# On the Synthesis and Reconfiguration of Pipelines

Diogo Telmo Neves[†], Rui C. Gonçalves[‡]
Department of Informatics, University of Minho
Campus of Gualtar, Braga, Portugal
Email: {[†]dneves, [‡]rgoncalves}@di.uminho.pt

***Abstract*—In recent years we have observed great advances in parallel platforms and the exponential growth of datasets in several domains. Undoubtedly, parallel programming is crucial to harness the performance potential of such platforms and to cope with very large datasets. However, quite often one has to deal with legacy software systems that may use third-party frameworks, libraries, or tools, and that may be executed in different multicore architectures. Managing different software configurations and adapt them for different needs is an arduous task, particularly when it has to be carried out by scientists or when dealing with irregular applications.**

**In this paper, we present an approach to abstract legacy software systems using workflow modeling tools. We show how a basic pipeline is modeled and adapted—using model transformations—to different application scenarios, either to obtain better performance, or more reliable results. Moreover, we explain how the system we provide to support the approach is easily extensible to accommodate new tools and algorithms.**

**We show how a pipeline of three irregular applications—all from phylogenetics—is mapped to parallel implementations. Our studies show that the derived programs neither downgrade performance nor sacrifice scalability, even in the presence of a set of asymmetric tasks and when using third-party tools.**

## I. Introduction

In recent years we have observed great advances in parallel platforms and the exponential growth of datasets in several domains (e.g., Physics, Medicine, Chemistry, and Biology). On one hand, we have entered the so-called multicore era [1]. On the other hand, Next-Generation Sequencing (NGS) platforms [2], the Large Hadron Collider accelerator [3], among other platforms, are producing data at an unprecedented scale and pace, which has become known as the *Big Data* problem.

Phylogenetics – the study of evolutionary relationships among a set of taxa – is an extremely important and challenging branch of research in Computational Biology. Nowadays, phylogenies are used in a daily basis and on a wide variety of fields: in linguistics, in forensics, in cancer research and treatment, in drugs research and design, among others [4]. However, due to the aforementioned advances in NGS, phylogenetics also suffers from the Big Data problem. Moreover, the problem is aggravated as the tree search space has a factorial growth, thus it becomes intractable even in the presence of a small set of taxa, as shown in Table I.

Usually, heuristics are used in order to reduce the tree search space. Nevertheless, depending on the size of the dataset, a phylogenetic analysis becomes a daunting task when using a strict sequential computational approach [5], [6]. Thus, to cope with very large datasets, many phylogenetic analyses would greatly benefit from the use of parallel programming (to harness the performance potential of parallel platforms and, therefore, to obtain results as soon as possible).

TABLE I
Number of unrooted binary trees.

| **#Taxa** | **#Trees** |
|---|---|
| $n$ | $(2n-5)!!$ |
| 2 | 1 |
| 3 | 1 |
| 4 | 3 |
| 5 | 15 |
| 10 | 2027025 |
| 20 | 221643095476700000000 |

The alignment of DNA sequences and the process of getting a phylogenetic tree from an alignment are examples of phylogenetic analyses that benefit from the use of parallel computing and, possibly, distributed computing. Fortunately, state of the art tools such as MAFFT [7] and RAxML [8] already provide support for multithreading. Since RAxML provides support for MPI [9], it is also possible to conduct analyses in distributed memory systems. Nevertheless, these tools often do not contemplate different usage scenarios required by scientists, such as: (i) out of the box support for batch processing; (ii) change parallelization strategies either by incorporating new algorithms or by composing existent ones; and (iii) synthesis and reconfiguration of a pipeline of phylogenetic programs (Figure 1 shows a possible pipeline that can be built based on existent phylogenetic programs).

Different application scenarios may require different implementations. For example, the most efficient way to handle one input or a batch of inputs is usually not same. Moreover, the characteristics of the different inputs must be taken into account. The hardware used also affects the design of implementation, so that it can make an optimal use of features such as multiple cores, CPU caches, network, etc. That is, a pipeline can usually be optimized for different inputs, or different target hardware platforms. However, scientists often do not have the required technical knowledge to perform this sort of adaptation. The approach we propose leverages from model transformations, it helps scientists dealing with the aforementioned issues and allows them to abstract from specific details that do not belong to their knowledge domain. It is based on a representation of the application domain that allows the definition of different implementations of common operations, optimized for different needs, which a scientist can choose and compose while synthesizing an optimized implementation for its use case. This approach also allows scientists

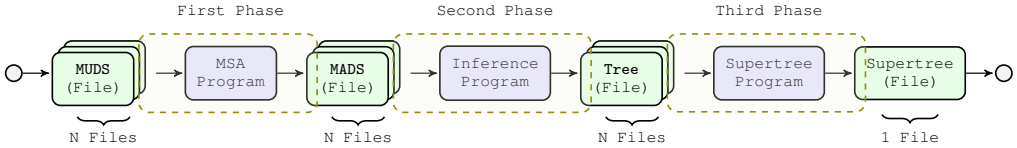



Fig. 1. Pipeline to get a supertree from a set of $N$ multiple unaligned DNA sequences (MUDS) files (MADS means multiple aligned DNA sequences).

to easily switch the external tools that are used to implement each abstract operation of the pipeline. Moreover, the domain representation can be easily extended to accommodate new operations or algorithms.

In summary, the contributions of this paper are:

- abstract legacy software pipelines using workflow modeling tools;
- show how a basic pipeline is incrementally adapted (using model transformations) to different application scenarios, either to obtain better performance, or more reliable results; and
- explain how the approach we propose is easily extensible to accommodate new tools and algorithms that implement the different phases of the pipeline.

As proof of concept, we used our approach to derive parallel implementations of the pipeline shown in Figure 1, which comprises three (legacy) irregular programs, all from phylogenetics (i.e., real world problems). Our studies show that our approach neither downgrades performance nor sacrifices scalability, even in the presence of asymmetric tasks and when using third-party tools. Moreover, it eases the task of specifying and reconfiguring the pipeline.

