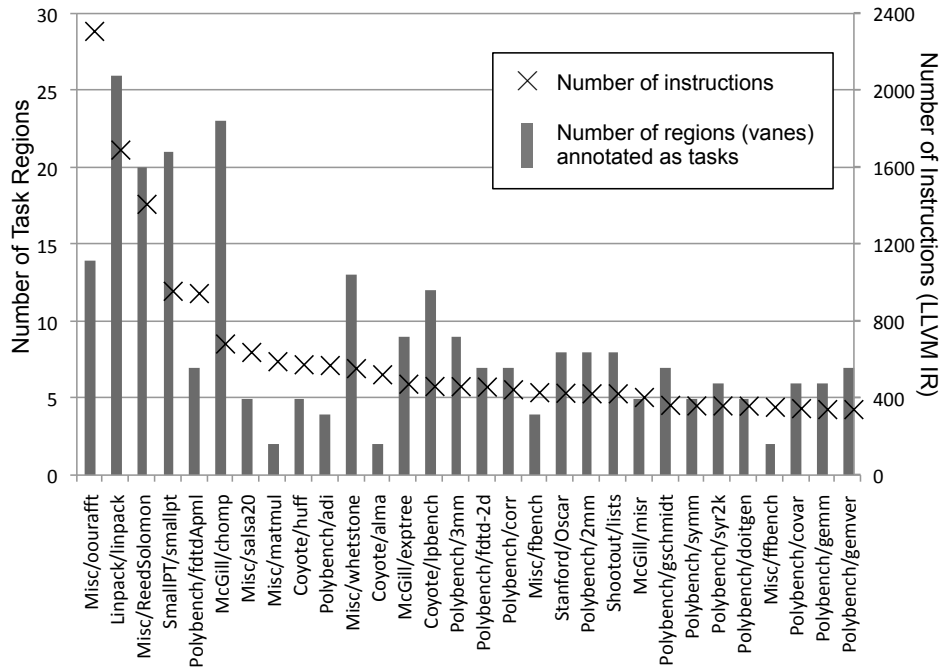


Figure 53. Relation between number of task regions and program size. Each benchmark is a complete C file.

interactions between data-structures end up forcing the OpenMP runtime to serialize execution of tasks. Most of these slowdowns were inferior to 10%. In one case, MiBench/office-stringsearch, our program was 11x slower.

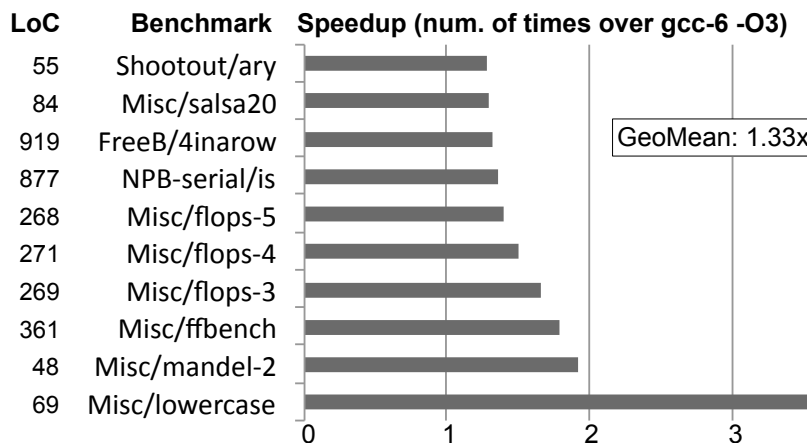


Figure 54. Speedups (in number of times) obtained by TaskMiner when applied onto the LLVM test suite. The larger the bar, the better. **LoC** stands for “Lines of Code”.

