# Estimation of Parallel Complexity with Rewriting Techniques

Christophe Alias, Carsten Fuhs, Laure Gonnord

# Estimation of Parallel Complexity with Rewriting Techniques

Author version, accepted to WST 2016 *

Christophe Alias
INRIA & LIP (UMR CNRS/ENS Lyon/UCB Lyon1/INRIA), Lyon, France

Carsten Fuhs
Birkbeck, University of London, United Kingdom

Laure Gonnord
University of Lyon & LIP (UMR CNRS/ENS Lyon/UCB Lyon1/INRIA), Lyon, France

July 20, 2016

**Abstract**

We show how monotone interpretations – a termination analysis technique for term rewriting systems – can be used to assess the inherent parallelism of recursive programs manipulating inductive data structures. As a side effect, we show how monotone interpretations specify a parallel execution order, and how our approach extends naturally affine scheduling – a powerful analysis used in parallelising compilers – to recursive programs. This work opens new perspectives in automatic parallelisation.

## 1 Introduction

The motivation of this work is the automatic transformation of sequential code into parallel code without changing its (big-step operational) semantics, only changing the computation order is allowed. We want to find out the limits of these approaches, by characterising the "maximum level of parallelism" that we can find for a given sequential implementation of an algorithm.

In this paper, we propose a way to estimate the parallel complexity which can be informally defined as the complexity of the program if it were executed on an machine with unbounded parallelism.

Such a result could come as by-product of the polyhedral-based automatic parallelisation techniques for array-based programs [6]. However, for general programs with complex data flow and inductive structures, such techniques have not been explored so far.

The contribution of this paper is a novel unifying way to express program dependencies for general programs with inductive data structures (lists, trees...) as well as a way to use complexity bounds of term rewriting systems in order to derive an estimation of this parallel complexity.

*This work was partially supported by a "BQR" funding at ENS De Lyon.

## 2 Intuitions behind the notion of "parallel complexity"

```
for(i=0; i<=N; i++)
  for (j=0; j<=N; j++)
    //Block S
    {
      m1[i][j] = Integer.MIN_VALUE;
      for(k=1; k<=i; k++)
        m1[i][j] = max(m1[i][j],H[i-k][j] + W[k]);

      m2[i][j] = Integer.MIN_VALUE;
      for(k=1; k<=j; k++)
        m2[i][j] = max(m2[i][j],H[i][j-k] + W[k]);

      H[i][j] = max(0,H(i-1,j-1)+s(a[i],b[i]),
                    m1[i][j],m2[i][j]);
    }
```

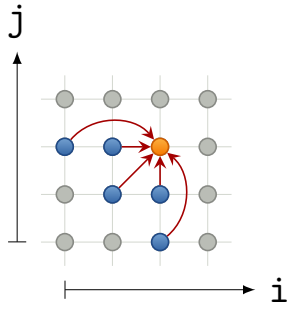


Figure 1: The Smith-Waterman sequence alignment algorithm and its dependencies. Each point $(i, j)$ represents an execution of the block $S$, denoted by $\langle S, i, j \rangle$.

```
public class Tree {
  private int val;
  private Tree left;
  private Tree right;

  public int treeMax() {
    int leftMax = Integer.MIN_VALUE;
    int rightMax = Integer.MIN_VALUE;
    if (this.left != null) {
      leftMax = this.left.treeMax();  // S1
    }
    if (this.right != null) {
      rightMax = this.right.treeMax();// S2
    }
    return Math.max(this.val, Math.max(leftMax, rightMax));
  }
}
```

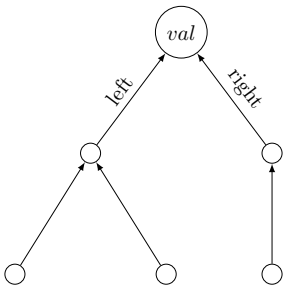


Figure 2: Maximum element of a tree algorithm and its call tree.