## II. Overview

Often, programmers have to deal with legacy software systems which may use third-party frameworks, libraries, or tools. Some examples are: in Computer Graphics, Direct3D, and OpenGL; in Computational Chemistry, GROMACS, and NAMD; in Dense Linear Algebra, BLAS, and LAPACK; and so forth. Computational Biology is no different.

In phylogenetics, it is quite common to use a phylogenetic tree to represent the relationships that exist among a set of taxa. The taxa can be formed by a set of multiple unaligned DNA sequences (MUDS). The alignment of MUDS can be performed by a multiple sequence alignment (MSA) tool—in this study we used MAFFT—which output is a set of multiple aligned DNA sequences (MADS), (see First Phase of Figure 1). Then, an inference analysis can be performed over each of those MADS files—in this study we used RAxML—which output is a tree (see Second Phase of Figure 1). Finally, the set of trees—known as input trees or source trees—becomes the input dataset to a supertree method—in this study we used SuperFine [5]—which is used to estimate a supertree on the entire set of taxa (see Third Phase of Figure 1).

The different tools used in the process we described are not integrated, thus the user (e.g., a scientist) has to call each tool, often for several inputs files to produce the desired result. In order to automate this workflow, scripts may be used. More sophisticated solutions—*workflow systems*—have emerged in the past years [10], in order to simplify the task of automating pipelines (and other types of workflows). In this work we explore the use of ReFlO [11] as a tool to allow the specification of pipelines by graphically dragging and connecting components. Moreover, ReFlO provides the ability to encode domain knowledge that can be used to incrementally derive optimized implementations from abstract specifications, which we will leverage to produce a system that gives users the flexibility to adapt applications to their use cases, namely to handle the scheduling and parallelization of multiple inputs.

## III. Synthesis and Reconfiguration of Pipelines

The approach and system we propose are based on the idea of encoding the different operations used in a certain domain (in this case phylogenetics). Those operations are viewed as reusable components, which can be orchestrated to specify a program (its workflow). Moreover, associated with the operations we have different implementations, optimized for specific hardware platforms, or for inputs with certain characteristics. The different implementations for an operation will allow end users to adapt a workflow for its specific use case. Those associations between operations and implementations are called *rewrite rules* in the ReFlO framework [11], and they comprise the model of the domain. There are two types of implementations: *primitives* associate a direct code implementation to an operation (e.g., calling an external tool), whereas *algorithms* specify an implementation of an operation as a composition of operations. It is common to find recursive algorithms that are specified in terms of the operation they implement (e.g., a divide and conquer algorithm that divides the input, by *recursively calling the operation being implemented*, and then merges the results). As we will see later, this is a very useful property of certain algorithms, which allows us to compose different algorithms for the same operation and incrementally derive more complex implementations from a small set of components and rewrite rules.

Given an abstract workflow specification, composed of the operations previously modeled, a user can use the rewrite rules to incrementally *refine* [12] the specification (choosing the most appropriate implementations for its use case), and synthesize an optimized implementation that fits its needs.

In the following we illustrate how we use the proposed approach to interactively model the workflow for the pipeline described in Section II, and synthesize different implementations for it by incrementally refine its abstract specification.

### A. Abstract Pipeline's Workflow

The abstract pipeline we use was described in Section II. It comprises three programs, each one abstracted as an operation. Thus, the base operations we have are `MSA`, `Inference`, and

`Supertree`. The input to these operations is a stream of file paths (representing the files to be processed). Those operations have other additional parameters that specify custom options for each operation.

These are examples of operations we encoded in ReFlO (other operations will be described later). Moreover, we also created: (i) rewrite rules to define possible implementations for the different operations we have, (ii) the source code for each primitive implementation of an operation, and (iii) a simple code generator. These artifacts, together with ReFlO, provide the tool support for the approach we propose, which allows a user to graphically specify workflows, and interactively map the specification to more efficient code for its use case.

The first step in the approach we propose is to specify the workflow's model. This is done creating an architecture model file in ReFlO, then creating an architecture box, and finally adding to it the operations and the connectors that express the desired workflow. In particular, for the pipeline we previously described, we would create the model depicted in Figure 2, which expresses the workflow of a pipeline that processes a given stream of file paths, producing intermediary results, and outputs the resulting file at the end.

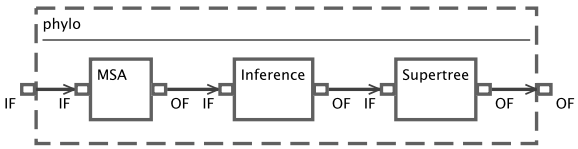



Fig. 2. Abstract pipeline's workflow modeled in ReFlO. Nubs attached to boxes are their input (`IF`) and output (`OF`) ports.

A model like this is all a user needs to obtain the code to execute the pipeline. Even though this is a *platform-independent model (PIM)*—as the operations present in the workflow are *abstract* (i.e., they do not force a particular implementation)—, we have associated to each operation a default (primitive) implementation, which is automatically chosen when generating code for the workflow.

The code generator walks through the workflow and generates Java code for each connector and box. Each connector is translated to an object of type `BlockingQueue<T>`, where `T` is given by the data type of the ports that uses the connector. By using blocking queues, when running the program, the thread running a box will automatically block while the queue is empty (i.e., when there are no elements to be processed). Each box is implemented by a class (it should extend class `Thread`, have a **constructor** that receives the connector objects and a `Config` object with custom options loaded from a file, and have its behavior defined in its `run` method). The first connector is filled with the values given as arguments to the program. Thus, when generating the code we just have to instantiate a class for each box (providing them the right connector objects), and add them to a list of threads, which are started at the end. As an example, Listing 1 provides an excerpt of the code generated for the workflow show in Figure 2.

```
// array containing threads for different boxes
List<Thread> threads = new ArrayList<>();
// connectors variables
BlockingQueue<String> q0 = new LinkedBlockingQueue<>();
BlockingQueue<String> q1 = new LinkedBlockingQueue<>();
BlockingQueue<String> q2 = new LinkedBlockingQueue<>();
BlockingQueue<String> q3 = new LinkedBlockingQueue<>();
Config confis = Config.load();
// fill first connector with program arguments
for(String arg : args) q0.put(arg);
q0.put(""); // empty string means no more args
// create a thread to run each box
threads.add(new MSA(q0, q1, configs));
threads.add(new Inference(q1, q2, configs));
threads.add(new Supertree(q2, q3, configs));
threads.forEach(Thread::start); // start threads, and
threads.forEach(Thread::join); // wait for their finish
```

