Automatic Parallelization of Recursive Functions with Rewriting Rules

Rodrigo C. O. Rocha
University of Edinburgh, UK
r.rocha@ed.ac.uk

Luís F. W. Góes
PUC Minas, Brazil
lfwgoes@pucminas.br

Fernando M. Q. Pereira
UFMG, Brazil
fernando@dcc.ufmg.br

Abstract

Functional programming languages have, since their early days, being thought as the holy grail of parallelism. And, in fact, the absence of race conditions, coupled with algorithmic skeletons such as map and reduce, have given developers the opportunity to write many different techniques aimed at the automatic parallelization of programs. However, there are many functional programs that are still difficult to parallelize. This difficulty stems from many factors, including the complex syntax of recursive functions. This paper provides new equipment to deal with this problem. Such instrument consists of an insight, plus a code transformation that is enabled by the said insight. Concerning the first contribution, we demonstrate that many recursive functions can be rewritten as a combination of associative operations. We group such functions into two categories, which involve monoid and semiring operations. Each of these categories admits a parallel implementation. To demonstrate the effectiveness of this idea, we have implemented an automatic code rewriting tool for Haskell, and have used it to convert six well-known recursive functions to algorithms that run in parallel. Our tool is totally automatic, and it is able to deliver non-trivial speedups onto the sequential version of the programs that it receives. In particular, we have observed improvements of 8.18× and 65%, on average, for the monoid and semiring-based functions, respectively.

Keywords: recursive functions, parallel computing, functional programming, algebraic framework, abstract algebra

1. Introduction

The advent of multi-core computers has greatly spread the use of parallel programming among application developers. Yet, writing code that runs in parallel is still a difficult and error-prone task. Thus, the automatic parallelization of code has surfaced as an effective alternative to the development of high-performant programs [1, 2, 3]. In this sense, functional programming languages appear as a promising alternative to the development of parallel code. They provide referential transparency, reducing shared data and eliminating side effects, which makes automatic parallelization much easier. However, in spite of years of research, automatic generation of parallelism, out of functional code, is not a solved problem [4, 5]. Testimony of this last statement is the fact that functional code is still manually parallelized, usually by means of parallel skeletons [5, 6, 7].

The main challenge faced by automatic parallelization tools in functional languages is the fact that parallelism is often hidden under the syntax of complex recursive functions. There are several techniques to discover parallelism, such as work targeting list homomorphisms [8, 9, 10, 11], or the work of Fisher and Ghuloum [12], which parallelizes imperative loops that can be translated as compositions of functions. Nevertheless, the programming languages community still lacks approaches to infer parallelism on recursive functions automatically. The goal of this paper is to contribute to the solution of this omission by extending the family of recursive functions that can be parallelized automatically.

To achieve this objective, we propose an algebraic framework for parallelizing two special classes of linear recursive functions. A linear recursive function is a function that only makes a single call to itself each time the function runs. The function that we parallelize need to have two additional properties. First: the recursive function must contain only operations that can be used to define monoids or semirings. Second: the propagation of arguments between recursive calls has to be defined by an invertible function. The proposed framework is based on the theory introduced by Fisher and Ghuloum [12]. Those authors have designed and tested an approach to parallelize imperative loops by transforming them in recurrence relations defined by the compositions of associative functions. Yet, we go beyond our predecessors in two ways: (i) we work on recursive functions, instead of imperative loops; (ii) we provide a more general definition of parallel function composition. As we illustrate in Section 2, the key idea behind our findings is the fact that algebraic structures such as groups, monoids and semirings let us decompose recursive functions into simpler components, which are amenable to automatic parallelization.

To validate the ideas discussed in this paper, we have used them to implement a source-to-source compiler, by means of rewriting rules, that performs automatic parallelization of Haskell code. In Section 4 we show how to parallelize six different – and well-known – recursive functions. Key to efficiency is the fact that we can transform elements in the family of parallelizable functions into list homomorphisms. This transformation, which we explain in Section 3, to the best of our knowledge, is novel. Our experiments show that our technique is effective and useful. The parallelized code obtained impressive performance gains over the original purely recursive implementation of the benchmarks, where we have observed improvements of 8.18$\times$ and 65%, on average, for the monoid and semiring-based functions, respectively. The parallel code that we produce shows good scalability when varying the number of threads or the input size. We have achieved speedups of almost up to 3.5$\times$ in a 4-core Intel i5 processor, comparing the parallelized code against its single-threaded counterpart. Furthermore, these parallel implementations present the same asymptotic complexity than their sequential counterparts in five of our benchmarks, and better asymptotic behavior in one of them. These results are even more meaningful if we consider that they have been obtained in a completely automatic way.

This paper extends our previous work [13], closing two years of investigation on the automatic parallelization of particular classes of recursive functions. This new report augments our preliminary work in three ways. First, we present a much deeper discussion of our results. At the time of our original publication, we had observed that some functions would show very little speedup when running in parallel. Careful profiling has revealed that the culprit, in this case, is the garbage collector. Second, we now implement the automatic parallelization of semiring based functions. Before, we had to parallelize those functions via manual transformations. Finally, the extra number of pages allowed us to touch more related work. Thus, Section 5 now provides the reader with a clearer perspective on where our contribution stays, when compared with previous work.

2. This Paper's Ideas in one Example

We will use the well-known factorial function as an example to introduce our ideas to the reader. This function can be defined as follows:

$$f(x) := \begin{cases} 1 & \text{if } x = 1 \\ x \cdot f(x-1) & \text{otherwise} \end{cases}$$

Factorial is a very simple function, and the reader familiar with the parallelization of reductions on commutative and associative operators will know immediately that this function has a very efficient parallel implementation. Key to perform this parallelization is the observation that factorial can be re-written as a sequence of multiplications, e.g.:$f(x) = x \cdot (x-1) \cdot \ldots \cdot 2 \cdot 1$.

A key property of multiplication – associativity – lets us solve them in a pairwise fashion. This possibility gives us the chance to run the above expression in $O(\ln x)$ time. The goal of this paper is to be able to apply this kind of parallelization automatically onto recursive functions. In order to achieve this objective, we shall be re-writing functions as a composition of simpler functions which are associative. In the case of factorial, this composition looks like:

$$f(x) = (f'_{x-1} \circ \ldots \circ f'_2 \circ f'_1)(1)$$

How do we find a suitable implementation of f'_i? We first provide this answer for a family of recursive functions which have the following format:

$$f(p) := \begin{cases} g_0(p) & \text{if } p = p_0 \\ g_1(p) \oplus f(h(p)) \oplus g_2(p) & \text{otherwise} \end{cases} \tag{1}$$

For any function f, with finite recursion, that can be written in the format above, we show that it is possible to decompose f into a composition of functions.

We first identify h, the function used to propagate the arguments of f, which we call the *hop* function. The *hop* function must be a well-defined invertible function. For the factorial example, we have the following *hop* function h: $h(x) = x - 1$, with inverse $h^{-1}(x) = x + 1$.

The *hop* function is fundamental for generating the next arguments of the sequence of compositions. Because the function f has a finite recursion and the *hop* function has an inverse, we can compute $x - 1$, the number of functions which will constitute the sequence of compositions. We find out $x - 1$ after solving the following equation: $p_0 = h^{x-1}(p)$. This equation always holds, because we have a finite recursion, which means that we can always reach the base p_0 after repeatedly applying the *hop* function to p, for a finite number of times. For the factorial example, the depth is exactly the initial argument x. After identifying the *hop* function and its inverse, we re-write f in a manner suitable for the composition of functions which do not contain a recursive call, e.g.: $f'_i(s) \coloneqq (i+1) \cdot s$, where s is the usually called accumulator parameter in funcional composition.

Now, we can write f such as $f(x) = (f'_{x-1} \circ f'_{x-2} \circ \cdots \circ f'_2 \circ f'_1)(1)$. This transformation is useful, considering that functional composition is an associative operation, which can often be parallelized if it is possible to symbolically compute and simplify intermediate compositions [12]. For instance, for $i > 1 \in \mathbb{Z}$, $(f'_i \circ f'_{i-1})(s) = (i+1) \cdot (i \cdot s)$ can be reassociated as $(f'_i \circ f'_{i-1})(s) = ((i+1) \cdot i) \cdot s$ which is essentially equivalent to the original function regarding computational complexity. Thus we can evaluate $f(x)$ by computing a reduction over a list of functions, using the functional composition operator, i.e., $f(x) = \circ/[f'_{x-1}, f'_{x-2}, \ldots, f'_1]$, where $\circ/\ell$ denotes the folding of the operation $\circ$ onto the list ℓ. Reductions with associative operators are a well-known parallel skeleton [4].

Section 3.2 generalizes the factorial example seen in this section. In Section 3.3 we extend our framework a bit further, showing that we can also parallelize functions in the format below. In this case, we consider two operators $\oplus$ and $\odot$, for which we only require that $\odot$ be associative, and $\oplus$ be both commutative and associative. Again, the hop function h must have an inverse function which can be computed efficiently.

$$f(p) \coloneqq \begin{cases} g_0(p) & \text{if } p = p_0 \\ g_1(p) \oplus (g_3(p) \odot f(h(p)) \odot g_4(p)) \oplus g_2(p) & \text{otherwise} \end{cases} \tag{2}$$

3. Automatic Parallelization of Recursive Functions

In this section we formalize the developments earlier seen in Section 2. To this end, we provide a few basic notions in Section 3.1. In Section 3.2, we show how to parallelize functions on the format given by Equation 1. In Section 3.3, we move on to deal with functions defined by Equation 2.

3.1. Technical Background

In the rest of this paper we shall use three notions borrowed from abstract algebra: *groups*, *monoids* and *semirings*. If S is a set, then we define these algebraic structures as follows:

- A group $G = (S, +)$ is a nonempty set closed under a binary-operation $+$, which is associative, invertible and has a zero element $\mathbf{0}$, the identity regarding $+$. For each $a \in S$, its inverse $-a$ also belongs to S. A group need not be commutative. A group $G = (S, +)$ is commutative if and only if $\forall a, b \in S, a + b = b + a$. If a group is commutative, it is usually called an *abelian* group [14].

- A monoid $M = (S, +)$ is a nonempty set closed under an associative binary-operation $+$ with identity $\mathbf{0}$. A monoid need not be commutative and its elements need not have inverses within the monoid.

- A semiring $R = (S, +, \cdot)$ is a nonempty set closed under two associative binary-operations $+$ and $\cdot$, called addition and multiplication, respectively [15, 16]. A semiring satisfies the following conditions:
  - $(S, +)$ is a commutative monoid with identity element $\mathbf{0}$;
  - $(S, \cdot)$ is a monoid with identity element $\mathbf{1}$;
  - Multiplication distributes over addition from either side;
  - Multiplication by $\mathbf{0}$ annihilates R, i.e. $a \cdot \mathbf{0} = \mathbf{0} \cdot a = \mathbf{0}$, for all $a \in S$.

3.1.1. Parallelizing Functional Composition

Functional composition is an associative binary operator over functions. Previous work [12, 17] has shown that, given a family of indexed functions $\mathcal{F}$ closed under functional composition $\circ$, a function $\psi : \mathbb{Z} \times \mathbb{Z} \to \mathcal{F}$ is a composition evaluator of $\mathcal{F}$ iff $i \psi j = f_i \circ f_j$, for $f_i, f_j \in \mathcal{F}$. If each function in $\mathcal{F}$ and the composition evaluator ψ are constant-time computations, then a sequence of n compositions can be efficiently computed in $O(n/p + \ln p)$, considering p processing units. By constant-time, we mean that there is no execution of any element in this chain of compositions that dominates the other asymptotically. A functional composition over $\mathcal{F}$ can be evaluated by using the composition evaluator ψ. Thus, given a sequence of compositions $f_n \circ f_{n-1} \circ \cdots \circ f_1$, this sequence can be efficiently computed using a reduction operator $\circ/[f_n, f_{n-1}, \ldots, f_1]$, since the following equivalence holds: $\circ/[f_n, f_{n-1}, \ldots, f_1] = \psi/[n, n-1, \ldots, 1]$.

3.2. Monoids

In this section, we generalize the solution presented in Section 2. We provide the formal description of the mechanism used for parallelizing the factorial recursive function, regarding general algebraic structures of groups and monoids.

Let S_G and S_M be sets and $M = (S_M, +)$ a monoid. Let $f : S_G \to S_M$ be a recursive function defined as:

$$f(x) \coloneqq \begin{cases} g_0(x_0) & \text{if } x = x_0 \\ g_1(x) + f(h(x)) + g_2(x) & \text{otherwise} \end{cases}$$

We assume that each $g_i : S_G \to S_M$ are pure and non-recursive functions, i.e. if $a \in S_G$ then $g_i(a) \in S_M$, for $i \in [0, 2]$. Furthermore, we assume that the *hop* function $h : S_G \to S_G$ is an invertible function over S_G such that, $\forall x, \exists k \in \mathbb{Z}$ for which $h^k(x) = x_0$, i.e., f is a finite recursive function.

Proposition 1. *The recursive function $f : S_G \to S_M$ can be written as a functional composition.*

Proof. We let $f'_i : S_M \to S_M$ be the following non-recursive function:

$$f'_i(s) = g_1((h^{-1})^i(x_0)) + s + g_2((h^{-1})^i(x_0))$$

In the above definition, we let $h^{-1} : S_G \to S_G$ be the inverse function of the hop h. Function $(h^{-1})^i(x_0)$ is the i-th functional power of $h^{-1} : S_G \to S_G$. To transform the recursive function into a composition, it is important to infer the depth of the recursive stack. Let $k > 0 \in \mathbb{Z}$ such that $h^k(x) = x_0$. Thus $f(x) = (f'_k \circ f'_{k-1} \circ \cdots \circ f'_2 \circ f'_1)(g_0(x_0))$.

