Weaving Parallel Threads
Searching for Useful Parallelism in Functional Programs

José Manuel Calderón Trilla[1(✉)], Simon Poulding[2], and Colin Runciman[1]

[1] University of York, York, UK
jmct@jmct.cc
[2] Blekinge Institute of Technology, Karlskrona, Sweden

Abstract. As the speed of processors is starting to plateau, chip manufacturers are instead looking to multi-core architectures for increased performance. The ubiquity of multi-core hardware has made parallelism an important tool in writing performant programs. Unfortunately, parallel programming is still considered an advanced technique and most programs are written as sequential programs.

We propose that we lift this burden from the programmer and allow the compiler to automatically determine which parts of a program can be executed in parallel. Historically, most attempts at auto-parallelism depended on static analysis alone. While static analysis is often able to find *safe* parallelism, it is difficult to determine *worthwhile* parallelism. This is known as the *granularity problem*. Our work shows that we can use static analysis *in conjunction with* search techniques by having the compiler execute the program and then alter the amount of parallelism based on execution speed. We do this by annotating the program with parallel annotations and using search to determine which annotations to enable.

This allows the static analysis to find the safe parallelism and shift the burden of finding worthwhile parallelism to search. Our results show that by searching over the possible parallel settings we can achieve better performance than static analysis alone.

1 Introduction

Moore's law has often provided a 'free lunch' for those looking to run faster programs without the programmer expending any engineering effort. Throughout the 1990 s in particular, an effective way of having a faster x86 program was to wait for Intel™ to release its new line of processors and run the program on your new CPU. Unfortunately, clock speeds have reached a plateau and we no longer get speedups for free [23]. Increased performance now comes from including additional processor cores on modern CPUs. This means that programmers have been forced to write parallel and concurrent programs when looking for improved wall-clock performance. Unfortunately, writing parallel and concurrent programs involves managing complexity that is not present in single-threaded programs. The goal of the work outlined in this paper is to convince

M. Barros and Y. Labiche (Eds.): SSBSE 2015, LNCS 9275, pp. 62–76, 2015.
DOI: 10.1007/978-3-319-22183-0_5

the reader that not all hope is lost. By looking for the *implicit parallelism* in programs that are written as single-threaded programs we can achieve performance gains without programmer effort.

Our work focuses on F-Lite: a pure, non-strict functional language that is suitable as a core language of a compiler for a higher-level language like Haskell [17]. We have chosen to use a non-strict language because of the lack of arbitrary side-effects [11], and many years of work in the area of implicit parallelism [6,10,13] however we feel that many of our techniques would transfer well to other language paradigms.

The primary contribution of this paper is to demonstrate that using search based on dynamic execution of the parallelised program is a robust way to help diminish the *granularity* problem that is difficult for static analysis alone. We show that for some programs, the combination of search and static analysis can achieve speed-ups that are nearly linear with respect to the number of cores.

The rest of this paper describes our technique in more detail. Section 2 discuss the main background to this work: implicit parallelism in functional languages. Section 3 provides a worked example to illustrate the static analysis we perform to determine potential parallelism. We describe our empirical method and results in Sect. 4. Lastly, we offer our conclusions and discuss related work in Sect. 6.

2 Implicit Parallelism in Functional Languages

In this section we will motivate and discuss the benefits and drawbacks of implicit parallelism in a lazy purely functional language. We will also give a high-level overview of *strictness analysis* which allows us to find safe parallelism in lazy languages.

2.1 Background

Research into parallelism in lazy purely functional languages has a long history that dates back to the early work on lazy functional languages [1,12,19,20][1]. Non-strictness makes it difficult to reason about when expressions are evaluated. This forces the programmer to avoid the use of arbitrary side-effects. The resulting purity means that functions in pure functional languages are *referentially transparent*, or the result of a function depends only on the values of its arguments (i.e. there is no global state that could effect the result of the function or be manipulated by the function).

Purity alone is of huge benefit when dealing with parallelism. Because functions do not rely on anything but their arguments the only communication between threads necessary is the result of the thread's computation, which is shared via the program's graph using the same mechanism used to implement laziness [19].

Laziness, while forcing the programmer to be pure (which is a boon to parallelism), is an inherently sequential evaluation strategy. Lazy evaluation only

[1] For a comprehensive review we suggest [7].

evaluates expressions when they are *needed*. This is what allows for the use of infinite data structures, only what is needed will be computed.

