Multilevel Task Parallelism Exploitation on Asymmetric Sets of Tasks and When Using Third-Party Tools

Diogo Telmo Neves
Department of Informatics, University of Minho
Campus of Gualtar, Braga, Portugal
Email: dneves@di.uminho.pt

***Abstract*—To harness the performance potential of current multicore processors, a multitude of algorithms, frameworks and libraries have been developed. Nevertheless, it is still extremely difficult to take advantage of the full potential of multicore processors. Moreover, in the presence of asymmetric sets of tasks, and/or when using third-party tools, this problem would only aggravate.**

In this paper, we present a software design and two algorithms to enable multilevel task parallelism exploitation on asymmetric sets of tasks and when using third-party tools. The proposed software design along with both algorithms allow, in a seamlessly way, the efficient exploitation of coarse-grained—inter-task—and fine-grained—intra-task—parallelism. Thus, it becomes possible to make a better and transparent usage of the performance potential of current multicore processors on shared-memory systems.

To assess the feasibility and the benefit of applying the proposed software design along with both algorithms, we used three irregular applications from Phylogenetics—real world problems—, and several input data sets with different characteristics. Our studies show groundbreaking results in terms of the achieved speedups and that scalability is not impaired.

I. Introduction

Applications are often written in a sequential fashion without regard to take full advantage of hardware resources such as multicore and manycore parallel platforms. To improve programming productivity and the performance of applications programmers have to rely on: different libraries (e.g. OpenMP [1], MPI [2], TBB [3], Pthreads [4], CUDA [5], and OpenCL [6]); frameworks (e.g. Galois [7], and YaSkel [8]); or parallel programming languages (e.g. Cilk [9], Charm++ [10], X10 [11], and Chapel [12]). Moreover, it is well-known that parallel algorithms are often fundamentally different from their sequential counterparts [13]. Thus, exploiting parallelism is a hard and complex task, which is usually done by domain experts that have to handle several details, such as: particularities of a given problem and respective domain; parallel programming issues; and specificities of different architectures. This difficult and challenging task becomes even harder when, for a given problem, one has to cope with third-party tools. Phylogenetics—the study of evolutionary relationships among a set of taxa—is a good example where all these dimensions may arise together within a single problem [14][15][16][17][18][19][20][21].

The use of third-party tools is recurrent in phylogenetics, like it is in many other domains (e.g., biochemistry, biophysics, and earthquake engineering). For instance, the use of an accurate and fast multiple sequence alignment (MSA) tool (e.g., MAFFT [18]) is crucial to perform specific phylogenetic analyses, otherwise the results of such analyses would be meaningless and, most likely, each analysis would not complete in a reasonable amount of time. Nowadays, many third-party tools provide support for multithreading and, eventually, for other complex schemes of parallelism exploitation. MAFFT, FastTree [19] and RAxML [20] are examples of such tools, which are suited to perform phylogenetic analyses. However, taking advantage of such support for multithreading is not always a straight-ahead decision since it often depends on the target platform, the input data set—a collection of data—, among other details. Moreover, in irregular applications—programs that manipulate pointer-based data structures, such as graphs and trees—the amount of parallelism is not predictable at compile time—it is very input dependent [22]—and may be subject of considerable variations at runtime [23]. Unfortunately, neither traditional programming models nor compilers are sufficient to leverage the complexity of exploiting all levels of parallelism in such applications [24][25][22].

Exploiting inter-task and intra-task parallelism is a common way to improve the performance of irregular applications [24][22]. Usually, these types of parallelism are exploited and fully controlled within the same application. In some cases, the former may be exploited by one application—the main application—,while the latter may be exploited by a different application—a third-party tool—, which gets called by the main application. We are deeply interested in this last scenario since, to the best of our knowledge, there is no work covering the exploitation of multilevel parallelism on such scenario. Hence, the motivation for this work.

In this paper, we present:

- a software design, shown in Figure 3, that allows to abstract the size of a task, which is estimated at runtime;
- Algorithm 1, which allows to estimate the number of threads per task; and
- Algorithm 2, a scheduling algorithm that relies on Algorithm 1.