The fact that this equality holds follows from simple expansion. At the base case, we have $f(x_0)$. There is no composition; hence, $f(x_0) = g_0(x_0)$. We now consider $f(x) = (f'_k \circ f'_{k-1} \circ \cdots \circ f'_2 \circ f'_1)(g_0(x_0))$. By expanding the right side of the equation, we have:

$$\begin{aligned} f(x) &= (f'_k \circ f'_{k-1} \circ \cdots \circ f'_2 \circ f'_1)(g_0(x_0)) \\ &= g_1((h^{-1})^k(x_0)) + (f'_{k-1} \circ \cdots \circ f'_2 \circ f'_1)(g_0(x_0)) + g_2((h^{-1})^k(x_0) \end{aligned}$$

Because $x = (h^{-1})^k(x_0)$, which follows from $h^k(x) = x_0$, we have that:

$$\begin{aligned} f(x) &= g_1((h^{-1})^k(x_0)) + (f'_{k-1} \circ \cdots \circ f'_2 \circ f'_1)(g_0(x_0)) + g_2((h^{-1})^k(x_0)) \\ &= g_1(x) + (f'_{k-1} \circ \cdots \circ f'_2 \circ f'_1)(g_0(x_0)) + g_2(x) \end{aligned}$$

By expanding the left side of the equation, we have the following:

$$\begin{aligned} f(x) &= g_1(x) + (f'_{k-1} \circ \cdots \circ f'_2 \circ f'_1)(g_0(x_0)) + g_2(x) \Leftrightarrow \\ g_1(x) + f(h(x)) + g_2(x) &= g_1(x) + (f'_{k-1} \circ \cdots \circ f'_2 \circ f'_1)(g_0(x_0)) + g_2(x) \Leftrightarrow \\ f(h(x)) &= (f'_{k-1} \circ \cdots \circ f'_2 \circ f'_1)(g_0(x_0)) \end{aligned}$$

After repeating this process in a similar manner for $k - 1$ times, we have:

$$\begin{aligned} &f(h^{k-1}(x)) = (f'_{k-(k-1)} \circ \cdots \circ f'_2 \circ f'_1)(g_0(x_0)) \Leftrightarrow \\ &f(h^{k-1}(x)) = f'_1(g_0(x_0)) \end{aligned}$$

By expanding both sides, it follows that

$$\begin{aligned}
f(h^{k-1}(x)) &= f_1'(g_0(x_0)) \Leftrightarrow \\
g_1(h^{k-1}(x)) + f(h^k(x)) + g_2(h^{k-1}(x)) &= f_1'(g_0(x_0)) \Leftrightarrow \\
g_1(h^{k-1}(x)) + f(x_0) + g_2(h^{k-1}(x)) &= g_1(h^{-1}(x_0)) + g_0(x_0) + g_2(h^{-1}(x_0)) \Leftrightarrow \\
g_1(h^{k-1}(x)) + g_0(x_0) + g_2(h^{k-1}(x)) &= g_1(h^{-1}(x_0)) + g_0(x_0) + g_2(h^{-1}(x_0))
\end{aligned}$$

Therefore the equality holds, since $h^{k-1}(x) = h^{-1}(x_0)$, which follows directly from $h^k(x) = x_0$. □

From Fisher and Ghuloum [12], we know that the composition of f_i' can be computed in parallel since, for $i > 1 \in \mathbb{Z}$, we have that:

$$\begin{aligned}
&(f_i' \circ f_{i-1}')(s) = g_1((h^{-1})^i(x_0)) + f_{i-1}'(s) + g_2((h^{-1})^i(x_0)) = \\
&g_1((h^{-1})^i(x_0)) + [g_1((h^{-1})^{i-1}(x_0)) + s + g_2((h^{-1})^{i-1}(x_0))] + g_2((h^{-1})^i(x_0)) = \\
&[g_1((h^{-1})^i(x_0)) + g_1((h^{-1})^{i-1}(x_0))] + s + [g_2((h^{-1})^{i-1}(x_0)) + g_2((h^{-1})^i(x_0))]
\end{aligned}$$

3.2.1. Defining the Hop Function

Let $G = (S_G, +)$ be a group with identity $\mathbf{0}$. If the *hop* function h is an invertible, we can calculate k. Let h be generally defined as $h(x) = e_1 + x + e_2$, where $e_1, e_2 \in S_G$. Then

$$\begin{aligned}
h^k(x) &= x_0 \Leftrightarrow \\
e_1 + \cdots + e_1 + e_1 + x + e_2 + e_2 + \cdots + e_2 &= x_0 \Leftrightarrow \\
ke_1 + x + ke_2 &= x_0 \Leftrightarrow \\
x + ke_2 &= (-ke_1) + x_0 \Leftrightarrow \\
x + ke_2 - x_0 &= (-ke_1) \Leftrightarrow \\
x + (ke_2 - x_0 + ke_1) &= \mathbf{0}
\end{aligned}$$

that is, $\exists k \in \mathbb{Z}$ such that $k > 0$ and $(ke_2 - x_0 + ke_1) \in S_G$ is the inverse of $x \in S_G$. Given the equality above, k can often be dynamically computed with a small overhead. If G is commutative, then $(ke_2 - x_0 + ke_1) = k(e_1 + e_2) - x_0$. From these notions, we define function h^{-1}, the inverse of the hopping function, as $h^{-1}(x) = (-e_1) + x + (-e_2)$. This equality is true, since:

$$\begin{aligned}
(h^{-1} \circ h)(x) &= (-e_1) + h(x) + (-e_2) \Leftrightarrow \\
(h^{-1} \circ h)(x) &= (-e_1) + [e_1 + x + e_2] + (-e_2) \Leftrightarrow \\
(h^{-1} \circ h)(x) &= \mathbf{0} + x + \mathbf{0} = x
\end{aligned}$$

3.2.2. Computing the Function Composition using List Homomorphism

Thus far, we have seen how to re-write a function (with certain properties) as a composition of non-recursive functions. We need now a way to implement this composition efficiently. To achieve efficiency, we use list homomorphisms.

We say that a function y is a list homomorphism if we have that $y(w +\!\!+\, z) = y(w) +\!\!+\, y(z)$, where $+\!\!+$ denotes list concatenation. In this section we derive a simple implementation for such a recursive function $f : S_G \to S_M$ by means of list homomorphism.

Proposition 2. *Let $h'(i) = (h^{-1})^i(x_0)$. Then we can write the functional composition $f(x) = (f'_k \circ f'_{k-1} \circ \cdots \circ f'_2 \circ f'_1)(g_0(x_0))$ as:*

$$f(x) = (+/(g_1 \circ h') \star [k, k-1, \ldots, 1]) + g_0(x_0) + (+/(g_2 \circ h') \star [1, 2, \ldots, k]) .$$

We recall that $+/\ell$ denotes the folding of the operation $+$ onto the list ℓ (See Section 2), and $k \star \ell$ denotes the mapping of function k onto every element of ℓ.

Proof. The functional composition expands as follows:

$$\begin{aligned} f(x) &= g'_1(x_0) + g_0(x_0) + g'_2(x_0), \text{where:} \\ g'_1(x_0) &= g_1((h^{-1})^k(x_0)) + g_1((h^{-1})^{k-1}(x_0)) + \cdots + g_1((h^{-1})(x_0)) \\ g'_2(x_0) &= g_2((h^{-1})(x_0)) + g_2((h^{-1})^2(x_0)) + \cdots + g_2((h^{-1})^k(x_0)) \end{aligned}$$

Since $h'(i) = (h^{-1})^i(x_0)$, then we can write:

$$\begin{aligned} g'_1(x_0) &= g_1(h'(k)) + g_1(h'(k-1)) + \cdots + g_1(h'(1)) \\ g'_2(x_0) &= g_2(h'(1)) + g_2(h'(2)) + \cdots + g_2(h'(k)) \end{aligned}$$

Therefore, it is possible to compute g'_1 and g'_2 using a list homomorphism, i.e.:

$$\begin{aligned} g'_1(x_0) &= +/g_1 \star (h' \star [k, k-1, \ldots, 1]) \\ g'_2(x_0) &= +/g_2 \star (h' \star [1, 2, \ldots, k]) \end{aligned}$$

The above expression can then be simplified as:

$$\begin{aligned} g'_1(x_0) &= +/(g_1 \circ h') \star [k, k-1, \ldots, 1] \\ g'_2(x_0) &= +/(g_2 \circ h') \star [1, 2, \ldots, k] \end{aligned}$$

□

3.3. Semirings

We now describe a second family of recursive functions that we can parallelize automatically by rethinking them under the light of algebraic structures. Let S_G and S_R be sets and $R = (S_R, +, \cdot)$ a semiring. Let $f : S_G \to S_R$ be defined as:

$$f(x) \coloneqq \begin{cases} g_0(x_0) & \text{if } x = x_0 \\ g_1(x) + g_3(x) \cdot f(h(x)) \cdot g_4(x) + g_2(x) & \text{otherwise} \end{cases}$$

We let $g_i : S_G \to S_R$ be a non-recursive function. Function $h : S_G \to S_G$ is the *hop*. Let $f'_i : S_R \to S_R$ be the following non-recursive function:

$$f'_i(s) = g_1((h^{-1})^i(x_0)) + g_3((h^{-1})^i(x_0)) \cdot s \cdot g_4((h^{-1})^i(x_0)) + g_2((h^{-1})^i(x_0))$$

We remind the reader that $(h^{-1})^i(x_0)$ is the i-th functional power of h^{-1}, the inverse function of the *hop* function h (see Section 3.2).

In order to transform the recursive function into a composition, again we must infer the depth of the recursive stack. Let $k > 0 \in \mathbb{Z}$ such that $h^k(x) = x_0$ (see Section 3.2). Thus:

$$f(x) = (f'_k \circ f'_{k-1} \circ \cdots \circ f'_2 \circ f'_1)(g_0(x_0))$$

This equality can be proved by a simple expansion, similar to Proposition 1.

Proposition 3. *The recursive function $f : S_G \to R_M$ can be written as a functional composition $(f'_k \circ f'_{k-1} \circ \cdots \circ f'_2 \circ f'_1)(g_0(x_0))$.*

Proof. (Sketch) The proof is similar to the one seen in Preposition 1, and also follows by induction on the number of invocations of the hop function necessary to transform x_0 into x. □

3.3.1. Computing the Function Composition using List Homomorphism

In this section we derive a simple implementation of a recursive function $f : S_G \to S_R$ by means of list homomorphism. In the same spirit of Proposition 2, we can also write functions that follow the semiring pattern as a combination of fold and map over lists:

Proposition 4. *Let $h'(i) = (h^{-1})^i(x_0)$. Then we can write the functional composition $f(x) = (f'_k \circ f'_{k-1} \circ \cdots \circ f'_2 \circ f'_1)(g_0(x_0))$ as a combination of fold and map operations on lists.*

Proof. (Sketch) If $h'(i) = (h^{-1})^i(x_0)$, then we can write:

$$f(x) = \phi_1(k) + \phi_3(k) \cdot g_0(x_0) \cdot \phi_4(k) + \phi_2(k)$$

where $k > 0 \in \mathbb{Z}$ such that $h^k(x) = x_0$ (see Section 3.2). We have that:

$$\begin{aligned}
\beta_1(i) &= (\cdot/(g_3 \circ h') \star [k, k-1, \ldots, i+1]) \, g_1(h'(i)) \, (\cdot/(g_4 \circ h') \star [i+1, \ldots, k]) \\
\phi_1(k) &= g_1(h'(k)) + (+/\beta_1 \star [k-1, k-2, \ldots, 1]) \\
\phi_3(k) &= (\cdot/(g_3 \circ h') \star [k, k-1, \ldots, 1]) \\
\phi_4(k) &= (\cdot/(g_4 \circ h') \star [1, 2, \ldots, k]) \\
\beta_2(i) &= (\cdot/(g_3 \circ h') \star [k, k-1, \ldots, i+1]) \, g_2(h'(i)) \, (\cdot/(g_4 \circ h') \star [i+1, \ldots, k]) \\
\phi_2(k) &= (+/\beta_2 \star [1, 2, \ldots, k-1]) + g_2(h'(k))
\end{aligned}$$

Showing that this inequality holds is a rather tedious process, as it involves a several algebraic expansions; thus, we omit it. Nevertheless, this proof follows the same approach seen in the proof of Proposition 2. □

It is important to notice that the reduction operator evaluates in the identity when the input list is empty. For example, in the definitions of ϕ_1 and ϕ_2, the reduction operators evaluate to $\mathbf{0}$ for $k \leq 1$, namely $(+/\beta_1 \star [k-1, k-2, \ldots, 1])$ and $(+/\beta_2 \star [1, 2, \ldots, k-1])$, as they would result in empty lists.

There are redundant computations in the previous definition. We can optimize the computation of β_1 and β_2 by pre-computing these redundant values using a scan operation. Considering left- and right-associative scan operations (*scanl* and *scanr*), we define lists v and w as follows:

$$\begin{aligned}[v_1, v_2, \ldots, v_{k-1}] &= \mathit{scanr} \cdot /(g_3 \circ h') \star [k, k-1, \ldots, 2] \\ [w_1, w_2, \ldots, w_{k-1}] &= \mathit{scanl} \cdot /(g_4 \circ h') \star [2, 3, \ldots, k]\end{aligned}$$

That is, $v_i = \cdot/(g_3 \circ h') \star [k, k-1, \ldots, k-i+1]$ and $w_i = \cdot/(g_4 \circ h') \star [k-i+1, \ldots, k-1, k]$. Once we have pre-computed v_i and w_i, we can define the following simplified construction:

$$\begin{aligned}\beta_1(i) &= v_i \cdot g_1(h'(i)) \cdot w_{k-i} \\ \phi_1(k) &= g_1(h'(k)) + (+/\beta_1 \star [k-1, k-2, \ldots, 1]) \\ \phi_3(k) &= v_1 \cdot (g_3 \circ h')(1) \\ \phi_4(k) &= (g_4 \circ h')(1) \cdot w_{k-1} \\ \beta_2(i) &= v_i \cdot g_2(h'(i)) \cdot w_{k-i} \\ \phi_2(k) &= (+/\beta_2 \star [1, 2, \ldots, k-1]) + g_2(h'(k))\end{aligned}$$

3.4. Examples

In this section we discuss different functions that we can parallelize automatically. We shall provide examples that we can parallelize using the monoid-based approach (Catalan Numbers and List Concatenation), and with the semiring-based approach (Financial Compound Interest, Horner's Method and Comb Filters). The actual performance of each of these examples is analyzed in Section 4.

Catalan Numbers. Catalan numbers form a sequence of positive integers that appear in the solution of several counting problems in combinatorics, including some generating functions. Catalan numbers are defined as follows:

$$C_n = \frac{2(2n-1)}{n+1} C_{n-1} \quad \text{where } C_1 = 1$$

which can be written as $f : \mathbb{Z} \to \mathbb{Q}$

$$f(x) \coloneqq \begin{cases} 1 & \text{if } x = 1 \\ \frac{2(2x-1)}{x+1} \cdot f(x-1) & \text{otherwise} \end{cases}$$

Similar to the factorial function seen in Section 2, the above function can be written as a composition of non-recursive functions: Let $h : \mathbb{Z} \to \mathbb{Z}$ be $h(x) = x - 1$, $g_1 : \mathbb{Z} \to \mathbb{Q}$ be $g_1(x) = \frac{2(2x-1)}{x+1}$ and $g_2 : \mathbb{Z} \to \mathbb{Q}$ be $g_2(x) = 1$.