The two reductions of *sqr* in Fig. 1 illustrate the key differences between lazy evaluation and eager, or strict, evaluation.

Eager Evaluation	Lazy Evaluation
$sqr\ (5 * 5)$	$sqr\ (5 * 5)$
$= sqr\ 25$	$= let\ x\ =\ 5 * 5\ in\ x * x$
$= let\ x\ =\ 25\ in\ x * x$	$= let\ x\ =\ 25\ in\ x * x$
$= 25 * 25$	$= 25 * 25$
$= 625$	$= 625$

Fig. 1. Eager and Lazy evaluation order for squaring a value.

In the case of eager evaluation the argument to *sqr* is evaluated *before* entering the function body. For lazy evaluation the argument is passed as a suspended computation that is only *forced* when the value is needed (in this case when x is needed in order to multiply $x * x$). Notice that under lazy evaluation $5 * 5$ is only evaluated once, even though it is used twice in the function. This is due to the *sharing* of the result. This is why laziness is often described as call-by-need *with sharing* [7].

In the case of *sqr* in Fig. 1, both eager and lazy evaluation required the same number of *reductions* to compute the final result. This is not always the case; take the following function definitions

$$\begin{aligned} &bot\ ::\ Int\ \rightarrow\ Int \\ &bot\ x\ =\ x + bot \end{aligned}$$

$$\begin{aligned} &const\ ::\ a\ \rightarrow\ b\ \rightarrow\ a \\ &const\ x\ y\ =\ x \end{aligned}$$

In an eager language the expression *const* 5 *bot* will never terminate, while it would return 5 in a lazy language as only the first argument to *const* is actually *needed* in its body.

This tension between the call-by-need convention of laziness with parallelism's desire to evaluate expressions *before* they are needed is well known [24]. The most successful method of combating this tension is through the use of *strictness analysis* [9,16,27].

2.2 Strictness, Demand Context, and Strategies

Here we will describe the method by which we identify the *safe* parallelism in F-Lite programs and arrange for the evaluation of these expressions in parallel.

The *strictness* properties of a function determine which arguments are definitely needed for the function to terminate, whereas the *demand* on an argument tells us *how much* of the argument's structure is needed. *Strategies* are functions that evaluate their argument's structure to a specific depth. By analysing the program for strictness and demand information, we can then generate strategies for the strict arguments to a function and evaluate the strategies in parallel to the body of the function. The strategies we generate will only evaluate the arguments to the depth determined by the demand analysis.

Strictness. Because we are working in a lazy language it is not always safe to evaluate the arguments to a function before we enter the body of a function. However, if a function uses the value of an argument within its body it is safe to evaluate that argument before, or in parallel to, the execution of the body of the function. In order to determine which arguments can be evaluated in this way modern compilers use *strictness analysis* [16]. More formally, a function f of n arguments

$$f\ x_1 \ldots\ x_i\ \ldots x_n = \ldots$$

is strict in its ith argument if and only if

$$f\ x_1 \ldots\ \bot\ \ldots x_n = \bot$$

What this states is that f is only strict in its ith argument if f becomes non-terminating[2] by passing a non-terminating value as its ith argument.

Knowing the strictness information of a function is the first step in automatic parallelisation. This is because if f is strict in its ith argument we do not risk introducing non-termination (which would not otherwise be present) by evaluating the ith argument in parallel. In other words, evaluating x_i in parallel would only introduce non-termination to the program if evaluating f with x_i would have resulted in f's non-termination anyway.

F-Lite has two primitives for taking advantage of strictness information: *par* and *seq*.

$$\begin{array}{ll} seq\ ::\ a \to b \to b & par\ ::\ a \to b \to b \\ seq\ x\ y = y & par\ x\ y = y \end{array}$$

Fig. 2. Semantics of `seq` and `par`.

Both functions return the value of their second argument. The difference is in their *side-effects*. *seq* returns its second argument only *after* the evaluation of its first argument. *par* forks the evaluation of its first argument in a new parallel

[2] In this paper we use the convention that $\bot$ represents erroneous or non-terminating expressions.

thread and then returns its second argument; this is known as *sparking* a parallel task [4].

Strictness analysis was a very active research area in the 1980's and the development of analyses that provide the type of strictness information outlined above is a well understood problem [2,5,16]. However, as outlined above, strictness analysis does not provide satisfactory information about complex data-structures [26]. This can be remedied by the use of *projections* to represent *demand*.