In this paper, we consider the derivation of a lower bound for the intrinsic parallelism of a sequential imperative program $P$ and thus an upper bound for the complexity of a fully parallelised version of $P$. Figures 1 and 2 depict the two motivating examples that will be studied in the remainder of this paper. Figure 1 is a loop kernel computing the Smith-Waterman optimal sequence alignment algorithm[1]. Figure 2 is a simple recursive function to compute the maximum element of a binary tree.

Usually, the parallelism is found by analysing the data dependencies between the operations executed by the program. There is a data dependency from operation $o_1$ to operation $o_2$ in the execution of a program if $o_1$ is executed before $o_2$ and both operations access the same

[1]See `https://en.wikipedia.org/wiki/Smith-Waterman_algorithm`. We consider two sequences of the same length $N$.

memory location. In particular, we have a flow dependency when the result of $o_1$ is required by $o_2$. Non-flow dependencies (write after write, write after read) can be removed by playing on the memory allocation [3] and can thus be ignored. Thus, we consider only flow dependencies referred as dependencies in the remainder of this paper. If there is no dependency between two operations, then they may be executed in parallel. While the dependency relation is in general undecidable, in practice we can use decidable over-approximations such that a statement that two operations are independent is always correct.

In Figure 1, each point represents an execution of the block $S$ computing `H[i][j]` for a given $i$ and $j$ in $[\![0, N]\!]$. Such an operation is written $\langle S, i, j\rangle$. The arrows represents the dependencies towards a given $\langle S, i, j\rangle$. For instance the diagonal arrow means that `H[i][j]` requires the value of `H[i-1][j-1]` to be computed.

In Figure 2, we depicted the execution trace of the recursive program on a tree. Here, the dependencies between computations resemble the recursive calls: the dependency graph (in the sense used for imperative programs) is the call tree.

All reorderings of computations respecting the dependencies are valid orderings (that we will name *schedule* in the following). In both cases, the dependencies we draw show a certain *potential parallelism*. Indeed, each pair of computations that do not transitively depend on each other can be executed in parallel; moreover, even a machine with infinite memory and infinitely parallel cannot do the computation in an amount of time which is shorter than the longest path in the dependency graph. The length of this longest path, referred to as *parallel complexity*, is thus an estimation of the *potential parallelism* of the program (its execution time with suitably reordered instructions on an idealised parallel machine).

Our goal in this paper is, given a sequential program $P$ and an over-approximation of its dependency relation, to find bounds on the parallel complexity of $P$ and the over-approximation of its dependency relation. Via the representation of the dependencies via (possibly constrained) rewrite rules, we can apply existing techniques to find such bounds for (constrained) rewrite systems (e.g., [8, 4]).

# 3 Computing the parallel complexity of programs

In this section we explain on the two running examples the relationship between termination and scheduling (this relationship was already explored a bit in [1]), and polynomial interpretations (more generally, proofs of complexity bounds for term rewriting) and parallel complexity. As the long-term goal is to devise compiler optimisations by automatic parallelisation for imperative programs, we consider programs with data structures such as `arrays`, `structs`, `records` and even `classes`. The main idea is that all these constructions can be classically represented as terms via the notion of *position*, and the dependencies as term rewriting rules. Contrarily to other approaches for proving termination of programs by an abstraction to rewriting (e.g., [11, 5]), we only encode the dependencies (and forget about the control flow).

## 3.1 First example: Smith-Waterman algorithm

For the Smith-Waterman program of Figure 1, we can derive (with polyhedral array dataflow analysis, as in [3]), forgetting about the local computations of $W$ scores, the following dependencies as constrained rewrite rules:

$$\mathsf{dep}(i, j) \to \mathsf{dep}(i-1, j-1) : 0 \le i \le n, 0 \le j \le n$$
$$\mathsf{dep}(i, j) \to \mathsf{dep}(i-k, j) : 0 \le i \le n, 0 \le j \le n, 1 \le k \le i$$
$$\mathsf{dep}(i, j) \to \mathsf{dep}(i, j-\ell) : 0 \le i \le n, 0 \le j \le n, 1 \le \ell \le j$$