Then, we can define a function $f'_i : \mathbb{Q} \to \mathbb{Q}$, such as

$$\begin{aligned}f'_i(s) &= g_1((h^{-1})^i(1)) \cdot s \cdot g_2((h^{-1})^i(1)) \Leftrightarrow \\ f'_i(s) &= g_1(i+1) \cdot s \cdot 1 \Leftrightarrow \\ f'_i(s) &= \frac{2(2i+1)}{i+2} \cdot s\end{aligned}$$

since $h^{-1}(x) = x + 1$. We can easily calculate that $h^k(x) = 1$ for $k = x - 1$, since $x + (k(-1) - 1 + k \cdot 0) = 0 \Leftrightarrow x - k - 1 = 0$.

Therefore, we can parallelize the function $f(x) = (f'_{x-1} \circ f'_{x-2} \circ \cdots \circ f'_2 \circ f'_1)(1)$. In order to apply Proposition 2, we first compute $h'(i) = (h^{-1})^i(1)$ and the composed functions $(g_1 \circ h')$ and $(g_2 \circ h')$. That is

$$\begin{aligned}
h'(i) &= 1 + i \\
(g_1 \circ h')(i) &= \frac{2(2(1+i) - 1)}{(1+i) + 1} \\
&= \frac{2(2i+1)}{i+2} \\
(g_2 \circ h')(i) &= 1
\end{aligned}$$

Afterwards, we can use the computed value of $k = x - 1$ with Proposition 2.

$$\begin{aligned}
f(x) &= (\cdot/(g_1 \circ h') \star [x-1, x-2, \ldots, 1]) \cdot 1 \cdot (\cdot/(g_2 \circ h') \star [1, 2, \ldots, x-1]) \\
&= (\cdot/(g_1 \circ h') \star [x-1, x-2, \ldots, 1])
\end{aligned}$$

List Concatenation. Let $L_{\mathbb{Z}}$ be the set of lists over the set of integers $\mathbb{Z}$. A list is an ordered sequence denoted by $A = [a_1, a_2, \ldots, a_n]$, where the empty list is denoted by $[]$. Concatenation is an associative binary-operation over a set of lists. Given two lists $A = [a_1, a_2, \ldots, a_n]$ and $B = [b_1, b_2, \ldots, b_m]$, we remind the reader that the concatenation of the lists A and B is denoted by $A +\!\!+\, B = [a_1, \ldots, a_n, b_1, \ldots, b_m]$. The identity element regarding concatenation is the empty list. Thus $(L_{\mathbb{Z}}, +\!\!+\,)$ is a non-commutative monoid. Let $f : \mathbb{Z} \to L_{\mathbb{Z}}$ be the following recursive function:

$$f(x) \coloneqq \begin{cases} [] & \text{if } x = 0 \\ [x] +\!\!+\, f(x-1) +\!\!+\, [x] & \text{otherwise} \end{cases}$$

Let $h : \mathbb{Z} \to \mathbb{Z}$ be $h(x) = x - 1$, $g_1 : \mathbb{Z} \to L_{\mathbb{Z}}$ be $g_1(x) = [x]$ and $g_2 : \mathbb{Z} \to L_{\mathbb{Z}}$ be $g_2(x) = [x]$. Then, we can define a simplified function $f'_i : L_{\mathbb{Z}} \to L_{\mathbb{Z}}$, such as:

$$\begin{aligned}
& f'_i(s) = g_1((h^{-1})^i(0)) +\!\!+\, s +\!\!+\, g_2((h^{-1})^i(0)) \Leftrightarrow \\
& f'_i(s) = g_1(i) +\!\!+\, s +\!\!+\, g_2(i) \Leftrightarrow \\
& f'_i(s) = [i] +\!\!+\, s +\!\!+\, [i]
\end{aligned}$$

since $h^{-1}(x) = x + 1$. We have that $h^k(x) = 0$ for $k = x$, since $x + (k(-1) - 0 + k \cdot 0) = 0 \Leftrightarrow x - k = 0$.

Therefore, we can parallelize the function $f(x) = (f'_x \circ f'_{x-1} \circ \cdots \circ f'_2 \circ f'_1)([])$. In order to apply Proposition 2, we first compute $h'(i) = (h^{-1})^i(0)$ and the composed functions $(g_1 \circ h')$ and $(g_2 \circ h')$. That is

$$\begin{aligned}
h'(i) &= i \\
(g_1 \circ h')(i) &= [i] \\
(g_2 \circ h')(i) &= [i]
\end{aligned}$$

Afterwards, we can use the computed value of $k = x$ with Proposition 2.

$$f(x) = (\mathbin{+\!\!+} /(g_1 \circ h') \star [x, x-1, \dots, 1]) \mathbin{+\!\!+} [] \mathbin{+\!\!+} (\mathbin{+\!\!+} /(g_2 \circ h') \star [1, 2, \dots, x])$$

Financial Compound Interest. We can define financial compound interest with periodic deposits recursively. Let $f : \mathbb{Z} \to \mathbb{R}$ be the following recursive function:

$$f(x) \coloneqq \begin{cases} y_0 & \text{if } x = 0 \\ (1+r) \cdot f(x-1) + y_x & \text{otherwise} \end{cases}$$

where y_0 is the initial deposit, r is the compounded rate, and y_x is the deposit on the x-th period. Let $h : \mathbb{Z} \to \mathbb{Z}$ is $h(x) = x - 1$, $g_1 : \mathbb{Z} \to \mathbb{R}$ is $g_1(x) = 0$, $g_3 : \mathbb{Z} \to \mathbb{R}$ is $g_3(x) = (1+r)$, $g_4 : \mathbb{Z} \to \mathbb{R}$ is $g_4(x) = 1$, $g_2 : \mathbb{Z} \to \mathbb{R}$ is $g_2(x) = y_x$. From these notions, we define a simplified function $f'_i : \mathbb{R} \to \mathbb{R}$ as follows:

$$\begin{aligned}
&f'_i(s) = g_1((h^{-1})^i(0)) + g_3((h^{-1})^i(0)) s g_4((h^{-1})^i(0)) + g_2((h^{-1})^i(0)) \Leftrightarrow \\
&f'_i(s) = g_1(i) + g_3(i) s g_4(i) + g_2(i) \Leftrightarrow \\
&f'_i(s) = (1+r)s + y_i \Leftrightarrow
\end{aligned}$$

since $h^{-1}(x) = x + 1$. We have that $h^k(x) = 0$ for $k = x$, since $x + (k(-1) - 0 + k \cdot 0) = 0 \Leftrightarrow x - k = 0$.

Hence, $f(x) = (f'_x \circ f'_{x-1} \circ \cdots \circ f'_2 \circ f'_1)(y_0)$ can be computed in parallel. In order to apply the parallelization proposed in Section 3.3, we first compute h', in addition to the composed functions $(g_1 \circ h')$, $(g_2 \circ h')$, $(g_3 \circ h')$, and $(g_4 \circ h')$. That is

$$\begin{aligned}
h'(i) &= i \\
(g_1 \circ h')(i) &= 0 \\
(g_2 \circ h')(i) &= y_i \\
(g_3 \circ h')(i) &= (1+r) \\
(g_4 \circ h')(i) &= 1
\end{aligned}$$

Afterwards, we can use the computed value of $k = x$ in order to rewrite $f(x)$ as

$$f(x) = \phi_1(x) + \phi_3(x) y_0 \phi_4(x) + \phi_2(x)$$

where

$$\begin{aligned}
\beta_1(i) &= (\cdot/(g_3 \circ h') \star [x, x-1, \dots, i+1])\, 0\, (\cdot/(g_4 \circ h') \star [i+1, \dots, x]) = 0 \\
\phi_1(k) &= 0 + (+/\beta_1 \star [x-1, x-2, \dots, 1]) = 0 \\
\phi_3(k) &= (\cdot/(g_3 \circ h') \star [x, x-1, \dots, 1]) \\
\phi_4(k) &= (\cdot/(g_4 \circ h') \star [1, 2, \dots, x]) = 1 \\
\beta_2(i) &= (\cdot/(g_3 \circ h') \star [x, x-1, \dots, i+1])\, y_i\, (\cdot/(g_4 \circ h') \star [i+1, \dots, x]) \\
&= (\cdot/(g_3 \circ h') \star [x, x-1, \dots, i+1])\, y_i \\
\phi_2(k) &= (+/\beta_2 \star [1, 2, \dots, x-1]) + y_x
\end{aligned}$$

Horner's Method. Horner's method is useful to solve polynomials defined recursively. Its implementation can be parallelized as done in the previous example of financial compound interest. Let c_i be the coefficients, for $0 \leq i \leq n$. Thus a polynomial of degree n can be evaluated, for a given value of x, by the following recursive formula, as described by the Horner's method:

$$f(n) \coloneqq \begin{cases} c_0 & \text{if } n = 0 \\ f(n-1) \cdot x + c_n & \text{otherwise} \end{cases}$$

This recursive function can be transformed in a similar way to the previous example, namely the financial compound interest. Let $h : \mathbb{Z} \to \mathbb{Z}$ is $h(n) = n-1$, $g_1 : \mathbb{Z} \to \mathbb{R}$ is $g_1(n) = 0$, $g_3 : \mathbb{Z} \to \mathbb{R}$ is $g_3(n) = 1$, $g_4 : \mathbb{Z} \to \mathbb{R}$ is $g_4(n) = x$, $g_2 : \mathbb{Z} \to \mathbb{R}$ is $g_2(n) = c_n$. After defining these functions, we can apply the semiring-based transformations using list homomorphism. We first we compute the inverse of h, i.e., $h^{-1}(n) = n + 1$ and the value of k for which $h^k(n) = 0$, which is $k = n$, since $n + (k(-1) - 0 + k \cdot 0) = 0 \Leftrightarrow n - k = 0$. Afterwards, we compute h', in addition to the composed functions $(g_1 \circ h')$, $(g_2 \circ h')$, $(g_3 \circ h')$, and $(g_4 \circ h')$. That is

$$\begin{aligned} h'(i) &= i \\ (g_1 \circ h')(i) &= 0 \\ (g_2 \circ h')(i) &= c_i \\ (g_3 \circ h')(i) &= 1 \\ (g_4 \circ h')(i) &= x \end{aligned}$$

Finally, we can use these functions in order to rewrite $f(n)$ as

$$f(n) = \phi_1(n) + \phi_3(n) c_0 \phi_4(n) + \phi_2(n)$$

where

$$\begin{aligned} \beta_1(i) &= (\cdot/(g_3 \circ h') \star [n, n-1, \ldots, i+1])\, 0\, (\cdot/(g_4 \circ h') \star [i+1, \ldots, n]) = 0 \\ \phi_1(k) &= 0 + (+/\beta_1 \star [n-1, n-2, \ldots, 1]) = 0 \\ \phi_3(k) &= (\cdot/(g_3 \circ h') \star [n, n-1, \ldots, 1]) = 1 \\ \phi_4(k) &= (\cdot/(g_4 \circ h') \star [1, 2, \ldots, n]) \\ \beta_2(i) &= (\cdot/(g_3 \circ h') \star [n, n-1, \ldots, i+1])\, c_i\, (\cdot/(g_4 \circ h') \star [i+1, \ldots, n]) \\ &= c_i\, (\cdot/(g_4 \circ h') \star [i+1, \ldots, n]) \\ \phi_2(k) &= (+/\beta_2 \star [1, 2, \ldots, n-1]) + c_n \end{aligned}$$

Comb Filter in Signal Processing. Comb filters have several applications in signal processing [18]. The following equation represents the feedback form used by comb filters: $y_t = \alpha y_{t-T} + (1-\alpha)x_t$, where x_t is the input signal at a given time t and α controls the intensity that the delayed signal is fed back into the output y_t given a delay time T. Let $f : \mathbb{Z} \to \mathbb{R}$ be the following recursive function:

$$f(t) \coloneqq \begin{cases} y_0 & \text{if } t = 0 \\ \alpha f(t-T) + (1-\alpha)x_t & \text{otherwise} \end{cases}$$

This recursive function can be transformed in a similar way to the previous example, namely the financial compound interest. Let $h : \mathbb{Z} \to \mathbb{Z}$ is $h(t) = t - T$, $g_1 : \mathbb{Z} \to \mathbb{R}$ is $g_1(n) = 0$, $g_3 : \mathbb{Z} \to \mathbb{R}$ is $g_3(n) = \alpha$, $g_4 : \mathbb{Z} \to \mathbb{R}$ is $g_4(n) = 1$, $g_2 : \mathbb{Z} \to \mathbb{R}$ is $g_2(n) = (1-\alpha)x_t$. After defining these functions, we can apply the semiring-based transformations using list homomorphism. We first we compute the inverse of h, i.e., $h^{-1}(t) = t + T$ and the value of k for which $h^k(t) = 0$, which is $k = \frac{n}{T}$, since $t + (k(-T) - 0 + k \cdot 0) = 0 \Leftrightarrow n - Tk = 0$. Afterwards, we compute h', in addition to the composed functions $(g_1 \circ h')$, $(g_2 \circ h')$, $(g_3 \circ h')$, and $(g_4 \circ h')$. That is

$$\begin{aligned} h'(i) &= Ti \\ (g_1 \circ h')(i) &= 0 \\ (g_2 \circ h')(i) &= (1-\alpha)x_{Ti} \\ (g_3 \circ h')(i) &= \alpha \\ (g_4 \circ h')(i) &= 1 \end{aligned}$$

Finally, we can use these functions in order to rewrite $f(n)$ as

$$f(n) = \phi_1(n) + \phi_3(n)y_0\phi_4(n) + \phi_2(n)$$

where

$$\begin{aligned} \beta_1(i) &= (\cdot/(g_3 \circ h') \star [k, k-1, \ldots, i+1])\, 0\, (\cdot/(g_4 \circ h') \star [i+1, \ldots, k]) = 0 \\ \phi_1(k) &= 0 + (+/\beta_1 \star [k-1, k-2, \ldots, 1]) = 0 \\ \phi_3(k) &= (\cdot/(g_3 \circ h') \star [k, k-1, \ldots, 1]) \\ \phi_4(k) &= (\cdot/(g_4 \circ h') \star [1, 2, \ldots, k]) = 1 \\ \beta_2(i) &= (\cdot/(g_3 \circ h') \star [k, k-1, \ldots, i+1])\, (1-\alpha)x_{Ti}\, (\cdot/(g_4 \circ h') \star [i+1, \ldots, k]) \\ &= (\cdot/(g_3 \circ h') \star [k, k-1, \ldots, i+1])\, (1-\alpha)x_{Ti} \\ \phi_2(k) &= (+/\beta_2 \star [1, 2, \ldots, k-1]) + (1-\alpha)x_{Tk} \end{aligned}$$

3.5. Implementation

We have used the ideas discussed in this paper to implement a source-to-source Haskell compiler based on rewriting rules and symbolic computation. Term rewriting systems are usually concerned with computing reduced forms of a given term with respect to a pre-defined set of rules. Several existing compilers also use rewriting rules as an infrastructure for implementing many specific optimizations [19, 20, 21, 22]. Compilers based on rewriting rules check that all term manipulations result in correctly typed terms, before performing the transformation specified by the appropriate rewriting rule that matches with the given term [23].