Demand. So far our discussion of strictness has only involved two levels of 'definedness': a defined value, or ⊥. This is the whole story when dealing with *flat* data-structures such as Integers, Booleans or Enumerations. However, in lazy languages nested data-structures have *degrees* of definedness.

Take the following example function and value definitions in F-Lite

```
length []     = 0                    sum []     = 0
length (x:xs) = 1 + length xs        sum (x:xs) = x + sum xs

definedList = [1,2,3,4]              infiniteList = [1,2,3...

partialList = [1,2,bot,4]            loop = loop
```

Both `length` and `sum` are functions on lists, but they use lists differently. `length` does not use the elements of its argument list. Therefore `length` would accept `definedList` and `partialList` (which has a non-terminating element) as arguments and still return the correct value. On the other hand `sum` *needs* the elements of the list, otherwise it would not be able to compute the sum. For this reason, `sum` only terminates if it is passed a fully defined list and would result in non-termination if passed `partialList`. Neither function would terminate if passed `infiniteList`, since even `length` requires the list to have a finite length (some functions do not require a finite list, such as `head`, the function that returns the first element in a list). With these examples we say that `length` *demands* a finite list, whereas `sum` *demands* a fully-defined list.

This additional information about a data-structure is extremely useful when trying to parallelise programs. If we can determine *how much* of a structure is needed we can then evaluate the structure to that depth in parallel.

The work that introduced this representation of demands was by Wadler and Hughes [27] using the idea of *projections* from domain theory. The technique we use in our compiler is a projection-based strictness analysis based on the work in Hinze's dissertation [9]. Hinze's dissertation is also a good resource for learning the theory of projection-based strictness analysis.

Strategies. With the more sophisticated information provided by projection-based analysis, we require more than simply *par* and *seq*. To this end we use the popular technique of *strategies* for parallel evaluation [15,25]. Strategies are designed to evaluate structures up to a certain depth in parallel to the use of those

structures. Normally, strategies are written by the programmer for use in hand-parallelised code. In order to facilitate auto-parallelisation we have developed a method to *derive* an appropriate strategy from the information provided to us by projection-based strictness analysis. The rules for the derivation are presented as a denotational semantics and can be found in our earlier work [3].

2.3 The Granularity Problem

We have now discussed how we find the parallelism that is implicit in our program, but none of the analysis we provide determines whether the safe parallelism is *worthwhile*. Often static analysis will determine that a certain structure is *safe* to compute in parallel, but it is very difficult to know when it is actually of any benefit. Parallelism has overheads that require the parallel tasks to be substantial enough to make up for the cost. A *fine-grained* task is unlikely to require more computation than the cost of sparking and managing the thread, let alone the potential to interrupt productive threads [7,10].

One of the central arguments in our work is that static analysis *alone* is insufficient at finding both the implicit parallelism and determining whether the introduced parallelism is substantial enough to warrant the overheads.

Our proposal is that the compiler should *run* the program and use the information gained from running it (even if it only looks at overall execution time) to *remove* the parallelism that is too fine-grained. By doing this we shift the burden of the granularity problem away from our static analysis and onto our search techniques. This way our static analysis is only used to determine the safe parallel expressions, and not the granularity of the expressions.

3 Overview

In this section we will present a high-level overview of our technique. This will provide the context for our discussion in the subsequent sections.

The program listed in Fig. 3 is the Tak program benchmark, often used for testing the performance of recursion in interpreters and code generated by compilers [14].

```
tak :: Int -> Int -> Int -> Int
tak x y z = case x <= y of
                True  -> z
                False -> tak (tak (x - 1) y z)
                             (tak (y - 1) z x)
                             (tak (z - 1) x y)

main = tak 24 16 8
```

Fig. 3. Source listing for Tak

After we perform our projection-based strictness analysis, and introduce the safe `par` annotations, we transform the program into a parallelised version. The result of this transformation on Tak is listed in Fig. 4.

```
tak x y z = case x <= y of
                True  -> z
                False -> let x' = tak ((x - 1)) y z
                             y' = tak ((y - 1)) z x
                             z' = tak ((z - 1)) x y
                         in (par x'
                                (par y'
                                    (seq z'
                                        (tak x' y' z'))))

main  = tak 24 16 8
```

Fig. 4. Source listing for Tak after analysis, transformation, and `par` placement

Each strict argument is given a name via a `let` binding. This is so that any parallel, or `seq`ed, evaluation can be shared between threads. When there are multiple strict arguments (as is the case for `tak`) we spark the arguments in left-to-right order except for the last strict argument, which we `seq`. This is a common technique that is used to avoid potential collisions [25]. Collisions occur when a thread requires the result of another thread before the result has been evaluated. By ensuring that one of the arguments is evaluated in the current thread (by using `seq`) we give the parallel threads more time to evaluate their arguments, lessening the frequency of collisions.