Listing 1. An excerpt of code generated for workflow from Figure 2.

```
public class Inference extends Thread {
   private BlockingQueue<String> IF;
   private BlockingQueue<String> OF;
   private Config configs;

   public Inference(BlockingQueue<String> IF,
        BlockingQueue<String> OF, Config configs) {
      this.IF = IF;
      this.OF = OF;
      this.configs = configs;
   }

   public void run() {
      // take values from input queues,
      // process each value, and
      // write results to output queues
   }
}
```

Listing 2. Basic structure for a box's code implementation.

In Listing 2, we show the structure of a class implementing a box (in this case, the `Inference` box). A constructor receives the objects (queues) associated to each port, and stores them in proper attributes of the class. The `run` method takes values from the input queue(s), process each of those values, and write results into the output queue(s). Typically the users reuse components already implemented. However, in case he wants to extend the system to support new operations and implementations, he can graphically add new rewrite rules to the system specifying those new components. He will also have to provide the code for new primitive implementations. Nevertheless, our system automatically generates a class template similar to the one presented in Listing 2, where the user only has to fill in the code that reads and processes inputs (usually calling an external tool).

### B. Parallelizing the Pipeline

The previous workflow (see Figure 2) uses the default implementations for each operation, which result in sequential code that essentially calls some external tools and processes one file at a time. However, we can easily derive different implementations by refining the initial workflow presented in Figure 2. We can, for example, refine the `Inference` operation with an implementation that uses FastTree [13] instead of RAxML (RAxML is the inference tool used by default) for the tree inference phase (see *Second Phase* in Figure 1), which is faster but provides less options than RAxML. This is done selecting the `Inference` box (Figure 2), choosing

the *refine* transformation from the context menu, and selecting `inf_ft` from the list of implementations shown. ReFlO will automatically generate the workflow depicted in Figure 3, where the `Inference` operation was replaced with the `inf_ft` primitive implementation.

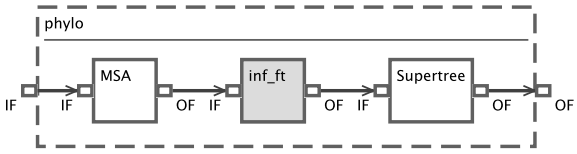



Fig. 3. Workflow after refining the `Inference` operation with its `inf_ft` primitive implementation.

More important than being able to switch primitive implementations is the ability to choose—and compose—algorithm implementations. For example, suppose the user has a multicore machine. He can run his workflows faster by using multiple threads to run an operation. Considering that the primitive implementations rely on tools such as MAFFT, FastTree, RAxML, etc., which have multithreading capabilities, this new parallelization step can be obtained by simply activating this option. Again, this alternate behavior will be obtained refining the initial workflow (Figure 2).

The `Inference` operation has two steps. First, given a file, a *task* that defines what to do with the file is created. This task also contains properties about the file, such as its size, among others. Then, the task is sent to a generic *executor* component, that runs it and outputs its resulting file. In its simplest form, the executor just runs tasks in a *first come, first served* basis. To allow the `Inference` operation to be parallelized, we start by refining the workflow to expose these two steps, as shown in Figure 4.

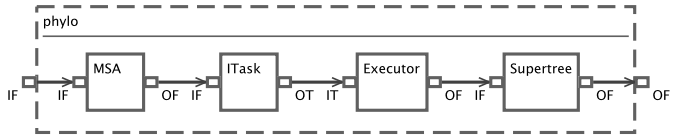



Fig. 4. Workflow after exposing the two steps of applying the `Inference` operation to a file.

Parallelization itself can now be obtained by refining the `Executor` operation with its multithreaded implementation, to explore *intra-task* parallelism. This refinement replaces the `Executor` with the composition `MTParam+Executor`. In practice, we just added a box before the `Executor` to determine the optimal number of threads that should be used when processing each task, and to set the proper task's property, as shown in Figure 5.

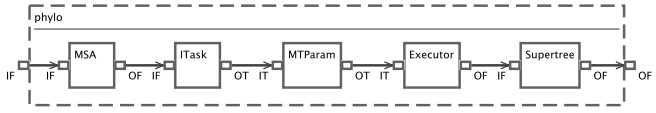



Fig. 5. Workflow after adding multithreading support to the `Inference` operation.

Multithreading offers limited scalability (in the case of FastTree the limit is 3 threads, but regardless of the tool, from a certain point, performance downgrades as more threads are added). This problem can be lowered by also processing more than one file in parallel (in case the operation can process each file individually). Thus, for example, the `Executor` operation can be refined again to also support a fork-join parallelization strategy (enabling support for *inter-task* parallelism). Such implementation creates `N` parallel executors to which a scheduler operation sends tasks to be processed, and that are merged at the end, as shown in Figure 6. The `Merge` operation sends feedback information—through `FB` port—about finished tasks to scheduler (`MPSchd`) so that it can optimize load balancing in the presence of irregular tasks, whose execution time is hard or impractical to estimate.

This workflow was obtained simply by applying another refinement to the `Executor` operation, that is, composing two different refinements for this operation. In this way, we obtained support for *intra-* and *inter-task* parallelism at the same time (which we call *hybrid* parallelism). Besides the two parallel workflows described, the user could have also refined the initial workflow (Figure 2) to support inter-task parallelism only. Thus, by composing the multithreading and multiprocess transformations in different ways, he can obtain different combinations of parallel strategies. This happens as the algorithms used are recursive. Moreover, at the end he can also choose the external tool to use, or change its list of arguments in order to be able to perform alternative analyses, by refining the `ITask` operation.