Our source-to-source compiler uses rewriting rules in order to automatically parallelize recursive functions that match either the monoid-based or the semiring-based transformation rules. We also use symbolic computation to find the inverse of the hop function and also to infer the depth of the recursive stack (as discussed in Sections 3.2 and 3.3). Below we show the source-to-source transformation that we produce for the list concatenation benchmark:

```
# Sequential version:
f :: Integer -> [Integer]
f 0 = []
f n = [n] ++ f(n-1) ++ [n]

# Parallel version:
f_g_1 :: Integer -> [Integer]
f_g_1 i = [i]
f_g_2 :: Integer -> [Integer]
f_g_2 i = [i]
f :: Integer -> [Integer]
f n = let k = n
      in (parFoldr1 (++) (map f_g_1 (reverse [1..k]))) ++ [] ++
         (parFoldr1 (++) (map f_g_2 [1..k]))
```

This sequential code is a direct implementation of the list concatenation example given in Section 3.4. The parallel code was automatically generated by our source-to-source compiler, which, in this case, follow the parallelization described in Section 3.2. It first extracts the terms $g_1(x) = [x]$ and $g_2(x) = [x]$ into separate functions `f_g_1` and `f_g_2`, respectively. The depth of the recursive stack is inferred at compile time, namely `k = n`. Finally, the recursive function is expressed in parallel by means of list homomorphism, as detailed in Section 3.2.

```
parallelize(func):
  type, base, recursion = parseFunction(func)
  checkAllowedTypes(type)
  hop = extractHopExpr(recursion)
  hopK = symbolicComposition(hop, k)
  kMax = solve(hopK, base.arg)
  hopInv = symbolicInversion(hop)
  hopInvI = symbolicComposition(hopInv, i)
  iArg = solve(hopInvI, base.arg)
  rewrite recursion with depth kMax:
    rewrite terms with iArg argument
    if uses semiring-based computation:
      optimize using the scan operator
    for constant terms:
      try to apply constant folding
```

Figure 1: Simplified overview of the algorithm for rewriting recursive functions with their parallel counterparts.

Figure 1 shows a simplified version of the algorithm for rewriting monoid- or semiring-based recursive functions with their parallel counterparts[1]. The source-

[1]The complete implementation of the prototype source-to-source compiler is available at `https://github.com/rcorcs/apref`.

to-source compiler implemented is able to automatically identify the parallelizable functions in a Haskell code, and then automatically rewrite these functions in an equivalent parallel form. After parsing a function definition, the source-to-source compiler verifies whether it is a recursive function with a single recursive call and whether the types are in the set of the allowed types, i.e., types that represent known groups, monoids, or semirings. For the symbolic computations applied to the hop function, we use a *computer algebra system* (CAS)[2]. For the theoretical description of these symbolic computations, we refer to Section 3.2.1. The function call `symbolicComposition(hop, k)` is responsible for symbolically computing $h^k(x)$ from the original hop function $h(x)$. Afterwards, we compute the depth of the recursive stack by solving $h^k(x) = x_0$, in terms of k, with the function call `solve(hopK, base.arg)`. After symbolically computing the inverse of the hop function, h^{-1}, we repeat the same process for solving the composition of the inverse hop function, resulting in the function $h'(i) = (h^{-1})^i(x_0)$, which enables the rewriting of the terms in the compositional form, using list homomorphism. Subsequent transformations consist mainly on straightforward rewriting rules, mostly implemented using symbolic substitution. For semiring-based functions, the scan-based optimization can also be applied, as described in Section 3.3. In addition, the constant folding optimization is able to simplify when the scan operation is applied on a list that contains only the repetition of the same constant term.

The current implementation only accepts hop functions based on the standard numeric types as it is the algebra most widely supported by the CAS. This limitation can be addressed by extending the CAS with other groups in order to enable their use in hop functions [3]. However, we believe that hop functions based on the standard numeric types cover most of the use cases. Moreover, the automatic parallelization is only performed when all symbolic manipulation of the hop function are properly computed. If the CAS is unable to handle the hop function, the transformation falls back to the original recursive implementation.

A more general example of the monoid-based case can be represented by the format below:

```
f :: Integer -> IMGSET
f e_0 = y_0
f n = g_1(n) * f(n-e_1) * g_2(n)
```

The parallel code, automatically generated by the source-to-source compiler, consists of three new functions: `f_g_1`, `f_g_2` and `f`, which have the following general format:

```
f_g_1 :: Integer -> IMGSET
f_g_1 i = g_1((i*e_1 + e_0))
f_g_2 :: Integer -> IMGSET
```

[2]The prototype source-to-source compiler is written in Python and uses the sympy library for symbolic computation.

[3]Implementing new algebras in sympy: `https://github.com/sympy/sympy/wiki/Algebras-in-SymPyCore`.

```
f_g_2 i = g_2((i*e_1 + e_0))
f :: Integer -> IMGSET
f n = let k = ((n - e_0)/e_1)
      in (parFoldr1 (*) (map f_g_1 (reverse [1..k]))) * y_0 *
         (parFoldr1 (*) (map f_g_2 [1..k]))
```

Notice that the hop function can be any expression based on the standard algebras with the numeric types, i.e., we do *not* restrict the hop function to be exactly the expression `n-e_1`. As previously explained, if the CAS is unable to handle the hop function, the original recursive implementation will not be parallelized. In the parallel code, both the `(i*e_1 + e_0)` and the `((n - e_0)/e_1)` expressions are automatically generated by the CAS, by means of symbolic computation. Moreover, the operator `*` represents any binary operator that would form a monoid with `IMGSET`.

Similarly, A general example of the semiring-based case can be represented by the format below:

```
f :: Integer -> IMGSET
f e_0 = y_0
f n = g_1(n) + g_3(n) * f(n-e_1) * g_4(n) + g_2(n)
```

The generated parallel code, without using the scan-based optimization (as described in Section 3.3), has the following format:

```
f_g_1 :: Integer -> IMGSET
f_g_1 i = g_1((i*e_1 + e_0))
f_g_1 :: Integer -> IMGSET
f_g_2 i = g_2((i*e_1 + e_0))
f_g_1 :: Integer -> IMGSET
f_g_3 i = g_3((i*e_1 + e_0))
f_g_1 :: Integer -> IMGSET
f_g_4 i = g_4((i*e_1 + e_0))
f_B_1 k i = (foldr1 (*) (map f_g_3 (reverse [(i+1)..k]))) *
            (f_g_1 i) * (foldr1 (*) (map f_g_4 [(i+1)..k]))
f_B_2 k i = (foldr1 (*) (map f_g_3 (reverse [(i+1)..k]))) *
            (f_g_2 i) * (foldr1 (*) (map f_g_4 [(i+1)..k]))
f_PHI_1 k = (f_g_1 k) +
            (parFoldr1 (+) (map (f_B_1 k) (reverse [1..(k-1)])))
f_PHI_2 k = (parFoldr1 (+) (map (f_B_2 k) [1..(k-1)])) + (f_g_2 k)
f_PHI_3 k = (parFoldr1 (*) (map f_g_3 (reverse [1..k])))
f_PHI_4 k = (parFoldr1 (*) (map f_g_4 [1..k]))
f :: Integer -> IMGSET
f n = let k = ((n - e_0)/e_1))
      in (f_PHI_1 k) + (f_PHI_3 k)*(y_0)*(f_PHI_4 k) + (f_PHI_2 k)
```

However, this parallel code without using the scan-based optimization is not sufficiently efficient, due to many redundant computation in several calls to functions `f_B_1` and `f_B_2`. Because of that, we also implement the scan-based optimization for the recursive functions based on semirings, as described in Section 3.3. The generated parallel code, with the scan-based optimization, has the following format:

```
f_g_1 :: Integer -> IMGSET
f_g_1 i = g_1(i*e_1 + e_0)
f_g_1 :: Integer -> IMGSET
f_g_2 i = g_2(i*e_1 + e_0)
f_g_1 :: Integer -> IMGSET
f_g_3 i = g_3(i*e_1 + e_0)
f_g_1 :: Integer -> IMGSET
f_g_4 i = g_4(i*e_1 + e_0)
f_B_1 v w k i = (v!!(i-1))*(f_g_1 i)*(w!!((k-i)-1))
f_B_2 v w k i = (v!!(i-1))*(f_g_2 i)*(w!!((k-i)-1))
f_PHI_1 v w k = (f_g_1 k) +
                (parFoldr1 (+) (map (f_B_1 v w k) (reverse [1..(k-1)])))
f_PHI_2 v w k = (parFoldr1 (+) (map (f_B_2 v w k) [1..(k-1)])) + (f_g_2 k)
f_PHI_3 v w k = (v!!0) * (f_g_3 1)
f_PHI_4 v w k = (f_g_4 k) * (w!!(k-2))
f :: Integer -> IMGSET
f n = let k = ((n - e_0)/e_1))
          v = scanr1 (*) (map f_g_3 (reverse [2..k]))
          w = scanl1 (*) (map f_g_4 [2..k])
      in (f_PHI_1 v w k)) +
         (f_PHI_3 v w k)*(y_0)*(f_PHI_4 v w k) +
         (f_PHI_2 v w k)
```

While the scan-based optimization is general and can always be performed, there are other optimizations that could be performed on specific cases. One such optimization that can be performed on the specific case of handling numeric terms, with the addition and multiplication operators, is the following: if the functions `f_g_3` and `f_g_4` are constant numeric functions, the scan operations could be avoided by replacing the index accessing with the equivalent symbolic solution for the scan computation, e.g. `(v!!i)` could be replaced with either `(g_3 e_0)*i` or `(g_3 e_0)^i`, if the scan operation is addition or multiplication, respectively. On the case of boolean terms with the operators *or* and *and*, if `f_g_3` and `f_g_4` are constant boolean functions, the index accessing could be replaced directly with the function expression, e.g. `(v!!i)` could be replaced directly with just `(g_3 e_0)`.

In order to generate parallel code in Haskell, we use the parallel library provided by the `Strategies` package, available in the Glasgow Haskell Compiler (GHC) [24, 5, 25]. In particular, we use the following implementation for the parallel fold-right operation.

```
parFoldr1 _ [x] = x
parFoldr1 mappend xs  = (ys ‘par‘ zs) ‘pseq‘ (ys ‘mappend‘ zs) where
  len = length xs
  (ys’, zs’) = splitAt (len ‘div‘ 2) xs
  ys = parFoldr1 mappend ys’
  zs = parFoldr1 mappend zs’
```

The expression `(ys ‘par‘ zs)` *sparks* the evaluation of `ys` and `zs` in parallel, while `‘pseq‘` guarantees that they have been evaluated before evaluating `(ys ‘mappend‘ zs)`. In GHC, *sparks* are not immediately executed, instead, they are queued for execution in FIFO order. The runtime converts a *spark* into a real thread when there is an idle CPU, and then run the new thread on the idle CPU, in such a way that the available parallelism is spread amongst the CPUs. In order to provide load-balancing in the parallel implementation, each processor has a *spark pool* that supports *lock-free work-stealing* [5].

4. Evaluation

In this section, we present our experimental evaluation of the proposed automatic parallelization, in addition to a description of basic aspects of the Glasgow Haskell Compiler (GHC) runtime system and a discussion of its effects on the experimental results. Section 4.2 provides a detailed discussion of our results.

4.1. Experimental Results

To validate the ideas discussed in this paper, we analysed the source-to-source Haskell compiler, implemented as described in Section 3.5, with the benchmarks presented in Section 3.4. The experiments were performed in an Intel i5 quad-core processor, with 3.20 GHz of clock and 16 GiB of RAM. All benchmarks were compiled using the Haskell compiler GHC version 8.2.1, with the following compilation flags: `-O2 -optc-O3 -optc-ffast-math`.

In this section, we show results for six benchmarks – three illustrating monoid-based parallelization (factorial, Catalan numbers and list concatenation), and three illustrating semiring-based parallelization (financial compound interest, Horner's method and comb filter). For the experiments with all six applications, we evaluated their speedup and scalability when varying the number of threads from 1 to 4, while fixing the input argument for each monoid-based and semiring-based benchmark in 100,000 and 10,000, respectively. We consider the average over a total of twenty executions, with an interval of 96% of confidence.

		Factorial	Catalan numbers	List concatenation
Sequential		1.048 / 1	0.837 / 1	4650.016 / 1
Parallel	1	0.110 / 9.527	0.252 / 3.321	0.401 / 11586.418
	2	0.100 / 10.480	0.170 / 4.922	0.337 / 13811.928
	3	0.101 / 10.351	0.144 / 5.821	0.336 / 13839.332
	4	0.099 / 10.506	0.143 / 5.862	0.336 / 13839.332

Table 1: This table shows an analysis of the monoid-based benchmarks, comparing the sequential and the parallel version when varying the number of threads from 1 to 4. It presents the execution time in seconds and the speedup over Sequential (*time (s) / speedup*).

Monoids. Figure 3 and Table 1 present the results for the parallelization of recursive functions based on monoids. While in Table 1 we compare the speedup of the parallelized version over the sequential version, Figure 3 compares the

		Financial comp. interest	Horner's method	Comb filter
Sequential		10.009 / 1	12.342 / 1	4.272 / 1
Parallel	1	19.333 / 0.518	25.432 / 0.485	8.435 / 0.506
	2	10.171 / 0.984	13.410 / 0.920	4.422 / 0.966
	3	7.226 / 1.385	9.545 / 1.293	3.097 / 1.380
	4	5.890 / 1.699	7.921 / 1.558	2.503 / 1.707

Table 2: This table shows an analysis of the semiring-based benchmarks, comparing the sequential and the parallel version when varying the number of threads from 1 to 4. It presents the execution time in seconds and the speedup over Sequential (*time (s) / speedup*).