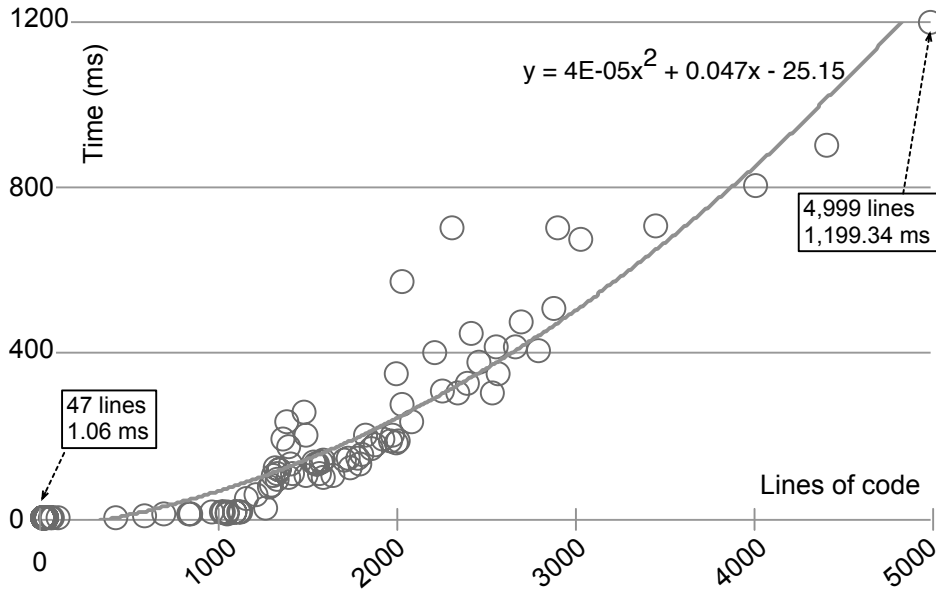


Figure 55. Runtime of TaskMiner vs size of input programs.

Nevertheless, we have measured speedups above 5% in 19 benchmarks. In *Misc/lowercase*, the speedup is above 3.5x, and in three cases, it is above 1.5x. Figure 54 shows our 10 largest speedups. We emphasize that this experiment did not involve any human intervention. Hence, *TaskMiner enables the discovery of parallelism at zero programming cost*.

5.5 TaskMiner's scalability

The analyses described in Chapter 4 have a worst-case quadratic time. This is the worst-case scenario of the symbolic range analysis of Section 4.2.1 and the task-expansion algorithm of Section 4.2.4. Nevertheless, our implementation is fast in practice. To support this statement, we have used *CSmith* [91], a random code generator, to produce 100 programs of varying sizes, which we then fed to the *TaskMiner*.

Figure 55 shows the result of this experiment. Each dot represents one of the 100 program generated by *CSmit*. We have fit a degree-2 polynomial on this dataset. We cannot control the size of programs produced by *CSmith*; hence, it is difficult to push the limit above 5,000 LoC. Our largest program had 4,999 lines of C, and *TaskMiner* could process it in 1.2 seconds. the R^2 of polynomials of degree 1, 2, 3 and 4 is, respectively, 0.795, 0.890, 0.909 and 0.911, suggesting very small differences between polynomials of degree 2 or more.

Chapter 6

Related work

Mainstream compilers have only recently added support to OpenMP 4.0’s task parallelism. An implementation of `clang` supporting tasks was released in March 14th, 2014. A few weeks later, in April 22nd, 2014, such support was also announced in `gcc` 4.9. Because the necessary infra-structure for the implementation of tasks is new, currently there are no tools, other than `TaskMiner`, that annotate programs with these directives. Nevertheless, there exist much research aiming at the automatic parallelization of programs. This section examines elements in this list related to our work.

Directive-based code annotation standards provide programmers with an easy-to-use parallel programming model. Task-based extensions to these standards have resulted in OpenMP 4.X, StarSs [10; 65; 68; 82] and OmpSs [17; 30]. Such programming models come with tools that help programmers to find the best annotations for their code. For example, Tareador [4] enables a programmer, by means of a graphical interface, to annotate sequential code, thus allowing the identification of potential task parallelization opportunities. Contrary to Tareador, this paper proposes an approach that enables the automatic insertion of OpenMP task annotations to relevant fragments of a sequential program.

The techniques that we use to map program regions to tasks are similar to analyses previously used in the generation of code for data-flow machines. Data-flow programming was originally proposed as a candidate to enable task-based parallelism [1; 23; 39]. Agrawal et al. [1] extended data-flow programming with task input/output specifications in Cilk++. Vandierendonck et al. [86] further extended Cilk++ with dependency clauses to facilitate the design of complex parallelization patterns. Both groups presented a unified scheduler based on fork-join parallelism [87] that enabled the execution of task-based applications. Other approaches have also used data-flow graphs to exploit parallelism, like Data-Driven Tasks [81], where the programmer can

use `put/await` clauses to determine task arguments before execution.

Function-based task parallelism was proposed by Gupta *et al.* [39] to use function arguments as a way to specify task dependencies. Notice that such research efforts focused on giving to programmers the tools to *construct* parallel programs. The automatic annotation of ordinary programs with data-flow constructs was not among their goals.

Many systems have been developed to extract task parallelism from sequential programs (semi) automatically. Examples include OSCAR (Optimally SCheduled Advanced multiprocessor), Multigrain Parallelizing Compiler [43; 48], MAPS (MPSoC Application Programming Studio) [18; 20], and DiscoPoP (Discovery of Potential Parallelism) [27; 57].