While static analysis has determined that `x'` and `y'` can be evaluated in parallel *safely*, it does not determine whether parallel evaluation of those expressions is *worthwhile*. In order to address this issue we take advantage of two key properties of our `par` annotations:

1. Each introduced `par` sparks off a unique subexpression in the program's source
2. The semantics of `par` (as shown in Fig. 2) allow us to return its second argument, ignoring the first, without changing the semantics of the program as a whole.

These two properties allow us to represent the `par`s placed by static analysis and transformation as a bit string. Each bit represents a specific `par` in the program AST. When a `par`'s bit is 'on' the `par` behaves as normal, sparking off its first argument to be evaluated in parallel and return its second argument. When the bit is 'off' the `par` returns its second argument, ignoring the first.

This allows us to change the *operational* behavior of the program without altering any of the program's semantics.

4 Experimental Setup and Results

In this section we evaluate the use of search in finding an effective enabling of `par`s that achieves a worthwhile speed-up when the parellelised program is run in a multi-core architecture. As a reminder, the starting point for our proposed technique is a program that was originally written to be run sequentially on a single core; static analysis identifies potential sites at which `par` functions *could* be applied; and then search is used to determine the subset of sites at which the `par` is actually applied.

4.1 Research Questions

Our hypothesis is that enabling a subset of the `par`s will often be preferable to enabling them all, hence the first research question:

RQ1. What speed-up is achieved by using search to enable a subset of `par`s compared to the enabling all the `par`s found by static analysis?

Since the overall goal is to speed-up a sequential program by parallelising it to use multiple cores, the second question is:

RQ2. What speed-up is achieved by parallelisation using search compared to the original software-under-test (SUT) executed as a sequential program?

In this empirical work, we consider two algorithms: a simple hill-climbing algorithm and a greedy algorithm:

RQ3. Which search algorithm achieves the larger speed-ups, and how quickly do these algorithms achieve these speed-ups?

Since some `par`s can only have an effect when one or more other `par`s are also enabled, there is an argument that a sensible starting point for both algorithms is to have all `par`s enabled. An alternative is to start with a random subset of the `par`s enabled. This motivates the final research question:

RQ4. Which form of initialisation enables the algorithm to find the best speed-ups: all `par`s enabled (we refer to this as *'all-on'* initialisation), or a random subset enabled (*'random'* initialisation)?

4.2 Algorithms

Representation. We represent the choice of enabled `par`s as a bit string where a 1 indicates that the `par` is applied at a site, and 0 that it is not. The length of the bit string is the number of potential `par`s annotations found by the static analysis.

Fitness. To facilitate experimentation, the SUTs are executed using a simulator which records the number of reductions made by each thread. A parameter to the simulator controls the number of cores available to the SUT, and thus the maximum number of threads that may be run in parallel. We choose the number of reductions made by the main thread as the fitness metric. The main thread

cannot complete until all the other threads it has started have completed, and so this number of reductions is an indication of the SUT's runtime. The simulator includes a realistic overhead of 250 reductions for handling each additional thread.

Hill-Climbing Algorithm. We utilise a simple hill-climbing algorithm in which the neighbours of the current bitstring are those formed by flipping a single bit. At each iteration, these neighbours of the current bitstring are considered in a random order, and the fitness evaluated for each in turn. The first neighbour that has a better fitness, i.e. fewer reductions are made by the main thread, than the current bitstring becomes the current bitstring in the next iteration. The algorithm terminates when no neighbour of the current bitstring has a better fitness.

Greedy Algorithm. The greedy algorithm considers the bits in representation in a random order. As each bit is considered, the bit is flipped from its current setting and the resulting bit string evaluated; the setting of the bit—current or flipped—with the better fitness is retained. The algorithm terminates once all the bits have been evaluated.