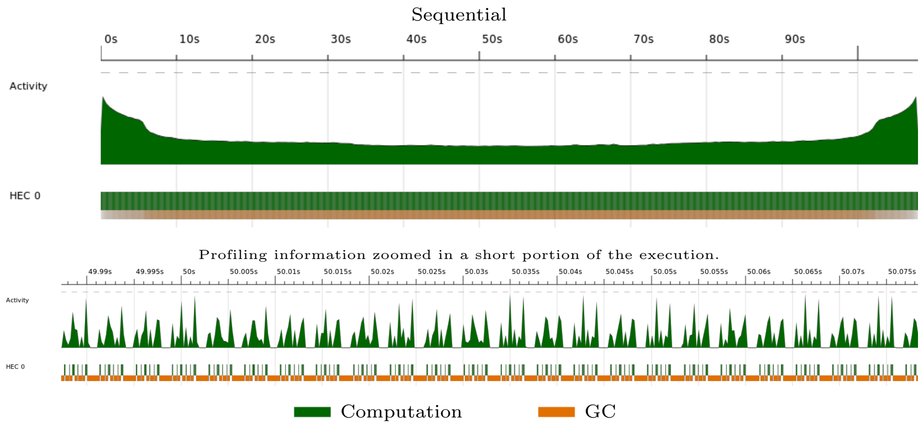

Figure 2: Profiling information for the list concatenation benchmark when concatenating two lists of 20,000 cells. Notice how the GC poses a large overhead in the execution time. About 70% of the execution time is exclusively dedicated to garbage collection.

speedup over the parallel version running with a single thread. For the parallelization of the three monoid-based applications we implemented the construction using the list homomorphism presented in Section 3.2. Parallelization was achieved by means of a right-associative fold operator implemented using the Parallel Haskell library. We achieved a maximum speedup of 1.76×, over the execution with a single thread, in the Catalan benchmark, and an average of 1.35× speedup with four threads for all the monoid-based benchmarks. If we ignore the list concatenation benchmark, the parallelization provides an average improvement of 8.18× over the original purely recursive implementation, with a maximum speedup of 10.51× and a minimum of 5.86×. The original recursive implementation of the list concatenation benchmark has an unreasonably long execution time for very large inputs, such as the one used in our experimental results. Table 1 shows an speedup of almost 14,000 over the original sequential recursive implementation. We discuss, in more detail, the Haskell *garbage collector* (GC) in Section 4.3, but here we look into some pieces of evidence that help to explain the results regarding the list concatenation benchmark. If we compare the number of registered garbage collection events in the profiling information, namely, the event *GC working*, the parallel implementation with four threads records 2,351 instances of such event (with an event log file of about 400 KiB), while the sequential recursive implementation incredibly records 7,232,694 instances of the same event (with an event log file of almost 4 GiB). Although we have not been able to precisely identify the reason for this extremely long execution time, the garbage collector seems to be a major source of overhead. An experiment involving the execution of list concatenation with a smaller input of 20,000 cells (see Figure 2) reveals that garbage collection starts to have a large impact on its performance. A total of 72% of the execution time is spent with the GC, with a total of 134.8 GiB of allocated memory during 107.82 seconds of execution. Figure 5 also sheds some light on the effects of the GC on the execution time of the sequential version. The parallel fold implementation of list concatenation does not suffer these shortcomings. Here, we emphasize that our sequential implementation of list concatenation follows the straightforward description of this algorithm, using the built-in Haskell implementation of lists. These GC overheads may differ with more efficient implementations of arrays or lists, such as the `Seq` data structure provided by the `Data.Sequence` package.

Semirings. Figure 4 and Table 2 present the results for the parallelization of recursive functions based on semirings. Similarly to before, while in Table 2 we compare the speedup of the parallelized version over the sequential version, Figure 4 compares the speedup over the parallel version running with a single thread. For the parallelization of the three semiring-based applications, we have implemented the construction using the list homomorphism discussed in Section 3.3. We have used the same parallel implementation of the right-associative fold operator which we applied on the monoid-based benchmarks. The results with the semiring-based functions suggest that the transformation introduces overheads to the parallelized version, which requires more parallelism to be beneficial, where performance profit over the sequential implementation is only observed with at least three threads. Our highest speedup was 3.37× in the Horner's Method benchmark over the single threaded execution, with an average speedup of 3.29× for the executions with four threads. The parallelization with four threads provides an average improvement 65% over the original recursive implementation.

Scalability. Figure 5 and the scalability of the parallel functions when varying the input argument. In all cases, we can see that, as the input argument increases so does the benefit of using the parallelized version of the recursive functions. For the benchmarks with short execution times, namely Factorial and Catalan numbers, the measurement has more noise due to the resolution of the measurement tool, in addition to the fact that noise from the environment has a larger effect relative to the benchmark's execution time. However, with enough repeated executions, we can see clear statistical performance gains from the automatic parallelization. Tables 3 and 4 present the raw data depicted in Figure 5. These tables show the average, over a total of twenty executions, of the wall-clock time for each input.

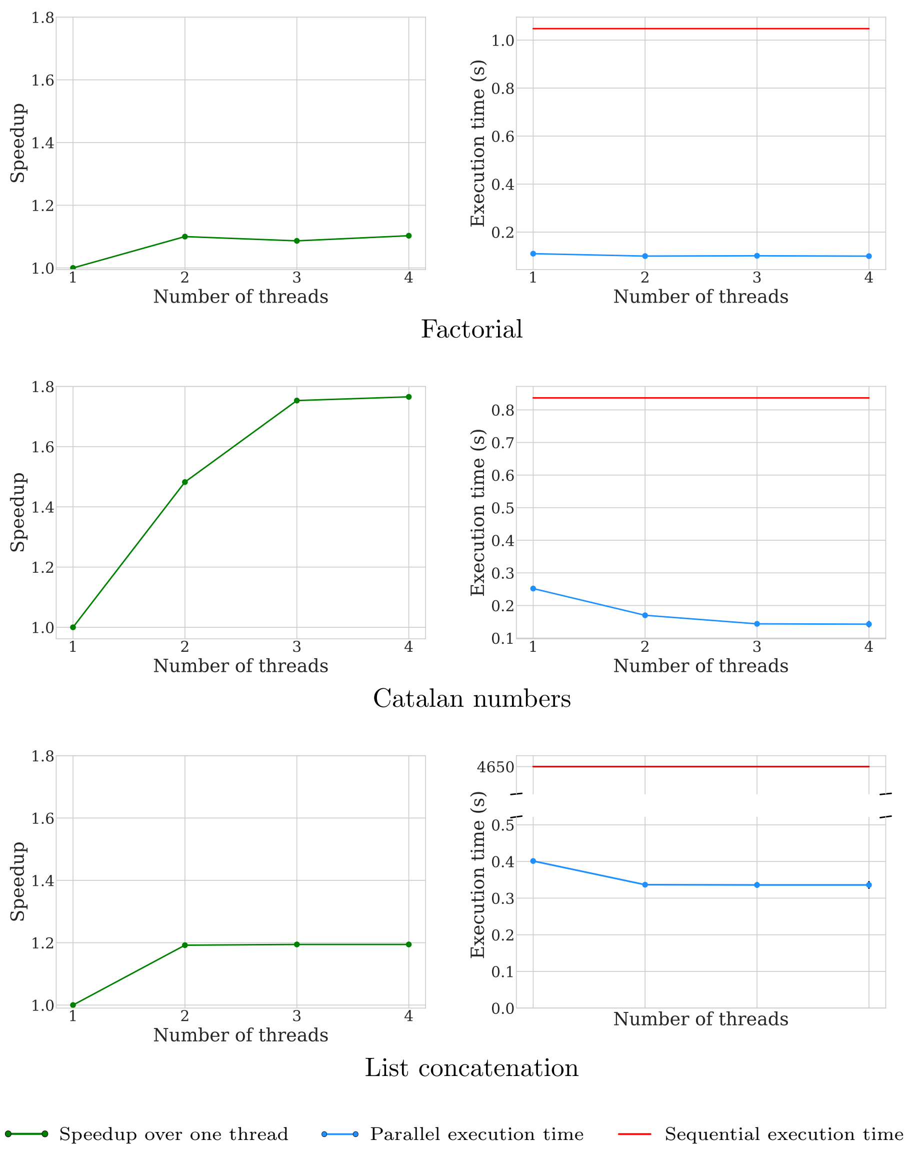

Figure 3: Analysis of the scalability (left) and the execution time (right) of the monoid-based benchmarks with input argument fixed as 100,000. Scalability compares the runtime of the parallel implementations that we generate automatically, running with 1, 2, 3 and 4 threads.

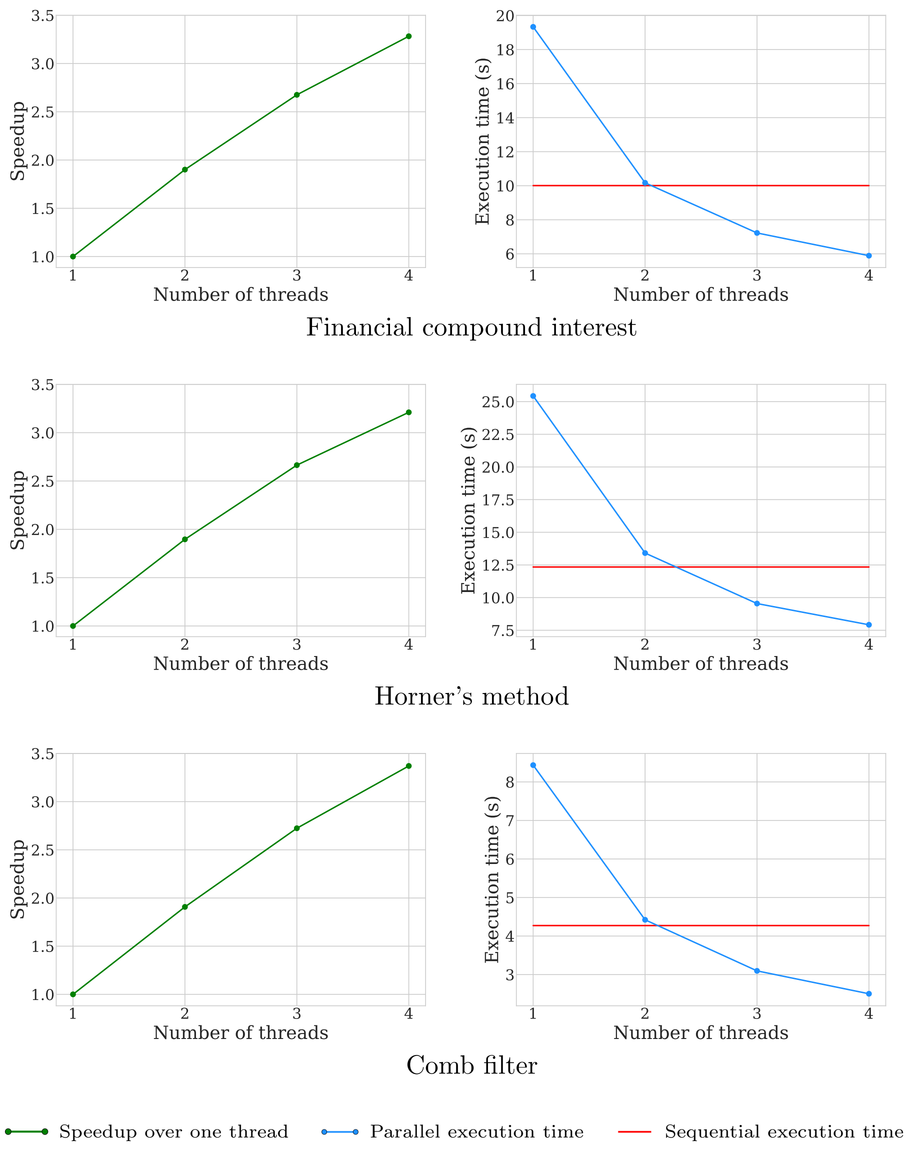

Figure 4: Analysis of the scalability (left) and the execution time (right) of the semiring-based benchmarks with input argument fixed as 10,000.

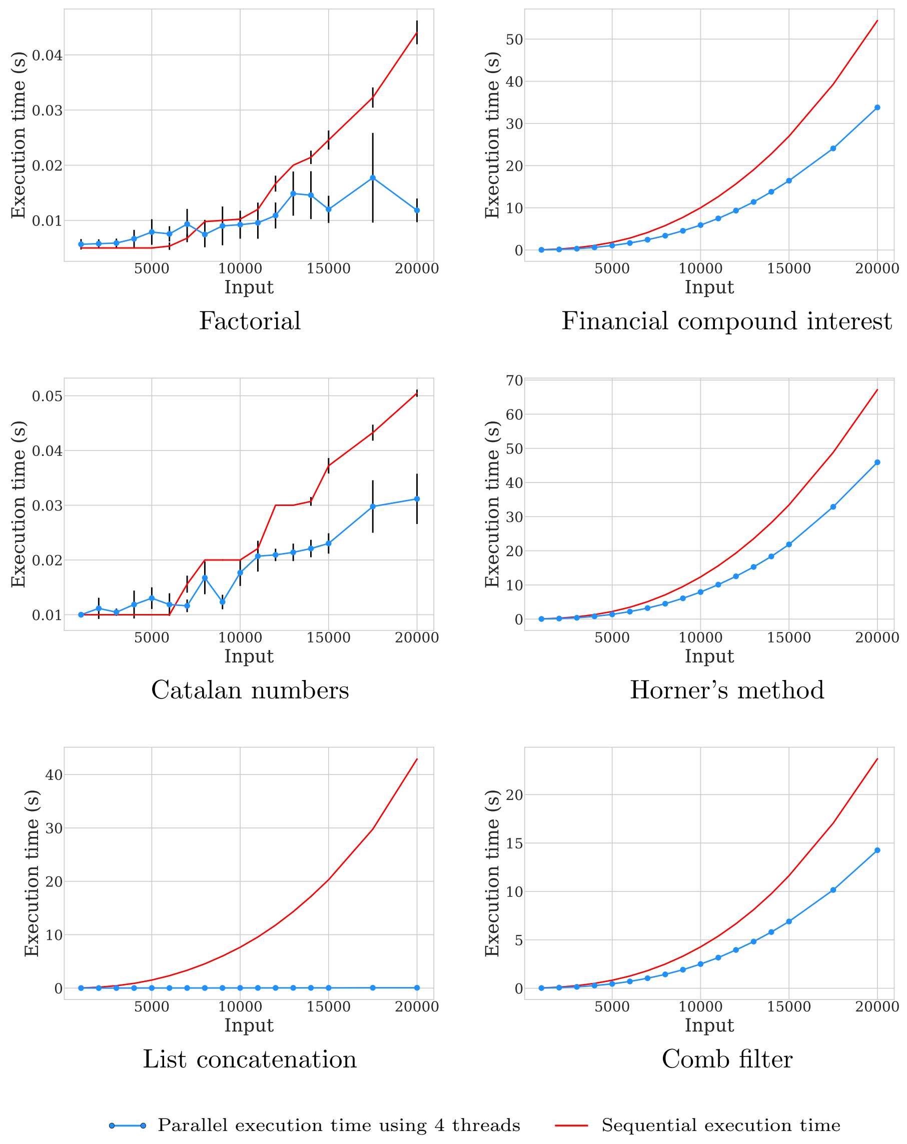

Figure 5: Analysis of the execution time when varying the size of the inputs given to each benchmark. The parallel programs use four threads. Short executions have much more noise in the measurement, hence the very long confidence intervals. For all other cases, the error bar is hidden by the plot line.