Besides parallelization systems, tools like Paraver [21; 22], Aftermath [29], DAGvis [41], and TEMANEJO [16; 15; 80] were designed to enable performance analysis and visualization of task-based programs. Although these tools help the programmer in adapting code to run in parallel, they are not fully automatic. For instance, the work that is the closest to ours, in reach and effectiveness, in our opinion, is the suite of techniques proposed by Ravishankar *et al.* [73]. The type of irregular loops that they handle is impressive; however, the lack of the runtime support à la OpenMP 4.X still requires them to modify code before parallelization. In their words: “*For all benchmarks and applications, all functions were inlined, and arrays of structures were converted to structures of arrays for use with our prototype compiler.*”

Chapter 7

Conclusion

This dissertation has described a methodology to annotate programs with task parallel **pragmas**, which we could demonstrate to be effective on general programs. This methodology does not introduce any fundamentally new static analysis or code optimization; in this field, we claim no contribution. Instead, our contributions lay into the overall design that eventually emerged from a two-year long effort to combine existing compilation techniques towards the goal of profiting from the powerful runtime system that OpenMP’s task parallelism brings forward.

During this period, we reached many dead-ends. Nevertheless, this experience let us single out a few elements from the compiler literature, such as symbolic range analysis and windmills, which we could use to solve challenges related to automatic code annotation. Much work is still left to be done, until we can reach a stage in which automatic annotations can consistently beat manual interventions. In particular, our methodology asks for more aggressive tuning strategies. Techniques such as Trancoso’s [28], Iwasaki’s [45] or Emani’s [34] have already been shown to be effective in this domain, and we want to explore them further. More specifically, the first public release of **TaskMiner** is already expected to feature Iwasaki’s cut-off [45].

Differently from data parallelism, the nature of task parallelism is much more dynamic. The methods described in this dissertation are, in majority, static. However, we employed dynamic guards in the form of **pragma** clauses to interfere in the behavior of the tasks. It’s important to notice, though, that we weren’t able to find greatly discernible gains in many benchmarks. The act of finding Task Parallelism statically is, still, a huge challenge.

It would be unrealistic to think that irregular parallelism could be found statically without trade-off issues. In summary, we learned that to find Task Parallelism statically we need to be a bit less conservative than the compiler and assume that some

dependencies might not always occur during runtime. Thus, the analyses we implemented rely on repetition structures in the Program Dependence Graph that are not entirely clear of data dependencies. These structures evidence code that is likely to be executed multiple times. In a traditional parallel paradigm, should the identified code bear any data iteration dependence pointed by the compiler, it would not be parallelized. Yet, we assume that this code is a potential task. The **TaskMiner** archetype employs a cost model to assess such task candidates. The cost model carries out the main issue related to finding tasks: it decides whether that task candidate is worth it or not.

In reality, the cost model is the crucial step in mining tasks statically. When studying forms of assessing a Task's profitability, three topics are fundamental: first, we should consider the number of static input and output dependencies for a given Task. Each dependency turns into an edge in the Task Graph; hence, it's important to take those into account when deciding whether that Task should be dispatched or not. Second, we should have an idea of how much it costs for our Runtime to dispatch and maintain a Task up and running. If the Task candidate is not coarse enough, then it might not lead to performance gains. In fact, it might even lead to slowdowns. Therefore, it is clear that finding a good workload threshold is essential for reaching performance gains. Lastly, we should also consider the conflict between the size of the tasks and the amount of parallelism that these tasks would provide. When splitting the program into tasks, we desire the maximum number of concurrent tasks possible, as long as those are above a certain workload threshold.

We argue that, despite the fact that **TaskMiner** annotated benchmarks that turned out to be slower than their sequential counterparts, our method is quick, flexible, automatic and highly customizable. The time an average programmer would spend setting up **TaskMiner** for a given runtime and a target architecture might pay off. We believe that, although not precise, the only way of mining task parallelism statically revolves around setting up a good cost model for task candidates. Should one decide to use a profiler or a strictly dynamic approach, it might not have results as fast as when using the **TaskMiner**. This is, once again, a classical clash between time and precision. Either we spend a considerable amount of time analyzing several executions of a given program, to find proper opportunities for task parallelism, or we simply look for code that could be parallelized if it fits a specific cost model.

Today, at the end of our work, we come to some conclusions. We split the act of task parallelism scavenging into three steps. We register here what we believe are the crucial problems to be solved in each of these steps. In step one, we find independent code that is likely to be executed more than once. Task parallelism would be easy

to watch if the compilers were a 100% exact in their stated data dependencies. So how do we increase the precision of the compiler when stating dependencies between tasks, without any extra cost? In step two, we determine if those potential tasks are likely to bring speed up gains. The problem related to this step is defined by this question: how do we infer a dynamic feature statically? Is it possible to have a good method to approximate a task's cost without running it? And lastly, step three consists in finding the highest level of parallelization that a program can reach. This last problem is related to finding the balance between number of tasks and size of tasks, and it's believed to be the final objective in automatic parallelization: how do we fully parallelize a program?