4.3 Software-Under-Test

SumEuler. SumEuler is a common parallel functional programming benchmark first introduced with the work on the $\langle \nu, G \rangle$-Machine in 1989 [1]. This program is often used a parallel compiler benchmark making it a 'sanity-check' for our work. We expect to see consistent speed-ups in this program when parallelised (9 `par` sites).

Queens + Queens2. We benchmark two versions of the nQueens program. Queens2 is a purely symbolic version that represents the board as a list of lists and does not perform numeric computation (10 `par` sites for Queens and 24 for Queens2). The fact that Queens2 has more than double the number of `par` sites for the same problem shows that writing in a more symbolic style provides more opportunity for *safe* parallelism.

SodaCount. Solves a word search problem for a given grid of letters and a list of keywords. Introduced by Runciman and Wakeling, this program was chosen because it exhibits a standard search problem and because Runciman and Wakeling hand-tuned and profiled a parallel version, demonstrating that impressive speed-ups are possible with this program [21] (15 `par` sites).

Tak. Small recursive numeric computation that calculates a Takeuchi number. Knuth describes the properties of Tak in [14] (2 `par` sites).

Taut. Determines whether a given predicate expression is a tautology. This program was chosen because the algorithm used is *inherently sequential*. We feel that it was important to demonstrate that not all programs have implicit parallelism within them, sometimes the only way to achieve parallel speed-ups is to rework the algorithm (15 `par` sites).

MatMul. List of list matrix multiplication. Matrix multiplication is an inherently parallel operation, we expect this program to demonstrate speed-ups when parallelised (7 `par` sites).

4.4 Method

The following four algorithm configurations were evaluated:

- hill-climbing with all-on initialisation
- greedy with all-on initialisation
- hill-climbing with random initialisation
- greedy with random initialisation.

Each algorithm configuration was evaluated for four settings of the number cores: 4, 8, 16 and 24 cores. Each algorithm/core count combination was evaluated against each of the seven SUTs described above.

Since both search algorithms are stochastic, multiple runs were made for each algorithm/core count/SUT combination, each using 30 different seeds to the pseudo-random number generator. For all runs, after each fitness evaluation, the best bit string found and its fitness (the number of reductions made by the main thread), was recorded.

In addition, the fitness (number of reductions) was evaluated for a bit string where all bits are set to 1: this equivalent to using the static analysis without optimisation using search. This evaluation was made for each combination of core count and SUT. Finally, the fitness was evaluated for the sequential version of each SUT.

4.5 Results

The results are summarised in Table 1. This table compares the speed-up, calculated as the ratio of the medians of the reduction counts, of hill-climbing with all-on initialisation compared to (a) the parallelisation that would result from the static analysis without optimisation; (b) the sequential version of the program; (c) the greedy algorithm with all-on initialisation; and (d) the hill-climbing algorithm with random initialisation. The speed-up is calculated as the factor by which the number of reductions is reduced, and so values greater than 1 indicate that the SUT parallelised using hill-climbing with all-on initialisation would be faster in the multi-core environment. Values in bold in the table indicate that differences between the algorithms used to calculate the speed-up are statistically significant at the 5 % level using a one- or two-sample Wilcoxon test as appropriate[3].

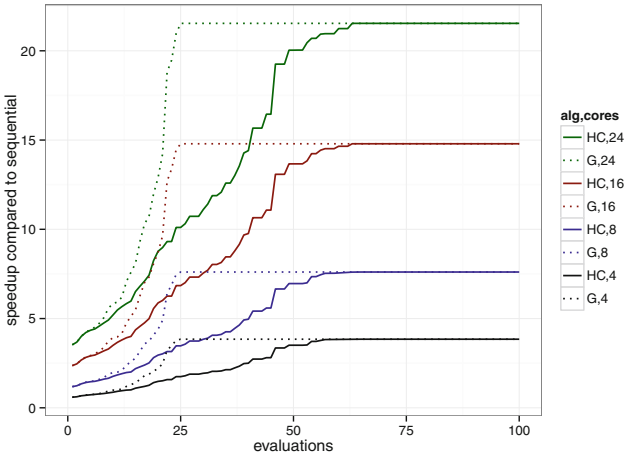
[3] Since in the following we discuss the results for each SUT, or combination of SUT and number of cores, individually as well as for the entire set of results as a family, we do not apply a Bonferroni or similar correction to the significance level. Nevertheless we note here that most of the currently significant differences would remain significant if such a correction were applied.