	Factorial		Catalan numbers		List concatenation	
Input	Seq.	Parallel	Seq.	Parallel	Seq.	Parallel
1 K	0.010	0.010	0.010	0.010	0.030	0.011
2 K	0.010	0.010	0.010	0.011	0.163	0.011
3 K	0.010	0.010	0.010	0.010	0.448	0.014
4 K	0.010	0.010	0.010	0.012	0.890	0.015
5 K	0.010	0.012	0.010	0.013	1.510	0.021
6 K	0.010	0.011	0.010	0.012	2.327	0.021
7 K	0.010	0.010	0.016	0.012	3.335	0.024
8 K	0.015	0.011	0.020	0.017	4.550	0.023
9 K	0.020	0.011	0.020	0.012	5.999	0.028
10 K	0.020	0.013	0.020	0.018	7.654	0.035
11 K	0.020	0.012	0.022	0.021	9.574	0.036
12 K	0.022	0.014	0.030	0.021	11.788	0.039
13 K	0.030	0.014	0.030	0.021	14.315	0.045
14 K	0.030	0.014	0.031	0.022	17.157	0.048
15 K	0.031	0.017	0.037	0.023	20.297	0.045
17.5 K	0.040	0.018	0.043	0.030	29.765	0.060
20 K	0.050	0.023	0.051	0.031	42.894	0.063

Table 3: Analysis of the wall-clock execution time (in seconds) when varying the size of the inputs given to each of the monoid-based benchmarks. The parallel programs use four threads.

	Financial comp. interest		Horner's method		Comb filter	
Input	Seq.	Parallel	Seq.	Parallel	Seq.	Parallel
1 K	0.055	0.035	0.065	0.044	0.034	0.023
2 K	0.222	0.135	0.267	0.161	0.114	0.068
3 K	0.550	0.308	0.663	0.392	0.266	0.146
4 K	1.067	0.615	1.300	0.805	0.501	0.279
5 K	1.816	1.064	2.221	1.384	0.827	0.453
6 K	2.824	1.652	3.472	2.194	1.263	0.704
7 K	4.148	2.434	5.093	3.221	1.816	1.037
8 K	5.781	3.389	7.106	4.504	2.496	1.433
9 K	7.731	4.553	9.527	6.087	3.310	1.915
10 K	10.008	5.903	12.341	7.924	4.272	2.502
11 K	12.647	7.480	15.609	10.093	5.385	3.166
12 K	15.655	9.334	19.329	12.524	6.665	3.959
13 K	19.023	11.389	23.545	15.272	8.116	4.823
14 K	22.793	13.795	28.233	18.351	9.765	5.807
15 K	26.969	16.428	33.433	21.862	11.624	6.897
17.5 K	39.282	24.079	48.810	32.877	17.054	10.142
20 K	54.392	33.799	67.189	45.926	23.704	14.256

Table 4: Analysis of the wall-clock execution time when varying the size of the inputs given to each of the semiring-based benchmarks. The parallel programs use four threads.

4.2. Discussion and Threats to Validity

On the Quality of Our Sequential Implementations. In the monoid-based cases, even the transformed code executed with a single thread can present better performance than the original purely recursive implementation. This performance gain can be attributed to optimizations that can take better advantage of fold-based implementations, such as deforestation [26, 27]. We would like to emphasize that the sequential programs are naïve implementations of well-known algorithms. No attempt to optimize them has been made. Those functions work only as starting points, from where we can derive parallel code – they have not been conceived to run fast.

On our Speedups. We have been able to observe actual speedups on the six benchmarks that we have played with. These speedups were usually sublinear, e.g., we could not observe a four-fold speedup in any of the cases. We believe that this asymptotic behavior is due to the overhead imposed by the reduction operator required by the proposed parallel construction. Reduction, as we use it, limits our speedups by a logarithmic factor. In other words, in a hypothetical Parallel Random-Access Machine (PRAM), we would still be limited by the complexity of carrying out a parallel reduction with a commutative and associative binary operator. Nevertheless, we would like to emphasize that all these results have been obtained by means of automatic transformations. In other words, the use of our techniques does not require any intervention from the programmer who has implemented the original version of each function that we parallelize.

Two benchmarks, namely, factorial and list concatenation, have presented limited scalability when varying the number of threads. We suggest two main factors that prevented better scalability: (*i*) both benchmarks produce reasonably large output, such that writing the output is a sequential portion that represents a significant percentage of the total execution time; (*ii*) garbage collection is implemented in a stop-the-world fashion. Concerning the first point, factorial and list concatenation produce reasonably large outputs, e.g. the factorial of 100,000 is a number of 456,575 digits and the output for the list concatenation is a string with 477,789 characters. In both cases, writing the output takes on average about 0.05 seconds, which represents a significant percentage of the total execution time, e.g. it is about half the execution time for the factorial benchmark. In Figure 6, the sequential portion during the second half of the execution mainly represents the processing required for writing the output. We believe that the last point – garbage collection – is sufficiently important to deserve a deeper analysis. Such analysis is the subject of the rest of this section.

4.3. The Impact of Garbage Collection in our Results

The GHC Runtime System. The Glasgow Haskell Compiler (GHC) [24, 25] runtime system consists of multiple *Haskell Execution Contexts* (HECs), which are virtual processors responsible for executing Haskell threads, i.e., the available *sparks* queued for execution. The runtime system maintains exactly one HEC for each CPU. Normally, HECs execute without any synchronization, locks, or atomic instructions. Synchronization is only required for load balancing, garbage collection, and other similar management tasks [28].

GHC implements a garbage collector (GC) that is a parallel, generational, copying collector [28], which means that it has a multi-threaded implementation, as opposed to a GC that runs concurrently with the program. Originally, the GC only runs when all HECs have stopped and agreed to garbage collect, in a *stop-the-world* fashion [5]. More recently [29], the current GHC runtime system implements a thread-local garbage collection, which is able to independently collect allocated space in the local heap of each thread, while the shared global heap still requires that all processors synchronize and cooperate in a parallel global garbage collection.

Profiling the Effects of Garbage Collection on Performance. In Figures 6, 7, 8, 9, and 10, the green area represents actual Haskell computation and the orange area represents garbage collection activity. At the top of each figure, the combined activity of all HECs are illustrated in a single curve, considering only the actual Haskell computation of the benchmark. Those figures also show a detailed view of the activity in each HEC at any given time during the execution of the benchmarks, where it differs Haskell computation in green and garbage collection in orange. HECs can also be idle, a state that is illustrated in white in our figures. We notice many occurrences of global garbage collection, where some HECs may be idle waiting for the GC to finish. We also notice that, in same cases, a thread-local garbage collection takes place, where at the same instant one HEC is performing Haskell computation while another HEC is performing garbage collection in its local heap [29].

From the profiling information shown in Figure 6 and Figure 8, we can conclude that the garbage collector poses a significant overhead onto factorial and list concatenation. For the factorial benchmark, the percentage of time spent with garbage collection varied from 24.4% (with a single thread) to 29.8% (with four threads), with the maximum pause for garbage collection varying from 0.0026 seconds to 0.0150 seconds. The synchronization during garbage collection, in a stop-the-world fashion, prevents the application to scale when increasing the number of threads.

Garbage collection represents a major portion of all the execution time in list concatenation, as shown in Figure 8. This benchmark has an allocation rate of more than 5 GiB/s, in respect only of the time spent in actual Haskell computation, allocating a total of about 750 MiB. Because of this particular aspect, the percentage of time spent with garbage collection is more than 65% of the total execution time. Increasing the number of threads does not affect much the garbage collection overhead, since the synchronization overheads are shadowed by performance gains from the parallel garbage collection, which is particularly beneficial for the list concatenation benchmark and its massive allocation requirements. However, the parallelizable portion of the actual application represents a very small percentage of the total execution time.

Because the Catalan benchmark has a smaller output, and a smaller mem-

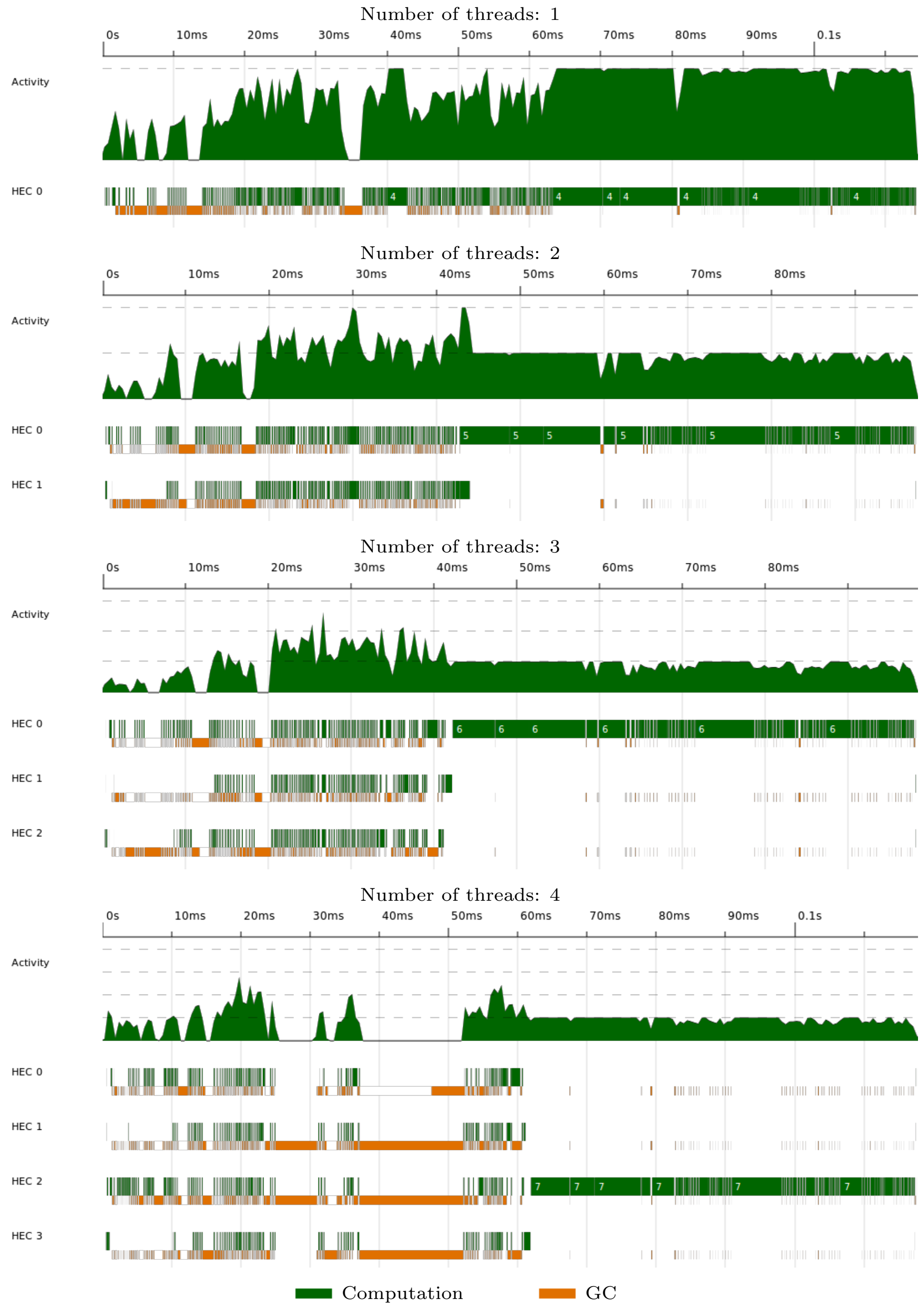

Figure 6: Profiling information for the Factorial benchmark. Notice how the second half of the execution is a uniquely sequential process, in this case, dedicated to writing the large output.

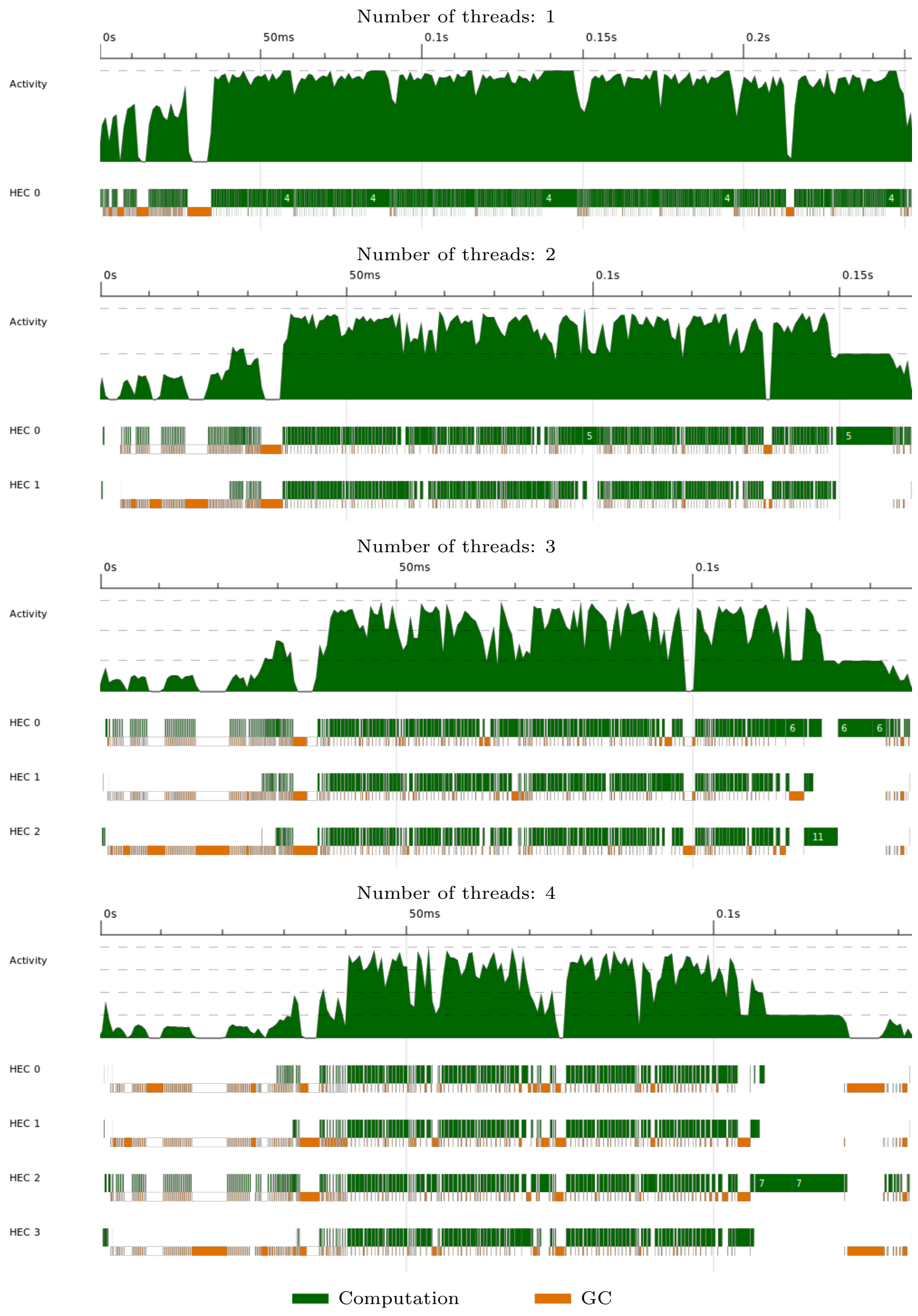

Figure 7: Profiling information for the Catalan benchmark. This benchmark has a reasonably large parallelizable portion and a fairly low garbage collection overhead.