I thank the committee for reading through this text and assessing the content here presented. During this long period, I have faced many different changes of direction, and I realized that this is naturally the path of research. We never know which new question an answer will lead us to. Thus, I finalize this dissertation by quoting Turing [83]:

We can only see a short distance ahead, but we can see plenty there that needs to be done.

Bibliography

- [1] K. Agrawal, C. E. Leiserson, and J. Sukha. Executing task graphs using work-stealing. In *Proceedings of the 2010 IEEE International Parallel and Distributed Processing Symposium*, pages 1–12, Atlanta, Georgia, US, 2010. IEEE.
- [2] P. Alves, F. Gruber, J. Doerfert, A. Lamprineas, T. Grosser, F. Rastello, and F. M. Q. a. Pereira. Runtime pointer disambiguation. In *Proceedings of Object-Oriented Programming, Systems, Languages & Applications (OOPSLA)*, pages 589–606, New York, NY, USA, 2015. ACM.
- [3] J. M. Andión, M. Arenaz, F. Bodin, G. Rodríguez, and J. T. no. Locality-aware automatic parallelization for GPGPU with OpenHMPP directives. *International Journal of Parallel Programming*, 44(3):620–643, 2016.
- [4] E. Ayguadé, R. M. Badia, D. Jiménez, J. R. Herrero, J. Labarta, V. Subotic, and G. Utrera. Tareador: A tool to unveil parallelization strategies at undergraduate level. In *Proceedings of Workshop on Computer Architecture Education (WCAE)*, pages 1:1–1:8, New York, NY, USA, 2015. ACM.
- [5] E. Ayguadé, N. Coptý, A. Duran, J. Hoeftlinger, Y. Lin, F. Massaioli, X. Teruel, P. Unnikrishnan, and G. Zhang. The design of openmp tasks. *Transactions on Parallel and Distributed Systems*, 20(3):404–418, 2009.
- [6] E. Ayguadé, N. Coptý, A. Duran, J. Hoeftlinger, Y. Lin, F. Massaioli, X. Teruel, P. Unnikrishnan, and G. Zhang. The design of OpenMP tasks. *IEEE Transactions on Parallel and Distributed Systems*, 20(3):404–418, 2009.
- [7] S. S. Bagsorkhi, M. Delahaye, S. J. Patel, W. D. Gropp, and W.-M. W. Hwu. An adaptive performance modelling tool for GPU architectures. In *Proceedings of Principles and Practice of Parallel Programming (PPoPP)*, pages 105–114, New York, NY, USA, 2010. ACM.

- [8] T. Bai, C. Ding, and P. Li. Assessing safe task parallelism in SPEC 2006 INT. In *Proceedings of the International Symposium on Cluster, Cloud and Grid Computing (CCGrid)*, pages 402–411. IEEE, 2015.
- [9] M. Bauer, S. Treichler, E. Slaughter, and A. Aiken. Legion: Expressing locality and independence with logical regions. In *Proceedings of the international conference on high performance computing, networking, storage and analysis*, page 66. IEEE Computer Society Press, 2012.
- [10] P. Bellens, J. M. Perez, R. M. Badia, and J. Labarta. Cellss: a programming model for the Cell BE architecture. In *Proceedings of Conference on Supercomputing (SC)*, pages 1–5, Tampa, FL, USA, Nov 2006. IEEE.
- [11] C. Bertolli, S. F. Antao, A. E. Eichenberger, K. O’Brien, Z. Sura, A. C. Jacob, T. Chen, and O. Sallénave. Coordinating GPU threads for OpenMP 4.0 in LLVM. In *Proceedings of the LLVM Workshop in High Performance Computing (LLVM-HPC)*, pages 12–21, New York, NY, USA, 2014. IEEE.
- [12] W. Blume and R. Eigenmann. Symbolic range propagation. In *Proceedings of the International Parallel & Distributed Processing Systems Symposium (IPPS)*, pages 357–363, Washington, DC, USA, 1994. IEEE.
- [13] R. D. Blumofe, C. F. Joerg, B. C. Kuszmaul, C. E. Leiserson, K. H. Randall, and Y. Zhou. Cilk: An efficient multithreaded runtime system. *Journal of parallel and distributed computing*, 37(1):55–69, 1996.
- [14] G. Bosilca, A. Bouteiller, A. Danalis, T. Herault, P. Lemarinier, and J. Dongarra. Dague: A generic distributed DAG engine for high performance computing. *Parallel Computing*, 38(1):37–51, 2012.
- [15] S. Brinkmann, J. Gracia, and C. Niethammer. Task debugging with TEMANEJO. In *Proceedings of Tools for High Performance Computing (HPC) 2012*. Springer Berlin Heidelberg, 2013.
- [16] S. Brinkmann, J. Gracia, C. Niethammer, and R. Keller. TEMANEJO — a debugger for task based parallel programming models. In *Proceedings of Parallel Computing Conference (ParCo)*, pages 11–13. IOS Press, 2011.
- [17] J. Bueno, L. Martinell, A. Duran, M. Farreras, X. Martorell, R. M. Badia, E. Ayguadé, and J. Labarta. Productive cluster programming with OmpSs. In *Proceedings of International European Conference on Parallel and Distributed Computing (Euro-Par)*, pages 555–566, Berlin, Heidelberg, 2011. Springer-Verlag.