There are additional levels of parallelization we can support. For example, we could add support for distributed memory parallelization, simply starting by refining the `Executor` with an algorithm similar to the one used to distribute tasks among processes, but that would distribute different sets of tasks among different machines. To make such an algorithm available, all that is needed is to incorporate new adequate rewrite rules associating the algorithm with the `Executor` operation, and add implementations for any new operation required by the algorithm.

Similar algorithms can be used to parallelize the `MSA` operation. In fact, only the first refinement would change, to expose an operation that creates alignment tasks (instead of inference tasks), which are then sent to the `Executor`. As we use a generic `Executor` component that handles task objects (which abstract the specificities of each phase of the pipeline), parallelization would be obtained with the same refinements of `Executor`. Those new refinement steps would seamlessly compose with the previously mentioned ones.

For the `Supertree` operation, we use SuperFine to estimate a supertree from the source trees—the output of the previous operation of the pipeline—and, then, refine that same supertree (i.e., try to resolve each polytomy—an internal node that has a degree greater than 3—, as described in [5]). Thus, SuperFine is used to refine the Supertree operation. This refinement of the model comprises different steps, as shown in Figure 7. First, we have `SCMT_PMR` operation, which: (i) estimates a supertree, on the entire set of taxa of the given source trees, by applying the strict consensus merger (SCM) algorithm, and (ii) generates a matrix representation

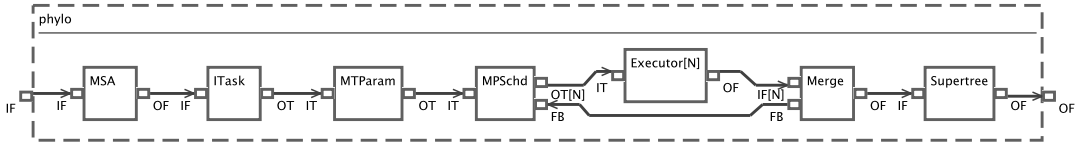



Fig. 6. Workflow after adding multiprocess support to the `Inference` operation. `N` is a variable that expresses how many times boxes or ports are replicated.

file per each polytomy present in the estimated supertree. Then, the `Inference` operation is called to process each matrix representation file (i.e., to run an inference over the matrix representation of each polytomy), producing a (new) tree per each matrix representation file. Finally, the `RPT` operation replaces each polytomy with its counterpart (new) tree. As parallelization refinement of the `Supertree` operation, we use the parallelization described in [6], which basically parallelizes the refinement phase of SuperFine (`Inference` operation). Therefore, the parallelization of the `Supertree` operation is done reusing the `Inference` refinements previously described.

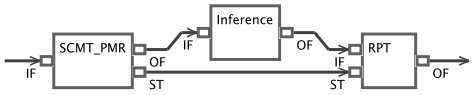



Fig. 7. Workflow resulting from refining `Supertree` operation (due to space limitations, we show only the boxes that replaced the `Supertree` operation).

## IV. Performance Results

### A. Experimental Design

We used in our evaluations one computing node at Stampede [14] supercomputer. A Stampede's computing node has two eight-core Xeon E5-2680 (2.27 GHz) processors, is configured with 32GB of memory, and runs CentOS release 6.5 (Final). We used MAFFT 7.017, RAxML 8.0.22, and SuperFine. MAFFT and RAxML were compiled using gcc 4.7.1 with -O3 optimization flag. RAxML was compiled with support for AVX, and with support for AVX and Pthreads. We used Python 2.7.3 EPD 7.3-2 (64-bit) to run SuperFine. We used JDK 1.8.0_20 (64-bit) to implement the system that allows to synthesize and reconfigure pipelines. We used the following biological data sets:

- Eukaryote, 22 files with multiple unaligned DNA sequences, studied originally in [15];
- CPL (Comprehensive Papilionoid Legumes), 2228 taxa, 39 source trees, studied originally in [16];
- Marsupials, 267 taxa, 158 source trees, studied originally in [17];
- Placental Mammals, 116 taxa, 726 source trees, studied originally in [18];
- Seabirds, 121 taxa, 7 source trees, studied originally in [19]; and
- THPL (Temperate Herbaceous Papilionoid Legumes), 558 taxa, 19 source trees, studied originally in [20].

Finally, we took the average running time of six runs for each program/thread-count/dataset combination.

### B. MSA Operation

Figure 8 shows the achieved speedups of four `MSA` programs, each of those programs was synthesized using the described system. Moreover, each of them corresponds to a derivation for the `MSA` operation, that was described earlier and depicted in Figure 2, and executed using as input the Eukaryote dataset. The programs are: (i) a derivation that executes in sequential mode (**Seq**); (ii) a derivation that exploits intra-task parallelism (**Intra-Par**), which rely on the ability of MAFFT to be executed using its support for multithreading (i.e., tasks are executed one after the other, but each may be executed with a number of threads that goes from one to the number of cores); (iii) a derivation that exploits inter-task parallelism (**Inter-Par**) (i.e., several task may be executed in parallel, which means parallel calls to MAFFT); and (iv) a derivation that exploits inter and intra-task parallelism (**Hybrid-Par**), that is a composition of ii and iii.

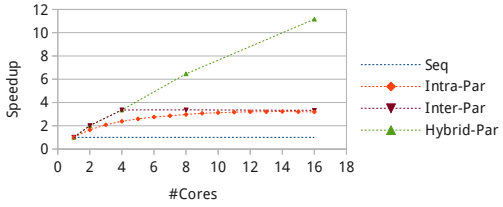



Fig. 8. Speedups of four `MSA` programs which input was the Eukaryote dataset.