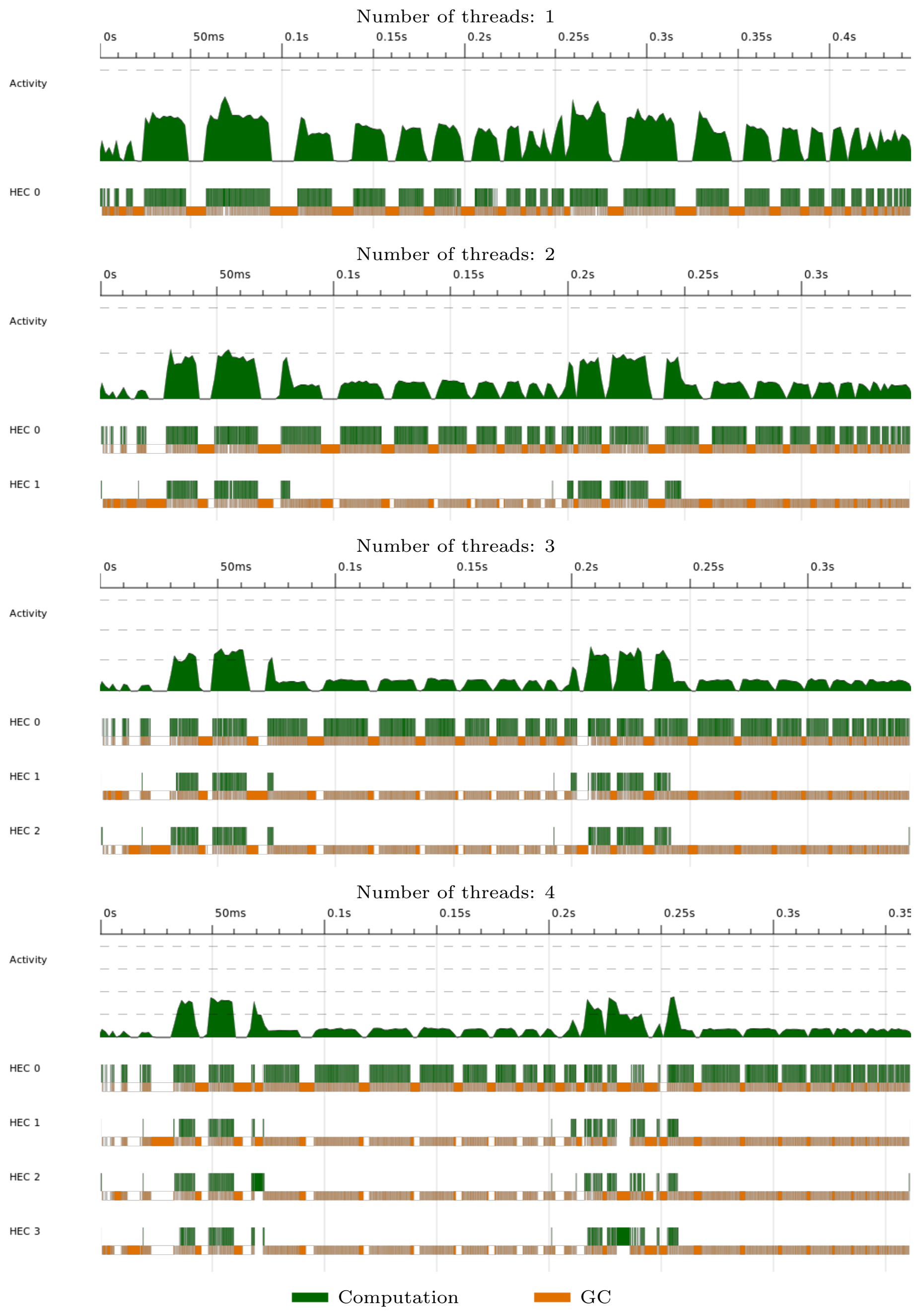

Figure 8: Profiling information for the List concatenation benchmark. The percentage of time spent in garbage collection represents more than 65% of the total execution time of this benchmark, posing a large overhead during its execution.

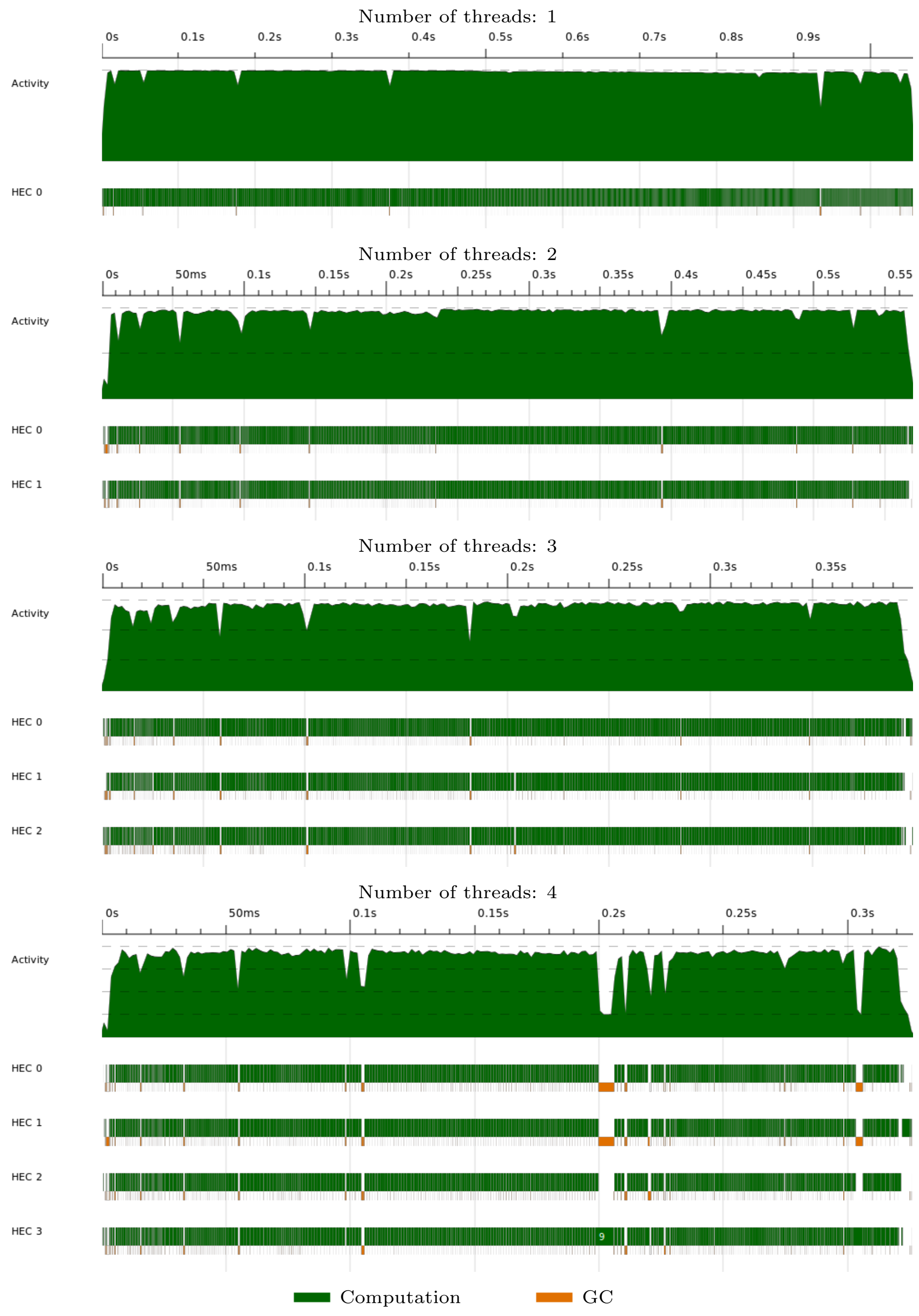

Figure 9: Profiling information for the Horner's method benchmark. Notice how the runtime system is able to leverage parallelism, with very little synchronization overhead as the number of threads increases, benefiting from load-balancing by the GHC runtime system.

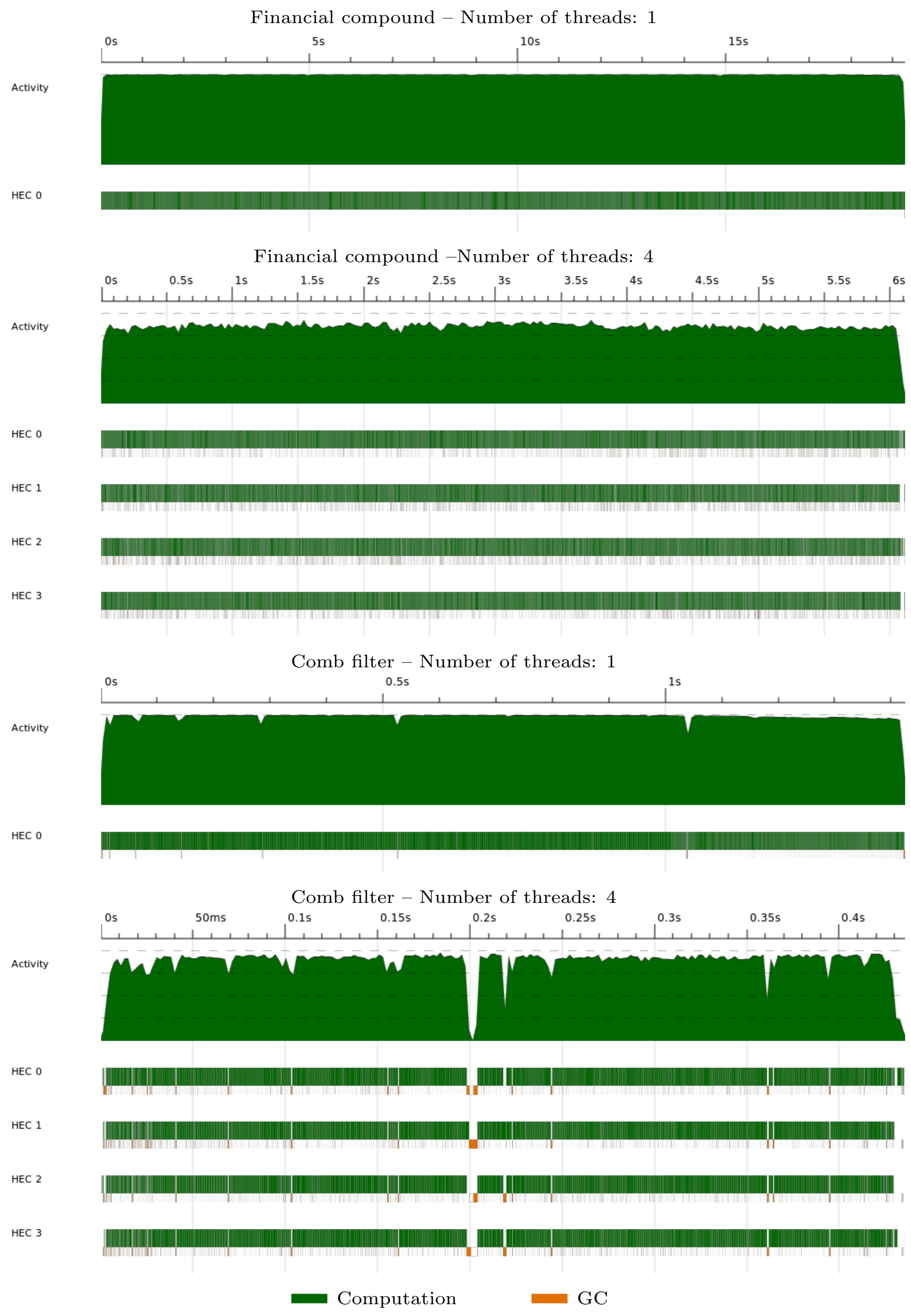

Figure 10: Profiling information for the Financial and the Comb filter benchmarks. Because these benchmarks behave similarly to Horner's method, we only show executions with 1 and 4 threads.

ory footprint, it benefits more from parallelism, when compared to the other two monoid-based benchmarks (see Figure 7). Due to its reasonable parallelizable portion and smaller garbage collection overhead, the Catalan benchmark presents good scalability, with a maximum speedup of 1.92× with four threads, improving 6.69× over the original purely recursive implementation.

All three semiring-based benchmarks have a similar runtime behavior, as illustrated in Figures 9 and 10. In those three cases, the sequential portion that writes the output is negligible compared to the parallelizable portion of the computation. Moreover, the garbage collection also poses a negligible overhead during runtime. Given the combination of these well behaved runtime aspects, all three benchmarks present good scalability.

5. Related Work

One of the perceived advantages of functional programming languages has always been the facility in which programs written in these languages could be parallelized. Advocates of parallelism in pure functional programming claim that programmers should not be concerned about how to parallelize their code – such task should be given to the compiler. Thus, historically, automatic parallelization has been part of the design of many purely functional programming languages [30]. Yet, we believe that an old statement, by Carriero and Gelernter is still valid: "This capability [of compilers] remains to be demonstrated, and achieving it automatically and generally strikes us as a difficult problem" [31]. This paper has presented a technique to address some of the shortcomings that Carriero and Gelernter have raised. Like us, several different researchers have proposed techniques to support the automatic parallelization of functional code.

Map-Reduce Patterns. There are several automatic parallelization techniques that, similarly to ours, seek common patterns in code. Perhaps the most widespread parallel patterns are map and reduce [32]. Map is inherently parallel, and runs in $O(1)$ on the PRAM world. Reduce can be parallelized up to $O(\ln n)$ time under that model. Thus, it comes as no surprise that both patterns can be discovered automatically, and subsequently transformed to run in parallel [10]. This kind of transformation is possible even in non-functional programs, as Mata *et al.* have demonstrated on C [33], or Morais *et al.* have demonstrated on Dinamica EGO [34, 35].