- [18] J. Castrillon, R. Leupers, and G. Ascheid. Maps: Mapping concurrent dataflow applications to heterogeneous MPSoCs. *IEEE Transactions on Industrial Informatics*, 9(1):527–545, 2013.
- [19] M. Castro, L. F. W. Góes, and J.-F. Méhaut. Adaptive thread mapping strategies for transactional memory applications. *Journal of Parallel and Distributed Computing*, 74(9):2845–2859, 2014.
- [20] J. Ceng, J. Castrillon, W. Sheng, H. Scharwachter, R. Leupers, G. Ascheid, H. Meyr, T. Isshiki, and H. Kunieda. MAPS: An integrated framework for MP-SoC application parallelization. In *Proceedings of The 45th ACM/IEEE Design Automation Conference*, pages 754–759, Anaheim, CA, USA, 2008. IEEE.
- [21] B. S. Center. Extrae project website. <https://tools.bsc.es/extrae>, 2018. Visited on 2018-07-01.
- [22] B. S. Center. Paraver: a flexible performance analysis tool. <https://tools.bsc.es/paraver>, 2018. Visited on 2018-07-01.
- [23] E. Chan, E. S. Quintana-Orti, G. Quintana-Orti, and R. van de Geijn. Supermatrix out-of-order scheduling of matrix operations for SMP and multi-core architectures. In *Proceedings of the 19th Annual ACM Symposium on Parallel Algorithms and Architectures*, pages 116–125, New York, NY, USA, 2007. ACM.
- [24] D. E. Culler, J. P. Singh, and A. Gupta. *Parallel computer architecture: a hardware/software approach*. Gulf Professional Publishing, 1999.
- [25] R. Cytron. Doacross: Beyond vectorization for multiprocessors. In *Proceedings of the International Conference on Parallel Processing, 1986*, 1986.
- [26] L. L. P. Da Mata, F. M. Q. Pereira, and R. Ferreira. Automatic parallelization of canonical loops. *Science of Computer Programming*, 78(8):1193–1206, 2013.
- [27] T. U. Darmstadt. DiscoPoP (discovery of potential parallelism) project website. <https://www.parallel.informatik.tu-darmstadt.de/multicore-group/discopop/>, 2018. Visited on 2018-07-01.
- [28] A. Diavastos and P. Trancoso. Auto-tuning static schedules for task data-flow applications. *Hand*, 400(500):600, 2017.
- [29] A. Drebes, A. Pop, K. Heydemann, A. Cohen, and N. Drach. Aftermath: A graphical tool for performance analysis and debugging of fine-grained task-parallel programs and run-time systems. In *Proceedings of Workshop on Programmability and*

- Architectures for Heterogeneous Multicores (MULTIPROG)*, pages 80–88, Vienna, Austria, Jan. 2014. ACM.
- [30] A. Duran, E. Ayguadé, R. M. Badia, J. Labarta, L. Martinell, X. Martorell, and J. Planas. OmpSs: A proposal for programming heterogeneous multi-core architectures. *Parallel Processing Letters*, 21(02):173–193, 2011.
- [31] A. Duran, J. Corbalán, and E. Ayguadé. An adaptive cut-off for task parallelism. In *Proceedings of Conference on Supercomputing (SC)*, pages 36:1–36:11, Piscataway, NJ, USA, 2008. ACM/IEEE.
- [32] A. Duran, J. M. Perez, E. Ayguadé, R. M. Badia, and J. Labarta. Extending the OpenMP tasking model to allow dependent tasks. In *Proceedings of International Workshop on OpenMP (IWOMP)*, pages 111–122, Heidelberg, Germany, 2008. Springer.
- [33] A. Duran, X. Teruel, R. Ferrer, X. Martorell, and E. Ayguadé. Barcelona OpenMP tasks suite: A set of benchmarks targeting the exploitation of task parallelism in OpenMP. In *Proceedings of International Conference on Parallel Processing (ICPP)*, pages 124–131, Washington, DC, USA, 2009. IEEE.
- [34] M. K. Emani and M. O’Boyle. Celebrating diversity: A mixture of experts approach for runtime mapping in dynamic environments. In *Proceedings of Conference on Programming Language Design and Implementation (PLDI)*, pages 499–508, New York, NY, USA, 2015. ACM.
- [35] K. Fatahalian, D. R. Horn, T. J. Knight, L. Leem, M. Houston, J. Y. Park, M. Erez, M. Ren, A. Aiken, W. J. Dally, et al. Sequoia: Programming the memory hierarchy. In *Proceedings of Conference on Supercomputing (SC)*, page 83. ACM, 2006.
- [36] J. Ferrante, K. J. Ottenstein, and J. D. Warren. The program dependence graph and its use in optimization. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 9(3):319–349, 1987.
- [37] L. F. W. Góes, C. P. Ribeiro, M. Castro, J.-F. Méhaut, M. Cole, and M. Cintra. Automatic skeleton-driven memory affinity for transactional worklist applications. *International Journal of Parallel Programming*, 42(2):365–382, 2014.
- [38] T. Grosser, A. Größlinger, and C. Lengauer. Polly - performing polyhedral optimizations on a low-level intermediate representation. *Parallel Processing Letters*, 22(4):28 pages, 2012.