Table 1. The speed-up, calculated as the ratio of the medians of the reduction counts, achieved by the hill-climbing algorithm using all-on initialisation compared to the default parallelisation from static analysis (static parallel), a sequential implementation of the SUT (sequential), the greedy algorithm (greedy), and hill climbing using random initialisation (random init). Speed-ups are rounded to 4 significant figures. Values in bold font are significant at the 5 % level.

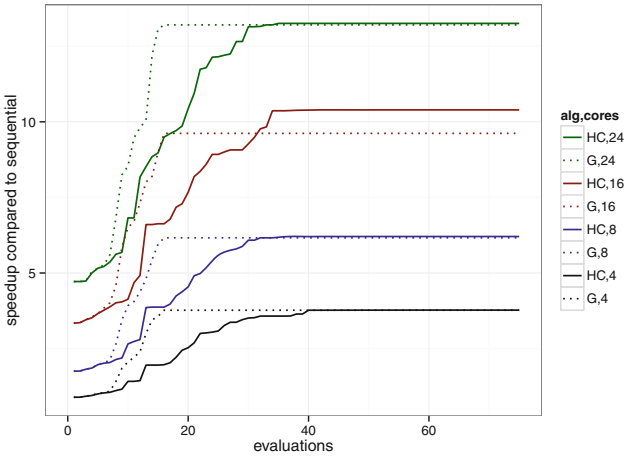
		Hill-climbing speed-up compared to:			
SUT	Cores	Static parallel	Sequential	Greedy	Random init
MatMul	4	**4.903**	**1.021**	1	1
MatMul	8	**4.625**	**1.021**	1	1
MatMul	16	**4.485**	**1.021**	1	1
MatMul	24	**4.439**	**1.021**	1	1
Queens	4	**1.080**	**1.294**	1	1
Queens	8	**1.043**	**1.369**	1	1
Queens	16	**1.017**	**1.401**	1	1
Queens	24	**1.003**	**1.401**	**1.000**	1
Queens2	4	**6.479**	**3.843**	1	1
Queens2	8	**6.421**	**7.607**	1	1
Queens2	16	**6.263**	**14.79**	1	1
Queens2	24	**6.101**	**21.54**	1	1
SodaCount	4	**4.237**	**3.773**	**1.000**	**1.055**
SodaCount	8	**3.544**	**6.207**	**1.007**	**1.071**
SodaCount	16	**3.110**	**10.40**	**1.081**	**1.072**
SodaCount	24	**2.810**	**13.26**	**1.004**	1
SumEuler	4	**1.494**	**3.948**	1	1
SumEuler	8	**1.486**	**7.773**	1	1
SumEuler	16	**1.460**	**14.77**	**1**	1
SumEuler	24	**1.432**	**20.69**	**1**	1
Tak	4	**1.609**	**1.560**	1	1
Tak	8	**1.609**	**3.118**	1	1
Tak	16	**1.608**	**6.230**	1	1
Tak	24	**1.608**	**9.330**	1	1
Taut	4	**1.000**	**1.000**	**1.000**	1
Taut	8	**1.000**	**1.000**	**1.000**	1.000
Taut	16	**1.000**	**1.000**	**1.000**	1
Taut	24	**1.000**	**1.000**	**1.000**	1

4.6 Discussion

RQ1. For most of SUTs there is a relatively large speed-up of the hill-climbing algorithm compared to the default parallelisation where all `par`s are enabled. The largest speed-ups are for Queens2 where we might expect a wall-clock run time that is more than 6 times better than the default parallelisation. For Queens and Taut the speed-ups are closer to 1, but are in all cases statistically significant.



(a) Queens2



(b) SodaCount

Fig. 5. The speed-up, calculated as the ratio of the medians of the reduction counts, obtained so far by the algorithm plotted against the number of fitness evaluations. HC and G indicate the hill-climbing and greedy algorithm respectively, both using all-on initialisation. The numbers following the algorithm abbreviation indicate the number of cores (Color figure online).

We conclude that the hill-climbing algorithm can improve parallel performance across a range of SUTs and across a range of core counts.