Polyhedral Patterns. Strategies based on matrix-multiplication are another well-known example of mining of parallel pattern. Kogge and Stone [36] have shown how to parallelize a recurrence equation by rewriting it in a form of matrix multiplication, also called *state-vector update* form. An expression e in a loop is a recurring expression if, and only if, e is computed from some loop-carried value. Sato and Iwasaki [37] have described a framework based on matrix multiplication for automatically parallelizing affine loops that consist of reduce or scan operations. They have also provided algorithms for recognizing the normal form and max-operators automatically. They have been able to report considerable speedups and high scalability by applying their framework onto simple benchmarks. Also along this line, Zou and Rajopadhye [38] have proposed a way to parallelize scan operations using the matrix multiplication framework with the polyhedral model [39, 40, 41]. They can handle arbitrary nested affine loops; the polyhedron model itself has already been used to parallelize different types of loops in imperative programming languages [42]. Contrary to our work, these previous approaches search for a way to deconstruct a loop as multiplication of matrices, we search for a way to deconstruct a function as a composition of monoid/semiring operations. The programs that can be parallelized by these two approaches are different.

Rewriting Rules. Rewriting rules represent a well-known optimization technique that gives programmers the ability to interact with the compiler. Such rules let the programmer specify relations that, once applied onto the program, yield better code [19]. This technique has been used with the goal to generate parallel code out of programs written in a sequential mindset. Steuwer et al. [20] use rewriting rules to perform source-to-source transformations, translating high-level functional expressions into a low-level functional representation close to OpenCL. They use a pre-defined set of rewriting rules in order to automatically search for an efficient device-specific OpenCL code. Although they provide good portable performance results [20, 21], their current implementation does not allow for automatically parallelizing recurrence expressions.

Composition Patterns. Fisher and Ghuloum [12] provide a generalized formalization for automatic parallelization of loops by extracting function composition as the main associative operator. If a function is closed under composition, its compositions can be computed efficiently. They describe loops that compute reduction or scan as the composition of its *modeling function.* For loops that fit the allowed format, they can be implemented in a manner that computes the composition of the modelling function in parallel. Our work improves on theirs, because we extend their approach to recursive functions. In fact, this is our main contribution: a general way to extract parallelism buried under the syntax of potentially convoluted recursive implementations. In addition, we also provide a more general definition of parallel code by means of algebraic structures such as monoids and semirings.

List Homomorphism Patterns. There exists vast literature about list homomorphisms [8, 9, 10, 11]. List homomorphism is a special class of natural recursive functions on lists, which has algebraic properties suitable for parallelism. These properties can also be extended to other data-structures, such as trees, as demonstrated by Morihata *et al* [43]. We rely on list homomorphisms to build efficient parallel computations of recursive functions. However, our approach does not target exclusively functions that work on lists. As many of our examples illustrate, we are able to generate parallel code for functions involving just primitive types, or even for more complex data-structures that can be processed by monoid-based operators.

6. Conclusion

This paper has presented a theoretical approach to parallelize linear recursive functions. This contribution is important because previous work have reported difficulties to infer parallel behavior out of recursive functions. In this case, parallelism is usually buried under heavy and convoluted syntax. We have delineated two classes of linear recursive functions which we can parallelize automatically. These functions are defined in terms of algebraic constructions, namely groups, monoids and semirings. By leveraging from these algebraic constructions we are able to rewrite the recursive functions in a fold-based parallel form using list-homomorphism. This transformation presents good scalability when increasing the number of threads and input size. Our experimental evaluation shows improvements of 8.18× and 65%, on average, for the monoid and semiring-based functions, respectively. Moreover, in some cases, the transformed code provides better performance than the original recursive code even when executed with a single thread, due to optimizations that can take better advantage of fold-based implementations.

In this paper, we do not address recursive functions with multiple recursive calls, such as the classic Fibonacci function or the MergeSort algorithm, because these cases have a trivial parallelization strategy that consists in executing each independent recursive call in parallel. For example, the Fibonacci function can be easily parallelized as shown below:

```
# Sequential version:
fib n | n <= 1    = n
      | otherwise = fib (n-1) + fib (n-2)

# Parallel version:
fib n | n <= 1    = n
      | otherwise = n1 ‘par‘ n2 ‘pseq‘ (n1+n2)
        where n1 = fib (n-1); n2 = fib (n-2)
```

An interesting work on this direction would be to develop an automatic granularity coarsening strategy in order to spawn only non-trivial computation, otherwise the cost of spawning it in parallel overshadows the benefits obtained by running it in parallel.

As future work, we intend to broaden the classes of recursive functions that our algebraic framework can automatically parallelize. One such extension is to handle recursive functions with multiple arguments. Similar to the constant folding and the scan-based optimization, we also intend to perform optimizations on the parallel code generated by our source-to-source compiler. For example, the scan operations could be performed in parallel and also using the efficient implementation of vectors provided by the `Data.Vector` library, which allows for a more efficient indexing. Another important future contribution would be to work on lowering the overhead posed by the garbage collector in parallel Haskell programs.

Acknowledgments

This work was partially supported by the UK Engineering and Physical Sciences Research Council (EPSRC) under grants EP/L01503X/1 for the University of Edinburgh, School of Informatics, Centre for Doctoral Training in Pervasive Parallelism, and also by the Institute for Computing Systems Architecture (ICSA) in the School of Informatics at the University of Edinburgh. Fernando Pereira and Luís Fabrício W. Góes are supported by grants from CAPES, FAPEMIG and CNPq.

References

[1] R. Govindarajan, J. Anantpur, Runtime dependence computation and execution of loops on heterogeneous systems, in: Proceedings of the 2013 IEEE/ACM International Symposium on Code Generation and Optimization, 2013, pp. 31:1–31:10.

[2] S. Misailovic, D. Kim, M. Rinard, Parallelizing sequential programs with statistical accuracy tests, ACM Transactions on Embedded Computing Systems 12 (2) (2013) 88:1–88:26.

[3] Z. Wang, G. Tournavitis, B. Franke, M. F. P. O'Boyle, Integrating profile-driven parallelism detection and machine-learning-based mapping, ACM Transactions on Architecture Code Optimization 11 (1) (2014) 2:1–2:26.

[4] K. Hammond, J. Berthold, R. Loogen, Automatic skeletons in template haskell, Parallel Processing Letters 13 (03) (2003) 413–424.

[5] S. Marlow, S. Peyton Jones, S. Singh, Runtime support for multicore haskell, in: Proceedings of the 14th ACM SIGPLAN International Conference on Functional Programming, ACM, 2009, pp. 65–78.

[6] C. Brown, M. Danelutto, K. Hammond, P. Kilpatrick, A. Elliott, Cost-directed refactoring for parallel erlang programs, International Journal of Parallel Programming 42 (4) (2013) 564–582.

[7] A. Collins, D. Grewe, V. Grover, S. Lee, A. Susnea, NOVA: A functional language for data parallelism, in: Proceedings of the 2014 ACM SIGPLAN International Workshop on Libraries, Languages, and Compilers for Array Programming, 2014, pp. 8–13.

[8] M. Cole, Parallel programming with list homomorphisms, Parallel Processing Letters 5 (02) (1995) 191–203.

[9] Z. Hu, H. Iwasaki, M. Takeichi, Formal derivation of efficient parallel programs by construction of list homomorphisms, ACM Transactions on Programming Languages and Systems 19 (3) (1997) 444–461.

[10] K. Morita, A. Morihata, K. Matsuzaki, Z. Hu, M. Takeichi, Automatic inversion generates divide-and-conquer parallel programs, in: Proceedings of the ACM SIGPLAN 2007 Conference on Programming Language Design and Implementation, 2007, pp. 146–155.

[11] Y. Liu, Z. Hu, K. Matsuzaki, Towards systematic parallel programming over mapreduce, in: Proceedings of the 17th International Conference on Parallel Processing, 2011, pp. 39–50.

[12] A. L. Fisher, A. M. Ghuloum, Parallelizing complex scans and reductions, in: Proceedings of the ACM SIGPLAN 1994 Conference on Programming Language Design and Implementation (PLDI), 1994, pp. 135–146.

[13] R. C. O. Rocha, L. F. W. Góes, F. M. Q. Pereira, An algebraic framework for parallelizing recurrence in functional programming, in: Proceedings of the 20th Brazilian Symposium on Programming Languages, Springer, 2016, pp. 140–155.

[14] J. J. Rotman, Advanced Modern Algebra, 2nd Edition, Prentice Hall, 2003.

[15] J. S. Golan, Semirings and their Applications, 1st Edition, Springer, 1999.

[16] J. S. Golan, Power Algebras over Semirings: With Applications in Mathematics and Computer Science, 1st Edition, Mathematics and Its Applications 488, Springer, 1999.

[17] A. Morihata, K. Matsuzaki, Automatic parallelization of recursive functions using quantifier elimination, in: Proceedings of the 10th International Symposium on Functional and Logic Programming, 2010, pp. 321–336.

[18] S. J. Schlecht, E. A. P. Habets, Connections between parallel and serial combinations of comb filters and feedback delay networks, in: Proceedings of the 2012 International Workshop on Acoustic Signal Enhancement, 2012, pp. 1–4.

[19] S. P. Jones, A. Tolmach, T. Hoare, Playing by the rules: rewriting as a practical optimisation technique in ghc, in: Haskell workshop, Vol. 1, 2001, pp. 203–233.

[20] M. Steuwer, C. Fensch, S. Lindley, C. Dubach, Generating performance portable code using rewrite rules: From high-level functional expressions to high-performance OpenCL code, in: Proceedings of the 20th ACM SIGPLAN International Conference on Functional Programming, 2015, pp. 205–217.

[21] T. Remmelg, T. Lutz, M. Steuwer, C. Dubach, Performance portable GPU code generation for matrix multiplication, in: Proceedings of the 9th Annual Workshop on General Purpose Processing Using Graphics Processing Unit, 2016, pp. 22–31.

[22] M. Steuwer, T. Remmelg, C. Dubach, Lift: A functional data-parallel IR for high-performance GPU code generation, in: Proceedings of the 2017 International Symposium on Code Generation and Optimization, CGO '17, IEEE Press, Piscataway, NJ, USA, 2017, pp. 74–85.

[23] E. Balland, P. Brauner, R. Kopetz, P. Moreau, A. Reilles, Tom: Piggybacking rewriting on java, in: Proceedings of the 18th International Conference on Term Rewriting and Applications, 2007, pp. 36–47.

[24] J. Berthold, S. Marlow, K. Hammond, A. A. Zain, Comparing and optimising parallel haskell implementations for multicore machines, in: Proceedings of the International Conference on Parallel Processing Workshops, 2009, pp. 386–393.

[25] S. Marlow, P. Maier, H. Loidl, M. Aswad, P. W. Trinder, Seq no more: better strategies for parallel haskell, in: Proceedings of the 3rd ACM SIGPLAN Symposium on Haskell, 2010, pp. 91–102.

[26] P. Wadler, Deforestation: Transforming programs to eliminate trees, Theoretical Computer Science 73 (2) (1988) 231–248.

[27] A. J. Gill, S. L. P. Jones, Cheap deforestation in practice: An optimizer for haskell., in: Proceedings of the 13th IFIP World Computer Congress, 1994, pp. 581–586.

[28] S. Marlow, T. Harris, R. P. James, S. Peyton Jones, Parallel generational-copying garbage collection with a block-structured heap, in: Proceedings of the 7th ACM International Symposium on Memory Management, 2008, pp. 11–20.

[29] S. Marlow, S. Peyton Jones, Multicore garbage collection with local heaps, in: Proceedings of the 2011 ACM International Symposium on Memory Management, ACM, 2011, pp. 21–32.

[30] M. C. Chen, A parallel language and its compilation to multiprocessor machines or VLSI, in: Proceedings of the 13th Annual ACM Symposium on Principles of Programming Languages, 1986, pp. 131–139.

[31] N. Carriero, D. Gelernter, Linda in context, ACM Communications 32 (4) (1989) 444–458.

[32] J. Dean, S. Ghemawat, MapReduce: Simplified data processing on large clusters, ACM Communications 51 (1) (2008) 107–113.

[33] L. L. P. Da Mata, F. M. Q. a. Pereira, R. Ferreira, Automatic parallelization of canonical loops, Science of Computer Programming 78 (8) (2013) 1193–1206.

[34] B. M. Ferreira, F. M. Q. Pereira, H. Rodrigues, B. S. Soares-Filho, Optimizing a geomodeling domain specific language, in: Proceedings of the 16th Brazilian Symposium on Programming Languages, 2012, pp. 87–101.

[35] B. M. Ferreira, B. S. Soares-Filho, F. M. Q. Pereira, The dinamica virtual machine for geosciences, in: Proceedings of the 19th Brazilian Symposium on Programming Languages, 2015, pp. 44–58.

[36] P. M. Kogge, H. S. Stone, A parallel algorithm for the efficient solution of a general class of recurrence equations, IEEE Transactions on Computers 22 (8) (1973) 786–793.

[37] S. Sato, H. Iwasaki, Automatic parallelization via matrix multiplication, in: Proceedings of the 32nd ACM SIGPLAN Conference on Programming Language Design and Implementation, 2011, pp. 470–479.

[38] Y. Zou, S. V. Rajopadhye, Scan detection and parallelization in "inherently sequential" nested loop programs, in: Proceedings of the 10th Annual IEEE/ACM International Symposium on Code Generation and Optimization, 2012, pp. 74–83.

[39] U. Bondhugula, M. M. Baskaran, S. Krishnamoorthy, J. Ramanujam, A. Rountev, P. Sadayappan, Automatic transformations for communication-minimized parallelization and locality optimization in the polyhedral model, in: Proceedings of the 17th International Conference on Compiler Construction, 2008, pp. 132–146.

[40] U. Bondhugula, A. Hartono, J. Ramanujam, P. Sadayappan, A practical automatic polyhedral parallelizer and locality optimizer, in: Proceedings of the ACM SIGPLAN 2008 Conference on Programming Language Design and Implementation, 2008, pp. 101–113.

[41] K. Trifunovic, D. Nuzman, A. Cohen, A. Zaks, I. Rosen, Polyhedral-model guided loop-nest auto-vectorization, in: Proceedings of the 18th International Conference on Parallel Architectures and Compilation Techniques, 2009, pp. 327–337.

[42] P. Feautrier, Automatic parallelization in the polytope model, in: The Data Parallel Programming Model: Foundations, HPF Realization, and Scientific Applications, 1996, pp. 79–103.

[43] A. Morihata, K. Matsuzaki, Z. Hu, M. Takeichi, The third homomorphism theorem on trees: downward & upward lead to divide-and-conquer, in: Proceedings of the 36th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, 2009, pp. 177–185.