- [39] G. Gupta and G. S. Sohi. Dataflow execution of sequential imperative programs on multicore architectures. In *Proceedings of the International Symposium on Microarchitecture (MICRO)*, pages 59–70, New York, NY, USA, 2011. ACM.
- [40] E. Hoffman, J. Loessi, and R. Moore. Constructions for the solution of the n queens problem. *Mathematics Magazine*, 42(2):66–72, 1969.
- [41] A. Huynh, D. Thain, M. Pericàs, and K. Taura. DAGViz: A DAG visualization tool for analyzing task-parallel program traces. In *Proceedings of the 2nd Workshop on Visual Performance Analysis*, pages 3:1–3:8, New York, NY, USA, 2015. ACM.
- [42] W.-m. Hwu. What is ahead for parallel computing. *Journal of Parallel and Distributed Computing*, 74(7):2574–2581, 2014.
- [43] K. Ishizaka, M. Obata, and H. Kasahara. Coarse-grain task parallel processing using the openmp backend of the oscar multigrain parallelizing compiler. In *Proceedings of International Symposium on High Performance Computing (HPC)*, pages 457–470. Springer, 2000.
- [44] S. Iwasaki and K. Taura. Autotuning of a cut-off for task parallel programs. In *Proceedings of International Symposium on Embedded Multicore/Many-core Systems-on-chip (MCSoc)*, pages 353–360, Piscataway, NJ, USA, 2016. IEEE.
- [45] S. Iwasaki and K. Taura. A static cut-off for task parallel programs. In *Proceedings of Parallel Architectures and Compiler Techniques (PACT)*, pages 139–150, Piscataway, NJ, USA, 2016. IEEE.
- [46] J. Jaeger, P. Carribault, and M. Pérache. Fine-grain data management directory for OpenMP 4.0 and OpenACC. *Concurrency and Computation: Practice and Experience*, 27(6):1528–1539, 2015.
- [47] J. C. Jenista, Y. H. Eom, and B. Demsky. OoOJava: an out-of-order approach to parallel programming. In *Proceedings of the 2nd USENIX conference on Hot Topics in Parallelism*, pages 11–11. USENIX Association, 2010.
- [48] H. Kasahara, M. Obata, and K. Ishizaka. Automatic coarse grain task parallel processing on SMP using OpenMP. In *International Workshop on Languages and Compilers for Parallel Computing*, pages 189–207. Springer, 2000.
- [49] K. Kennedy and J. R. Allen. *Optimizing compilers for modern architectures: a dependence-based approach*. Morgan Kaufmann Publishers Inc., 2001.

- [50] G. A. Kildall. A unified approach to global program optimization. In *Proceedings of Symposium on Principles of Programming Languages (POPL)*, pages 194–206, New York, NY, USA, 1973. ACM.
- [51] M. Kulkarni, M. Burtcher, R. Inkulu, K. Pingali, and C. Casçaval. How much parallelism is there in irregular applications? In *ACM SIGPLAN Notices*, volume 44, pages 3–14. ACM, 2009.
- [52] M. Kulkarni, K. Pingali, B. Walter, G. Ramanarayanan, K. Bala, and L. P. Chew. Optimistic parallelism requires abstractions. *ACM SIGPLAN Notices*, 42(6):211–222, 2007.
- [53] J. LaGrone, A. Aribuki, C. Addison, and B. Chapman. A runtime implementation of OpenMP tasks. In *Proceedings of International Workshop on OpenMP (IWOMP)*, pages 165–178, Heidelberg, Germany, 2011. Springer.
- [54] C. Lattner and V. Adve. LLVM: A compilation framework for lifelong program analysis & transformation. In *Proceedings of International Symposium on Code Generation and Optimization (CGO)*, pages 75–81, Washington, DC, USA, 2004. IEEE Computer Society.
- [55] S. Lee and R. Eigenmann. OpenMPC: Extended OpenMP programming and tuning for GPUs. In *Proceedings of Conference on Supercomputing (SC)*, pages 1–11, Washington, DC, USA, 2010. IEEE.
- [56] C. E. Leiserson. The Cilk++ concurrency platform. In *Proceedings of Design and Automation Conference (DAC)*, pages 522–527, New York, NY, USA, 2009. ACM.
- [57] Z. Li, R. Atre, Z. Huda, A. Jannesari, and F. Wolf. Unveiling parallelization opportunities in sequential programs. *Journal of Systems and Software*, 117(C):282–295, 2016.
- [58] D. C. S. Lucas and G. Araujo. The batched DOACROSS loop parallelization algorithm. In *Proceedings of High Performance Computing & Simulation International Conference (HPCS)*, pages 476–483. IEEE, 2015.
- [59] M. Maalej, V. Paisante, P. Ramos, L. Gonnord, and F. M. Q. a. Pereira. Pointer disambiguation via strict inequalities. In *Proceedings of International Symposium on Code Generation and Optimization (CGO)*, pages 134–147, Piscataway, NJ, USA, 2017. IEEE Press.