As shown in Figure 8, the **Hybrid-Par** derivation outperforms by far any other derivation and exhibits an efficiency level of roughly 70%. On the contrary, **Inter-Par** and **Intra-Par** derivations exhibit poor performance, yet they outperform the **Seq** derivation, as expected. It is important to notice that MAFFT by itself (see **Intra-Par** speedups in Figure 8) is not able to cope efficiently with the Eukaryote dataset and, therefore, with no other.

### C. Inference Operation

We also synthesized four `Inference` programs, in the same way that we did to evaluate the `MSA` operation (see Section IV-B). For the sake of brevity, we decided to show only—in Table II—results for the hybrid derivation (i.e., the `Inference` program that exploits inter and intra-task parallelism).

In phylogenetics it is well known that, in general, inference operations are more time-consuming with increasing number of taxa, which is easily deduced from numbers shown in Table I. Thus, it is expected that running an inference analysis over an alignment file that has more species than another one would take longer to perform. As well as what happens with increasing number of species, the increasing number of sites—the length of DNA sequence—also contributes to a more time-consuming inference operation. These facts may be used to establish a metric, although not perfect, that allows to quantify the size of each task. We elide here any further discussion

TABLE II
AVERAGE RUNNING TIMES (ART, IN SECONDS) FOR THE HYBRID DERIVATION OF INFERENCE OPERATION USING THE EUKARYOTE DATASET (#T IS THE NUMBER OF THREADS USED TO PROCESS A TASK).

| | | | #Cores | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | Taxa | | 1 | | 2 | | 4 | | 8 | | 16 | |
| File | #Species | #Sites | #T | ART | #T | ART | #T | ART | #T | ART | #T | ART |
| 12S_Asco | 1314 | 470 | 1 | 99.4 | 1 | 99.4 | 1 | 99.3 | 1 | 99.3 | 1 | 100.9 |
| 16S_H | 752 | 331 | 1 | 55.3 | 1 | 55.2 | 1 | 55.2 | 1 | 55.2 | 1 | 64.1 |
| LSU_P1 | 11580 | 238 | 1 | 786.1 | 1 | 787.2 | 1 | 786.2 | 1 | 787.8 | 1 | 844.6 |
| LSU_P10 | 1919 | 112 | 1 | 163.1 | 1 | 163.2 | 1 | 163.1 | 1 | 164.1 | 1 | 182.7 |
| LSU_P12 | 1723 | 180 | 1 | 77.7 | 1 | 77.5 | 1 | 77.6 | 1 | 77.6 | 1 | 82.2 |
| LSU_P13 | 1267 | 240 | 1 | 75.7 | 1 | 75.7 | 1 | 75.7 | 1 | 75.6 | 1 | 92.6 |
| LSU_P2 | 11700 | 274 | 1 | 1287.5 | 1 | 1285.2 | 1 | 1285.1 | 1 | 1287.6 | 1 | 1324.2 |
| LSU_P3 | 8357 | 121 | 1 | 678.1 | 1 | 679.8 | 1 | 676.3 | 1 | 682.3 | 1 | 708.1 |
| LSU_P4 | 6445 | 308 | 1 | 839.7 | 1 | 841.0 | 1 | 839.1 | 1 | 839.0 | 1 | 876.9 |
| LSU_P5 | 3579 | 115 | 1 | 159.8 | 1 | 160.2 | 1 | 159.5 | 1 | 160.0 | 1 | 174.7 |
| LSU_P7 | 2021 | 118 | 1 | 68.6 | 1 | 68.7 | 1 | 68.6 | 1 | 68.7 | 1 | 78.0 |
| LSU_P8 | 2017 | 145 | 1 | 102.1 | 1 | 101.9 | 1 | 102.0 | 1 | 101.9 | 1 | 111.4 |
| LSU_P9 | 1954 | 173 | 1 | 84.1 | 1 | 83.9 | 1 | 83.9 | 1 | 83.9 | 1 | 100.7 |
| MAT_K | 11855 | 842 | 1 | 1669.4 | 1 | 1669.8 | 1 | 1668.3 | 1 | 1669.7 | 1 | 1763.8 |
| NADH | 4864 | 1169 | 1 | 909.7 | 1 | 909.1 | 1 | 908.9 | 1 | 908.0 | 1 | 958.7 |
| RBCL | 13043 | 1710 | 1 | 3087.8 | 1 | 3081.7 | 1 | 3111.9 | 2 | 2122.7 | 3 | 1645.2 |
| SSU_1a | 20462 | 488 | 1 | 3759.1 | 1 | 3758.5 | 1 | 3806.1 | 2 | 2996.2 | 4 | 2829.9 |
| SSU_1b | 20439 | 364 | 1 | 4849.8 | 1 | 4838.7 | 1 | 4891.9 | 2 | 3728.1 | 3 | 3154.8 |
| SSU_3C | 19599 | 172 | 1 | 1916.6 | 1 | 1916.1 | 1 | 1930.3 | 1 | 2028.1 | 1 | 2134.3 |
| SSU_4a | 19552 | 135 | 1 | 1238.3 | 1 | 1239.1 | 1 | 1240.0 | 1 | 1248.6 | 1 | 1368.0 |
| SSU_4b | 19336 | 250 | 1 | 2182.8 | 1 | 2183.9 | 1 | 2215.3 | 1 | 2289.1 | 2 | 1488.4 |
| SSU_4X | 19377 | 33 | 1 | 640.9 | 1 | 640.7 | 1 | 641.7 | 1 | 652.4 | 1 | 681.3 |
| | | **ART** | | **24731.5** | | **12370.2** | | **6286.0** | | **4053.7** | | **3154.9** |

about this topic—how to measure the size of a task?—since it is out of the scope of this paper. With the size of each task it is possible to decide how many threads should be used to process a task, which is achieved through the `MTParam` operation. The multiprocessing scheduler—`MPSchd`—may revise the number of threads to optimize it when there are other processes/tasks running in parallel (see Section III-B). The number of threads per task also depends on the number of cores.