RQ2. For Queens2 and SumEuler, the speed-up compared the sequential version of these SUTs is almost linear: it approaches the number of cores available. For example, for SumEuler on 4 cores, the speed-up compared to the sequential version is 3.95. A linear speed-up is the best that can be achieved, and so these results are indicative that our proposed technique could be very effective in practice. Meanwhile, for other SUTs such as MathMaul and Taut, there is little speed-up over the sequential version of the SUT.

RQ3. The results show that for most SUTs, there is little difference in the speed-up achieved by the hill-climbing and greedy algorithm. (For clarity, the table shows the comparison only between the two algorithms using all-on initialisation, but similar results are obtained when initialisation is random.) Only for SodaCount is there a non-trivial and statistically significant difference between the hill climber and greedy algorithm for all core sizes. Figure 5 performs a further analysis for this research question: for two of the SUTs, it plots the best speed-up (compared to sequential) obtained so far by the algorithm against the number of fitness evaluations. For Queens2 at all core counts, the greedy algorithm finds the same best speed-up as the hill-climbing, but finds it in fewer fitness evaluations, i.e. the search is faster. For SodaCount, the greedy algorithm finds its best speed-up in relatively few evaluations. The hill-climber takes longer but finds a better speed-up at all cores counts; the difference is most noticeable in the results for 16 cores. For frequently-used SUTs that account for a significant part of a system's performance, the additional effort required to find the best parallelisation using hill-climbing may be justified, but will depend on context.

RQ4. For most SUTs there is no statistically significant difference between all-on and random initialisation. For SodaCount, the all-on initialisation is slightly better for core counts of 4, 8, and 16. This result provides evidence that all-on initialisation may be beneficial, but requires further investigation to confirm the generality.

5 Related Work

Research into parallel *functional* programming has been an active research area since the early 1980s. Before research into implicit parallelism fell out of favor, much of the work focused on the use of static analysis alone in parallelising programs [7,10]. Harris and Singh used runtime feedback to *find* parallelism in functional programs without the use of static analysis [8]. Our approach can be seen as reversal of their approach, *introduce* parallelism at compile-time and *remove* parallelism using runtime feedback.

A number of researchers in the late 1990 s applied metaheuristic search to transform serial *imperative* programs into parallel ones. Both Nisbet [18] and

Williams [28] independently targeted FORTRAN programs using metaheuristics to find an appropriate sequence of code transformation to enable the program to take advantage of a target parallel architecture. The Paragen framework described by Ryan and his collaborators applies genetic programming to optimise a tree-like representation of parallelising transformations that are applied to blocks of code, and a linear representation of transformations that are applied to loops in the program [22]. The fitness used by Paragen is a combination of the speed-up obtained and the equivalence of the serial and parallel versions of the program based on a post hoc analysis of data dependencies. The two key differences from the work described in this paper are that: (a) here the search does not derive a sequence of transformations, but instead determines which potential transformations, found by prior static analysis, are enabled; and, (b) any transformed parallel program is guaranteed to be equivalent to the original serial program by construction. We believe that these differences may facilitate scalability in our approach.

6 Conclusions

We have shown in this paper that the combination of static analysis and search can parallelise programs. For some programs we are able to achieve close to linear speed-ups which is as performant as can expected. As future work we will investigate more sophisticated algorithms, including genetic algorithms and estimation of distribution algorithms; and confirm the scalability of our approach.

References

1. Augustsson, L., Johnsson, T.: Parallel graph reduction with the $\langle v, G\rangle$-machine. In: Proceedings of the Fourth International Conference on Functional Programming Languages and Computer Architecture. FPCA 1989, pp. 202–213. ACM, New York (1989)
2. Burn, G.L., Hankin, C., Abramsky, S.: Strictness analysis for higher-order functions. Sci. Comput. program. **7**, 249–278 (1986)
3. Calderón Trilla, J.M., Runciman, C.: Improving implicit parallelism. In: Proceedings of the ACM SIGPLAN Symposium on Haskell. Haskell 2015 (2015). Under submission
4. Clack, C., Peyton Jones, S.: The four-stroke reduction engine. In: Proceedings of the 1986 ACM Conference on LISP and Functional Programming, pp. 220–232. ACM (1986)
5. Clack, C., Peyton Jones, S.L.: Strictness analysis-a practical approach. In: Jouannaud, J.-P. (ed.) Functional Programming Languages and Computer Architecture. LNCS, vol. 201, pp. 35–49. Springer, Heidelberg (1985)
6. Hammond, K.: Parallel functional programming: an introduction (1994). http://www-fp.dcs.st-and.ac.uk/~kh/papers/pasco94/pasco94.html
7. Hammond, K., Michelson, G.: Research Directions in Parallel Functional Programming. Springer-Verlag (2000)
8. Harris, T., Singh, S.: Feedback directed implicit parallelism. SIGPLAN Not. **42**(9), 251–264 (2007). http://doi.acm.org/10.1145/1291220.1291192