To get an estimation of the parallel complexity, following the inspiration of previous work on termination proofs for complexity analysis (e.g., [8, 1, 4]), we first try to generate a *polynomial interpretation* [10] to map function symbols and terms to polynomials:

$$Pol(\mathsf{dep}(x_1, x_2)) = x_1 + x_2$$

Essentially, this says that iteration $\langle S, x_1, x_2\rangle$ can be executed at time stamp $x_1 + x_2$. As $0 \leq x_1, x_2 \leq N$, the maximal timestamp is $2N = O(N)$. Not only does this mean that the program may be parallelised, but it provides an actual reordering of the computation, along the parallel wavefront $x_1 + x_2$.

This linear interpretation (or ranking function) provides us with a bound in $O(N^1)$ for the parallel complexity of the program. Thus, the *degree of sequentiality* is 1. As the overall runtime of the program is in $O(N^2)$, this gives us a *degree of parallelism* of $2 - 1 = 1$ [9].

Let us point out that we would have obtained the same results using affine scheduling techniques from the polyhedral model [6]. The interesting fact here is that our apparatus is not restricted to regular programs (for loops, arrays) as the polyhedral model. Also, current complexity analysis tools like KoAT [4] are able to compute similar results within a reasonable amount of time. The next section show how our technique applies to a recursive program on inductive data structures.

### 3.2 Second example: computing the maximum element in a tree

An object of class `Tree` is represented by the term $\mathsf{Tree}(\mathit{val}, \mathit{left}, \mathit{right})$, for some terms *val*, *left*, *right* that represent its attributes [11]. For instance, the Java object defined in Figure 3 corresponds to the following term:

$$t = \mathsf{Tree}(2, \mathsf{Tree}(3, \mathsf{null}, \mathsf{null}), \mathsf{Tree}(4, \mathsf{Tree}(7, \mathsf{null}, \mathsf{null}), \mathsf{null}))$$

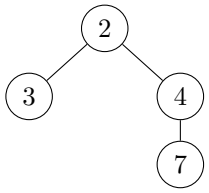

Figure 3: The tree $t$.

Recall that to address entries of an $n$-dimensional array, we use vectors $(i_1, \ldots, i_n) \in \mathbb{N}^n$ as indices or "positions". Then for an array `A`, we say that `A[`$i_1$`]`$\ldots$`[`$i_n$`]` is the "entry" at the array's position $(i_1, \ldots, i_n)$. Similarly to array entries, we would also like to address particular "entries" of a term, i.e., its *subterms*.

For more general terms, we can use a similar notion: *positions*. For our tree $t$, we have $\mathit{Pos}(t) = \{\varepsilon, 1, 2, 3, 21, 22, 23, \ldots\}$, giving an absolute way to access each element or subarray. For instance, 21 denotes the first element of the left child of the tree (i.e., the number 3).

Now let us consider the program in Figure 2. This function computes recursively the maximum value of an integer binary tree. Clearly, the computation of the maximum of a tree depends on the computation of each of its children. However, the computation of the max of each child is independent from the other. There is thus potential parallelism.

Here we observe structural dependencies from the accesses to the children of the current node. Like in the previous example, from the program we generate the following "dependency-rewrite rules" (note that similar to the dependency pair setting for termination proving [2], it suffices to consider "top-rewriting" with rewrite steps only at the root of the term):

$$\mathsf{dep}(\mathsf{Tree}(\mathit{val}, \mathit{left}, \mathit{right})) \to \mathsf{dep}(\mathit{left}) \qquad (S1)$$
$$\mathsf{dep}(\mathsf{Tree}(\mathit{val}, \mathit{left}, \mathit{right})) \to \mathsf{dep}(\mathit{right}) \qquad (S2)$$