As shown in Table II, with 8 cores some tasks are executed with more than one thread, namely: RBCL, SSU_1a, and SSU_1b. The attribution may evolve as long as the number of cores increases. As it is possible to observe in all tasks that were processed with more than one thread, there is no evident pay-off of such attribution. Nevertheless, the exploitation of intra-task parallelism also contributed to obtain smaller running times with 8 and 16 cores. It still deserves to be mentioned that the running time obtained when using 16 cores is limited by the time that is required to process SSU_1b task.

TABLE III
SPEEDUPS FOR THE FOR THE HYBRID DERIVATION OF INFERENCE OPERATION USING THE EUKARYOTE DATASET.

| | #Cores | | | | |
|---|---|---|---|---|---|
| | **1** | **2** | **4** | **8** | **16** |
| **Speedup** | 1.0 | 2.0 | 3.9 | 6.1 | 7.8 |

We now focus our attention to the speedups obtained with the hybrid derivation, shown in Table III. The results show increasing speedups as cores are added, even though the efficiency starts to decrease from a certain number of cores (somewhere between 4 and 8 cores). Efficiency will gradually diminish due to the lack of pay-off while exploiting intra-task parallelism. Solving this issue is not easy since, in this case, we rely on the ability of RAxML to exploit intra-task parallelism. All in all, these results corroborate that it is possible to synthesize efficient pipelines with our system, even when one has to rely on third-party tools.

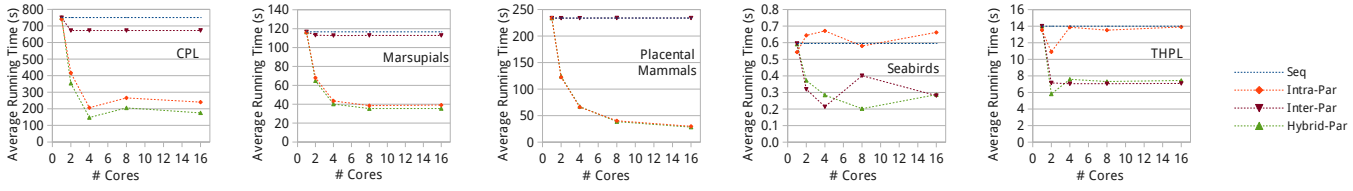
### *D. Supertree Operation*

We also synthesized four `Supertree` programs, in the same way that we did to evaluate the `MSA` operation and the `Inference` operation (see, respectively, Section IV-B and Section IV-C). To evaluate the performance of `Supertree` operation we decided to use not the Eukaryote dataset but the CPL, the Marsupials, the Placental Mammals, the Seabirds, and the THPL datasets. These datasets have very different characteristics, as mentioned in Section IV-A, and, above all, they are much smaller than Eukaryote dataset. Moreover, those very different characteristics also applies to the set of polytomies that each consensus tree—the estimated supertree—has, as shown in Table IV. Thus, they are optimal to test how well a program that was synthesized with our system behaves with such datasets. In Figure 9, we show the results for the sequential derivation (**Seq**), the intra derivation (**Intra-Par**), inter derivation (**Inter-Par**), and the hybrid derivation (**Hybrid-Par**).

TABLE IV
OVERVIEW OF THE POLYTOMIES OF EACH CONSENSUS TREE.

| | | CPL | Marsupials | Pla. Mam. | Seabirds | THPL |
|---|---|---|---|---|---|---|
| **# Polytomies** | | 105 | 18 | 1 | 10 | 36 |
| **Degree** | **Minimum** | 3 | 3 | 114 | 4 | 3 |
| | **Maximum** | 531 | 199 | 114 | 12 | 94 |
| | **Sum** | 1287 | 273 | 114 | 71 | 312 |
| | **Mode** | 3 | 3 | - | 6 | 3 |
| | **Median** | 4 | 4 | 114 | 6 - 7 | 4 |
| | **Mean** | 12.3 | 15.2 | 114.0 | 7.1 | 8.7 |

The first point to notice from results shown in Figure 9 is that the program obtained from the hybrid derivation has, in general, the best performance when compared with the performance of each other. The second point to notice is that the program obtained from the hybrid derivation exhibits good scalability since its behavior does not suffer with very different problem sizes. To behave well in a wide variety of conditions is an extremely important fact that contributes for the success of a program. The third point to notice is that the program obtained from the hybrid derivation, which simply reuses the refinements required for the intra and inter derivations, has the ability to take the best of each derivation. This fact is corroborated by the achieved results with datasets that are so different one from another, as aforementioned (see Section IV-A and Table IV). The CPL, the Marsupials, and the Placental Mammals datasets have asymmetric sets of tasks. Typically, these datasets have one very large polytomy which takes much longer to process than any other of the same dataset (i.e., this sort of polytomies dominate the running time). This kind of datasets are more suited to exploit intra-task parallelism with success. On the contrary, datasets like the Seabirds and the THPL dataset, since there is more equilibrium in their tasks, are more suited to exploit inter-task parallelism with success. Since the program obtained from




(a) (b) (c) (d) (e)

Fig. 9. Average running times of four `Supertree` programs using several biological datasets.

the hybrid derivation is prepared to exploit inter and intra-task parallelism, it behaves well either in the presence of symmetric or asymmetric sets of tasks.

We should also refer that the characteristics of the mentioned datasets as well the implementation of RAxML are responsible for some decrease in performance (see Figures 9(a), 9(d), and 9(e)). For instance, the Seabirds dataset has only ten polytomies, each of those is very small and the respective task requires less than a second to be computed. In the case of CPL dataset the observed behavior is due to RAxML, which is corroborated by the observed behavior of the **Intra-Par** derivation (when running in **Intra-Par** mode the polytomies are processed one after the other).

## V. Related Work

Workflow systems for scientific usage are gaining more and more popularity nowadays. Well-known graphical systems include SciRun [21], Kepler [22], Triana [23], VisTrails [24], or Taverna [25]. Deelman *et al.* provide a survey comparing different workflow systems [10]. Some of these systems support workflow optimizations based on traditional compiler techniques, however they offer limited support for user defined transformations, namely to specify how abstract components can be refined for specific usage scenarios. In our work we relied on ReFlO [11], due to its support for transformations, which adds to the ease of use of graphical tools the flexibility needed to obtain efficient implementations.