9. Hinze, R.: Projection-based strictness analysis: theoretical and practical aspects. Inaugural dissertation, University of Bonn (1995)
10. Hogen, G., Kindler, A., Loogen, R.: Automatic parallelization of lazy functional programs. In: Krieg-Brückner, B. (ed.) ESOP 1992. LNCS, vol. 582, pp. 254–268. Springer, Heidelberg (1992)
11. Hughes, J.: Why functional programming matters. Comput. J. **32**(2), 98–107 (1989)
12. Hughes, R.J.M.: The design and implementation of programming languages. Ph.D. thesis, Programming Research Group, Oxford University, July 1983
13. Jones, M., Hudak, P.: Implicit and explicit parallel programming in haskell (1993). Distributed via FTP at http://nebula.systemsz.cs.yale.edu/pub/yale-fp/reports/RR-982.ps.Z. Accessed July 1999
14. Knuth, D.E.: Textbook examples of recursion. In: Lifschitz, V. (ed.) Artificial Intelligence and Theory of Computation, pp. 207–229. Academic Press, Boston (1991)
15. Marlow, S., Maier, P., Loidl, H., Aswad, M., Trinder, P.: Seq no more: better strategies for parallel haskell. In: Proceedings of the Third ACM Haskell Symposium on Haskell, pp. 91–102. ACM (2010)
16. Mycroft, A.: The theory and practice of transforming call-by-need into call-by-value. In: Robinet, B. (ed.) International Symposium on Programming. LNCS, vol. 83, pp. 269–281. Springer, Heidelberg (1980)
17. Naylor, M., Runciman, C.: The reduceron reconfigured. ACM Sigplan Not. **45**(9), 75–86 (2010)
18. Nisbet, A.: GAPS: A compiler framework for genetic algorithm (GA) optimised parallelisation. In: Proceedings of the International Conference and Exhibition on High-Performance Computing and Networking, pp. 987–989. HPCN Europe 1998 (1998)
19. Peyton Jones, S.L.: Parallel implementations of functional programming languages. Comput. J. **32**(2), 175–186 (1989)
20. Plasmeijer, R., Eekelen, M.V.: Functional Programming and Parallel Graph Rewriting, 1st edn. Addison-Wesley Longman Publishing Co., Inc., Boston (1993)
21. Runciman, C., Wakeling, D. (eds.): Applications of Functional Programming. UCL Press Ltd., London (1996)
22. Ryan, C., Ivan, L.: Automatic parallelization of arbitrary programs. In: Langdon, W.B., Fogarty, T.C., Nordin, P., Poli, R. (eds.) EuroGP 1999. LNCS, vol. 1598, pp. 244–254. Springer, Heidelberg (1999)
23. Sutter, H.: The free lunch is over: a fundamental turn toward concurrency in software. Dr. Dobbs J. **30**(3), 202–210 (2005)
24. Tremblay, G., Gao, G.R.: The impact of laziness on parallelism and the limits of strictness analysis. In: Proceedings High Performance Functional Computing, pp. 119–133. Citeseer (1995)
25. Trinder, P.W., Hammond, K., Loidl, H.W., Peyton Jones, S.L.: Algorithm + strategy = parallelism. J. Funct. Program. **8**(1), 23–60 (1998)
26. Wadler, P.: Strictness analysis on non-flat domains. In: Abramsky, S., Hankin, C.L. (eds.) Abstract Interpretation of Declarative Languages, pp. 266–275. Ellis Horwood, Chichester (1987)
27. Wadler, P., Hughes, R.J.M.: Projections for strictness analysis. In: Kahn, G. (ed.) FPCA 1987. LNCS, vol. 274, pp. 385–407. Springer, Heidelberg (1987)
28. Williams, K.P.: Evolutionary algorithms for automatic parallelization. Ph.D. thesis, Department of Computer Science, University of Reading, December 1998