- [60] C. Meenderinck and B. Juurlink. Nexus: Hardware support for task-based programming. In *Proceedings of Symposium on Digital System Design (DSD)*, pages 442–445, New York, NY, USA, 2011. Springer.
- [61] G. S. D. Mendonça, B. C. F. Guimarães, P. R. O. Alves, F. M. Q. Pereira, M. M. Pereira, and G. Araújo. Automatic insertion of copy annotation in data-parallel programs. In *Proceedings of International Symposium on Computer Architecture and High Performance Computing (SBAC-PAD)*, pages 34–41. IEEE, 2016.
- [62] G. Mendonça, B. Guimarães, P. Alves, M. Pereira, G. Araújo, and F. M. Q. a. Pereira. DawnCC: Automatic annotation for data parallelism and offloading. *ACM Transactions on Architecture and Code Optimization*, 14(2):13:1–13:25, 2017.
- [63] R. E. Moreira, S. Collange, and F. M. Quintão Pereira. Function call re-vectorization. In *Proceedings of Principles and Practice of Parallel Programming (PPoPP)*, pages 313–326, New York, NY, USA, 2017. ACM.
- [64] C. Nguyen and P. J. Rhodes. Tipp: Parallel delaunay triangulation for large-scale datasets. In *Proceedings of the 30th International Conference on Scientific and Statistical Database Management, SSDBM '18*, pages 8:1–8:12, New York, NY, USA, 2018. ACM.
- [65] J. M. Perez, R. M. Badia, and J. Labarta. A dependency-aware task-based programming environment for multi-core architectures. In *Proceedings of International Conference on Cluster Computing*, pages 142–151, Tsukuba, Japan, 2008. IEEE.
- [66] G. Piccoli, H. N. Santos, R. E. Rodrigues, C. Pousa, E. Borin, and F. M. Quintão Pereira. Compiler support for selective page migration in NUMA architectures. In *Proceedings of Parallel Architectures and Compiler Techniques (PACT)*, pages 369–380, New York, NY, USA, 2014. ACM.
- [67] K. Pingali, D. Nguyen, M. Kulkarni, M. Burtscher, M. A. Hassaan, R. Kaleem, T.-H. Lee, A. Lenharth, R. Manevich, M. Méndez-Lojo, D. Prountzos, and X. Sui. The Tao of parallelism in algorithms. In *Proceedings of Conference on Programming Language Design and Implementation (PLDI)*, pages 12–25, New York, NY, USA, 2011. ACM.
- [68] J. Planas, R. M. Badia, E. Ayguadé, and J. Labarta. Hierarchical task-based programming with StarSs. *International Journal of High Performance Computing Applications*, 23(3):284–299, 2009.