We also provide an approach to incrementally parallelize applications through transformations. We follow ideas originally advocated by *knowledge-based software engineering* [26], and recently recovered by the *Design by Transformation* approach [27], which promote the use of transformations to map a specification to an efficient implementation. The ability to switch components' implementation was also explored by the RISPP approach [28]. Other approaches promoting the incremental addition of parallelism relied on Aspect-Oriented Programming (e.g. [29], [30]). Systems such as Spiral [31] or Build to Order BLAS [32] support the automatic generation of efficient low-level kernel functions, where empirical search is employed to choose the best implementation. However, they require regular application domains, where performance can be predicted, either using cost functions or empirical search, which is not feasible in domains such as phylogenetics.

Algorithmic skeletons [33], which can express the structure of common patterns used in parallel programming [34], [35], have also been used to parallelize applications. A survey on the use of algorithmic skeletons for parallel programming is presented in [36]. These methodologies/frameworks raise the level of abstraction, and remove parallelization concerns from domain code. However, they have limited support for the addition of domain-specific composition patterns (i.e., their extensibility is limited). On the other hand, ReFlO is a system designed to allow users to extend the set of rewrite rules that can be used to refine workflows, providing users the ability to add support for custom composition patterns.

## VI. Conclusions and Future Work

The use of computational tools to help scientists in their research is a reality in science nowadays. However, scientists often rely on multiple external computational tools, which need to be orchestrated together, to solve their problems. This task becomes harder as the complexity of the problems grow, requiring more and more sophisticated solutions to obtain results in a timely manner. The ability to reuse tools and libraries is important to handle this problem, but it is equally important to provide reusable strategies to reconfigure and optimize generic specifications. In this paper, we have shown that the ability for a system to synthesize—either through refinement, composition, or both—and reconfigure a workflow has many benefits regarding the specification and adaptation of scientific pipelines, namely to support different parallelization strategies. We followed an MDE approach, based on the use of high-level models to specify workflows, and reusable model transformations to map those models to efficient implementations and to generate code. The approach we propose allows users to shift the focus from the technological issues of orchestrating multiple tools to the research problems they have at hands by handling the composition of multiple tools and the batch processing of files, and by mechanizing the transformation of workflows to support parallelization and the use of specific tools. Moreover, the system developed can be easily extended to accommodate new algorithms, tools, or parallelization strategies, for example, which can be shared and reused among the users of the domain. That is, our approach provides both performance and flexibility. It can be broadly applied to other domains, not necessarily scientific ones.

We showed how the same parallel strategies are reused multiple times with multiple programs in the same workflow. Moreover, we showed the importance of recursive algorithms to allow the generation of implementations using multiple parallelization strategies by composing different combinations of a small set of refinements. Thus, the proposed approach and respective system are worth to be adopted, as corroborated by the performance results that were achieved with

irregular applications and with a variety of datasets. Datasets that generate asymmetric sets of tasks turn even harder the exploitation of parallelism. Nonetheless, we have shown that the proposed approach and respective system are capable to cope with such datasets, in a transparent and efficient manner. Moreover, we have shown that the underlying third-party tools when used alone are not prepared to handle datasets—either with symmetric or asymmetric sets of tasks—as well as our approach and systems are.

We are currently developing support for MPI, in order to enable the use of multiple machines, in this way increasing the levels of parallelism supported, and coping with the growth of datasets. Moreover, we are also planning to provide optimization transformations to allow the execution of multiple phases of the pipeline simultaneously (currently, operations such as scheduling impose a barrier that requires the previous phase to finish before the next can start). We are also enriching the set of refinements we provide, in order to add support for other features such as checkpointing, which is becoming more important as we use more and more computational resources that are feed with bigger and bigger sets of tasks (which increase the probability of occurring failures).

## ACKNOWLEDGMENTS

This research was partially supported by Fundação para a Ciência e a Tecnologia (grants: SFRH/BD/42634/2007, and SFRH/BD/47800/2008). We thank João Luís Sobral, Rui Silva, Don Batory and Tandy Warnow for fruitful discussions and valuable feedback. We thank Keshav Pingali for his valuable support and sponsorship to execute jobs on TACC machines. We are very grateful to the anonymous reviewers for the evaluation of our paper and for the constructive critics.

## REFERENCES

[1] X.-H. Sun and Y. Chen, "Reevaluating Amdahl's Law in the Multicore Era," *J. Parallel Distrib. Comput.*, vol. 70, no. 2, pp. 183–188, 2010.

[2] E. R. Mardis, "Next-generation sequencing platforms," *Annu. Rev. Anal. Chem.*, vol. 6, no. 1, pp. 287–303, 2013.

[3] P. Messina, "Challenges of the lhc: the computing challenge," in *Prestigious Discoveries at CERN*. Springer Berlin Heidelberg, 2003, pp. 117–133.

[4] Z. Yang and B. Rannala, "Molecular phylogenetics: principles and practice," *Nat. Rev. Gen.*, vol. 13, no. 5, pp. 303–314, 2012.

[5] M. S. Swenson, R. Suri, C. R. Linder, and T. Warnow, "SuperFine: Fast and Accurate Supertree Estimation," *Syst. Biol.*, 2011.

[6] D. T. Neves, T. Warnow, J. L. Sobral, and K. Pingali, "Parallelizing superfine," in *SAC*, 2012.

[7] K. Katoh and D. M. Standley, "MAFFT Multiple Sequence Alignment Software Version 7: Improvements in Performance and Usability," *Mol. Biol. Evol.*, vol. 30, no. 4, pp. 772–780, 2013.

[8] A. Stamatakis, "RAxML Version 8: A tool for Phylogenetic Analysis and Post-Analysis of Large Phylogenies," *Bioinformatics*, 2014.

[9] M. Snir, S. W. Otto, D. W. Walker, J. Dongarra, and S. Huss-Lederman, *MPI: The Complete Reference*. MIT Press, 1995.