We can use the following polynomial interpretation (analogous to the notion of a ranking function) to prove termination and also a complexity bound:

$$Pol(\mathsf{dep}(x_1)) = x_1 \qquad \text{and} \qquad Pol(\mathsf{Tree}(x_1, x_2, x_3)) = x_2 + x_3 + 1$$

Or, using interpretations also involving the maximum function [7]:

$$Pol(\mathsf{dep}(x_1)) = x_1 \qquad \text{and} \qquad Pol(\mathsf{Tree}(x_1, x_2, x_3)) = \max(x_2, x_3) + 1$$

Thus we interpret a tree as the maximum of its two children plus one to prove termination of the dependency relation of the original sequential program, essentially mapping a tree to its *height*. This means that the parallel complexity (i.e., the length of a chain in the dependency relation) of the program is bounded by the height of the input data structure.

Indeed, the two recursive calls could be executed in parallel, with a runtime bounded by the height of the tree on a machine with unbounded parallelism. The interpretation $Pol(\mathsf{Tree}(x_1, x_2, x_3)) = \max(x_2, x_3) + 1$ induces a wavefront for the parallel execution along the levels of the tree, i.e., the nodes at the same depth in the tree.

In contrast, the overall runtime of the original sequential program is bounded only by the *size* of the input tree, which may be exponentially larger than its depth.

# 4 Conclusion and Future Work

In this paper we showed some preliminary results on the (automatable) computation of the parallel complexity of programs with inductive data structures.

In the future, we will investigate a complete formalisation of these preliminary results, and test for applicability in more challenging programs like *heapsort* and *prefixsum*. As we said in the introduction, expressing the parallel complexity is the first step toward more ambitious use of rewriting techniques for program optimisation. The work in progress includes the computation of parallel schedules from the rewriting rules or their interpretation, and then parallel code generation from the obtained schedules.

# References


[1] Christophe Alias, Alain Darte, Paul Feautrier, and Laure Gonnord. Multi-dimensional rankings, program termination, and complexity bounds of flowchart programs. In *Proc. SAS '10*, pages 117–133, 2010.

[2] Thomas Arts and Jürgen Giesl. Termination of term rewriting using dependency pairs. *Theoretical Computer Science*, 236:133–178, 2000.

[3] Denis Barthou, Albert Cohen, and Jean-François Collard. Maximal static expansion. *International Journal of Parallel Programming*, 28(3):213–243, June 2000.

[4] Marc Brockschmidt, Fabian Emmes, Stephan Falke, Carsten Fuhs, and Jürgen Giesl. Analyzing runtime and size complexity of integer programs. *ACM TOPLAS*, 2016. To appear.

[5] Stephan Falke, Deepak Kapur, and Carsten Sinz. Termination analysis of C programs using compiler intermediate languages. In *Proc. RTA '11*, pages 41–50, 2011.

[6] Paul Feautrier. Some efficient solutions to the affine scheduling problem, I, one-dimensional time. *International Journal of Parallel Programming*, 21(5):313–348, October 1992.

[7] Carsten Fuhs, Jürgen Giesl, Aart Middeldorp, Peter Schneider-Kamp, René Thiemann, and Harald Zankl. Maximal termination. In *Proc. RTA '08*, pages 110–125, 2008.

[8] Nao Hirokawa and Georg Moser. Automated complexity analysis based on the dependency pair method. In *Proc. IJCAR '08*, pages 364–379, 2008.

[9] Richard M. Karp, Raymond E. Miller, and Shmuel Winograd. The organization of computations for uniform recurrence equations. *Journal of the ACM*, 14(3):563–590, 1967.

[10] Dallas S. Lankford. On proving term rewriting systems are Noetherian. Technical Report MTP-3, Louisiana Technical University, Ruston, LA, USA, 1979.

[11] Carsten Otto, Marc Brockschmidt, Christian von Essen, and Jürgen Giesl. Automated termination analysis of Java Bytecode by term rewriting. In *RTA '10*, pages 259–276, 2010.