- [69] J. Planas, R. M. Badia, E. Ayguadé, and J. Labarta. SSMART: Smart scheduling of multi-architecture tasks on heterogeneous systems. In *Proceedings of Workshop on Accelerator Programming Using Directives (WACCPD)*, pages 1:1–1:11, New York, NY, USA, 2015. ACM.
- [70] G. Poesia, B. Guimarães, F. Ferracioli, and F. M. Q. a. Pereira. Static placement of computation on heterogeneous devices. *Proceedings of Object-Oriented Programming, Systems, Languages and Applications (OOPSLA)*, 1:50:1–50:28, 2017.
- [71] P. Pratikakis, H. Vandierendonck, S. Lyberis, and D. S. Nikolopoulos. A programming model for deterministic task parallelism. In *Proceedings of the Workshop on Memory Systems Performance and Correctness*, pages 7–12. ACM, 2011.
- [72] P. Ramos, G. Mendonca, G. Leobas, G. Araujo, C. Divino, and F. Magno. Automatic identification and annotation of tasks in structured programs. In *Proceedings of Parallel Architectures and Compiler Techniques (PACT)*, Cyprus, 2018. ACM.
- [73] M. Ravishankar, J. Eisenlohr, L.-N. Pouchet, J. Ramanujam, A. Rountev, and P. Sadayappan. Automatic parallelization of a class of irregular loops for distributed memory systems. *ACM Transactions on Parallel Computing*, 1(1):7:1–7:37, 2014.
- [74] H. G. Rice. Classes of Recursively Enumerable Sets and Their Decision Problems. *Transactions of the American Mathematical Society*, 74(2):358–366, 1953.
- [75] L. Rideau, B. P. Serpette, and X. Leroy. Tilting at windmills with Coq: Formal verification of a compilation algorithm for parallel moves. *Journal of Automated Reasoning*, 40(4):307–326, 2008.
- [76] C. J. Rossbach, Y. Yu, J. Currey, J.-P. Martin, and D. Fetterly. Dandelion: a compiler and runtime for heterogeneous systems. In *Proceedings of the Symposium on Operating Systems Principles*, pages 49–68. ACM, 2013.
- [77] S. Rus, L. Rauchwerger, and J. Hoeflinger. Hybrid analysis: Static and dynamic memory reference analysis. In *Proceedings of the International Conference on Supercomputing (ICS)*, pages 251–283, Piscataway, NJ, USA, 2002. IEEE.
- [78] D. B. Skillicorn. Models for practical parallel computation. *International Journal of Parallel Programming*, 20(2):133–158, 1991.
- [79] O. Standard. The OpenACC programming interface. Technical report, CAPs, 2013.

- [80] T. H. P. C. C. Stuttgart. TEMANEJO project website. <https://www.hlr.de/de/solutions-services/service-portfolio/programming/hpc-development-tools/temanejo/>, 2018. Visited on 2018-07-01.
- [81] S. Tasirlar and V. Sarkar. Data-driven tasks and their implementation. In *Proceedings of the International Conference on Parallel Processing*, pages 652–661, Taipei City, Taiwan, 2011. IEEE Computer Society.
- [82] E. Tejedor, M. Farreras, D. Grove, R. M. Badia, G. Almasi, and J. Labarta. ClusterSs: A task-based programming model for clusters. In *Proceedings of the International Symposium on High-Performance Parallel and Distributed Computing (HPDC)*, pages 267–268, New York, NY, USA, 2011. ACM.
- [83] A. Turing. Computing machinery and intelligence. page 460, 1950.
- [84] G. Tzenakis, A. Papatriantafyllou, J. Kesapides, P. Pratikakis, H. Vandierendonck, and D. S. Nikolopoulos. BDDT: block-level dynamic dependence analysis for deterministic task-based parallelism. In *ACM SIGPLAN Notices*, volume 47, pages 301–302. ACM, 2012.
- [85] P. Unnikrishnan, J. Shirako, K. Barton, S. Chatterjee, R. Silvera, and V. Sarkar. A practical approach to DOACROSS parallelization. In *Proceedings of the European Conference on Parallel Processing*, pages 219–231. Springer, 2012.
- [86] H. Vandierendonck, P. Pratikakis, and D. S. Nikolopoulos. Parallel programming of general-purpose programs using task-based programming models. In *Proceedings of the 3rd USENIX Conference on Hot Topic in Parallelism*, pages 13–13, Berkeley, CA, USA, 2011. USENIX Association.
- [87] H. Vandierendonck, G. Tzenakis, and D. S. Nikolopoulos. A unified scheduler for recursive and task dataflow parallelism. In *Proceedings of Parallel Architectures and Compiler Techniques (PACT)*, pages 1–11, Galveston, Texas, USA, 2011. IEEE.
- [88] P. Virouleau, P. BRUNET, F. Broquedis, N. Furmento, S. Thibault, O. Aumage, and T. Gautier. Evaluation of OpenMP dependent tasks with the KASTORS benchmark suite. In *Proceedings of International Workshop on OpenMP (IWOMP)*, pages 16 – 29, Heidelberg, Germany, 2014. Springer.
- [89] C.-K. Wang and P.-S. Chen. Generating task clauses for OpenMP programs. 2014.

- [90] Y. Wu and J. R. Larus. Static branch frequency and program profile analysis. In *Proceedings of the International Symposium on Microarchitecture (MICRO)*, pages 1–11, New York, NY, USA, 1994. ACM.
- [91] X. Yang, Y. Chen, E. Eide, and J. Regehr. Finding and understanding bugs in C compilers. In *Proceedings of Conference on Programming Language Design and Implementation (PLDI)*, pages 283–294, New York, NY, USA, 2011. ACM.