[10] E. Deelman, D. Gannon, M. Shields, and I. Taylor, "Workflows and e-science: An overview of workflow system features and capabilities," *Future Gener. Comput. Syst.*, vol. 25, no. 5, pp. 528–540, 2009.

[11] R. C. Gonçalves, D. Batory, and J. L. Sobral, "ReFlO: An interactive tool for pipe-and-filter domain specification and program generation," *Software & Systems Modeling*, 2014.

[12] N. Wirth, "Program development by stepwise refinement," *Commun. ACM*, vol. 14, no. 4, pp. 221–227, 1971.

[13] M. N. Price, P. S. Dehal, and A. P. Arkin, "FastTree 2 – Approximately Maximum-Likelihood Trees for Large Alignments," *PLoS ONE*, vol. 5, no. 3, 2010.

[14] Texas Advanced Computing Center (TACC), The University of Texas at Austin. [Online]. Available: http://www.tacc.utexas.edu/

[15] P. A. Goloboff, S. A. Catalano, J. Marcos Mirande, C. A. Szumik, J. Salvador Arias, M. Källersjö, and J. S. Farris, "Phylogenetic analysis of 73 060 taxa corroborates major eukaryotic groups," *Cladistics*, vol. 25, no. 3, pp. 211–230, 2009.

[16] M. McMahon and M. Sanderson, "Phylogenetic Supermatrix Analysis of GenBank Sequences from 2228 Papilionoid Legumes," *Syst. Biol.*, vol. 55, no. 5, pp. 818–836, 2006.

[17] M. Cardillo, O. R. P. Bininda-Emonds, E. Boakes, and A. Purvis, "A species-level phylogenetic supertree of marsupials," *J. Zool.*, vol. 264, pp. 11–31, 2004.

[18] O. R. P. Bininda-Emonds, M. Cardillo, K. E. Jones, R. D. E. MacPhee, R. M. D. Beck, R. Grenyer, S. A. Price, R. A. Vos, J. L. Gittleman, and A. Purvis, "The delayed rise of present-day mammals," *Nature*, vol. 446, pp. 507–512, 2007.

[19] M. Kennedy and R. D. M. Page, "Seabird supertrees: combining partial estimates of procellariiform phylogeny," *The Auk*, vol. 119, pp. 88–108, 2002.

[20] M. Wojciechowski, M. Sanderson, K. Steele, and A. Liston, "Molecular phylogeny of the "temperate herbaceous tribes" of papilionoid legumes: a supertree approach," *Adv. Legume Syst.*, vol. 9, pp. 277–298, 2000.

[21] S. Parker, M. Miller, C. Hansen, and C. Johnson, "An integrated problem solving environment: the scirun computational steering system," in *HICSS*, 1998.

[22] I. Altintas, C. Berkley, E. Jaeger, M. Jones, B. Ludascher, and S. Mock, "Kepler: An extensible system for design and execution of scientific workflows," in *SSDBM*, 2004.

[23] I. Taylor, M. Shields, I. Wang, and A. Harrison, "Visual grid workflow in triana," *Journal of Grid Computing*, vol. 3, no. 3–4, pp. 153–169, 2006.

[24] S. P. Callahan, J. Freire, E. Santos, C. E. Scheidegger, C. T. Silva, and H. T. Vo, "Managing the evolution of dataflows with vistrails," in *ICDEW*, 2006.

[25] K. Wolstencroft, R. Haines, D. Fellows, A. Williams, D. Withers, S. Owen, S. Soiland-Reyes, I. Dunlop, A. Nenadic, P. Fisher, J. Bhagat, K. Belhajjame, F. Bacall, A. Hardisty, A. Nieva de la Hidalga, M. P. Balcazar Vargas, S. Sufi, and C. Goble, "The taverna workflow suite: designing and executing workflows of web services on the desktop, web or in the cloud," *Nucleic Acids Res.*, vol. 41, no. W1, pp. 557–561, 2013.

[26] C. Green, D. Luckham, R. Balzer, T. Cheatham, and C. Rich, "Report on a knowledge-based software assistant," Kestrel Institute, Tech. Rep., 1983.

[27] D. Batory, R. C. Gonçalves, B. Marker, and J. Siegmund, "Dark knowledge and graph grammars in automated software design," in *SLE*, 2013.

[28] L. Bauer, M. Shafique, S. Kramer, and J. Henkel, "Rispp: Rotating instruction set processing platform," in *DAC*, 2007.

[29] B. Harbulot and J. R. Gurd, "Using AspectJ to separate concerns in parallel scientific java code," in *AOSD*, 2004.

[30] C. A. Cunha, J. L. Sobral, and M. P. Monteiro, "Reusable aspect-oriented implementations of concurrency patterns and mechanisms," in *AOSD*, 2006.

[31] M. Püschel, J. M. F. Moura, B. Singer, J. Xiong, J. Johnson, D. Padua, M. Veloso, and R. W. Johnson, "Spiral: A generator for platform-adapted libraries of signal processing algorithms," *Int. J. High Perform. Comput. Appl.*, vol. 18, no. 1, pp. 21–45, 2004.

[32] G. Belter, E. R. Jessup, I. Karlin, and J. G. Siek, "Automating the generation of composed linear algebra kernels," in *SC*, 2009.

[33] M. Cole, *Algorithmic skeletons: structured management of parallel computation*. MIT Press, 1991.

[34] J. Darlington, A. J. Field, P. G. Harrison, P. H. J. Kelly, D. W. N. Sharp, Q. Wu, and R. L. While, "Parallel programming using skeleton functions," in *PARLE*, 1993.

[35] D. T. Neves and J. L. Sobral, "Improving the separation of parallel code in skeletal systems," in *ISPDC*, 2009.

[36] H. González-Vélez and M. Leyton, "A survey of algorithmic skeleton frameworks: high-level structured parallel programming enablers," *Softw. Pract. Exper.*, vol. 40, no. 12, pp. 1135–1160, 2010.