978-1-4673-7148-3/15 $31.00 © 2015 IEEE
DOI 10.1109/ISPDC.2015.13

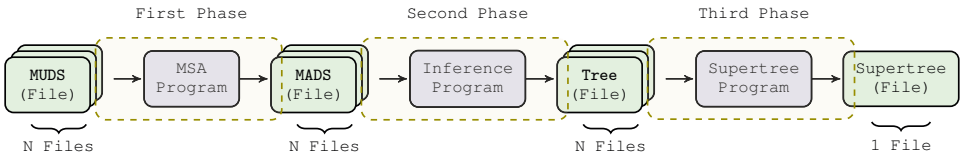


Fig. 1. A computational workflow to get a supertree from a set of N MUDS files.

The proposed software design along with both algorithms allow, in a seamlessly way, the efficient exploitation of coarse-grained—inter-task—and fine-grained—intra-task—parallelism, including on asymmetric sets of tasks and when using third-party tools. Thus, it becomes possible to make a better and transparent usage of the performance potential of current multicore processors on shared-memory systems.

To assess the feasibility and the benefit of applying the proposed software design along with both algorithms, we used three irregular applications from Phylogenetics—real world problems—, and several input data sets with different characteristics. Those applications can be used, for instance, in a computational workflow [26] that allows to get a supertree—a tree on the full set of taxa—from a set of files with multiple unaligned DNA sequences (MUDS), as shown in Figure 1 (MADS means multiple aligned DNA sequences). Specifically, the examples are: (i) the alignment of multiple DNA sequences (see first phase shown in Figure 1); (ii) the inference of gene trees (see second phase shown in Figure 1); and (iii) the supertree refinement operation [27][28][29] (see third phase shown in Figure 1). Besides the use of real-world applications, we also decided to use biological data sets to show the validity of the proposed software design along with both algorithms when applied to real-world problems with real data. Our studies show groundbreaking results in terms of the achieved speedups and that scalability is not impaired.

II. Problem Characterization

The exploitation of multilevel parallelism has been studied in the past. For instance, it was described in [30][31][32], and more recently in [22] and [33]. However, to the best of our knowledge, there is no work describing the exploitation of such parallelism on asymmetric sets of tasks and when using third-party tools.

The use of third-party tools turns this problem—multilevel parallelism exploitation—much harder since, in most cases, one has to treat third-party tools as black boxes. As an example, each phase shown in Figure 1 uses a third-party tool. Those tools are treated as building blocks (i.e., components) of the computational workflow shown in Figure 1. Therefore, exploiting parallelism in this type of problems is much harder since one does not fully control such exploitation. A radical alternative would be to fully implement the computational workflow shown in Figure 1, but that is profoundly discouraged since: it would become a daunting task; and, worse, one hardly would come up with a competitive tool either in terms of performance or accuracy.

This problem—multilevel parallelism exploitation—aggravates when the input data set generates an asymmetric set of tasks, and when it is difficult, if not impossible, to estimate precisely the cost to process each task. In many problems, including complex ones (e.g., some graph problems [22]), there is no need to perform cost estimation since the cost to process each task is roughly the same. However, this does not necessarily holds for all problems, which is the case of the computational workflow shown in Figure 1. Within this workflow there are three types of tasks: (i) on the first phase, a task relies on MAFFT to align a set of unaligned DNA sequences; (ii) on the second phase, a task relies on RAxML to produce a gene tree from an alignment of DNA sequences; and (iii) on the third phase, a task relies on FastTree to refine each polytomy[1] found in the (estimated) supertree. The tasks of a phase must be executed before any of the following phases. This sort of dependency is solved by using a barrier at the end of each phase [26].

In phylogenetics, data sets are usually composed by a set of (input) files, which, typically, have the same format (e.g., PHYLIP, FASTA, NEXUS, or Newick). As an example, the Eukaryote data set (studied originally in [34]) is composed by 22 files, each in the PHYLIP format and containing hundreds or thousands of species and respective DNA sequences (each sequence is composed by a number of sites, also known as characters). Table I shows the characterization of the Eukaryote data set.

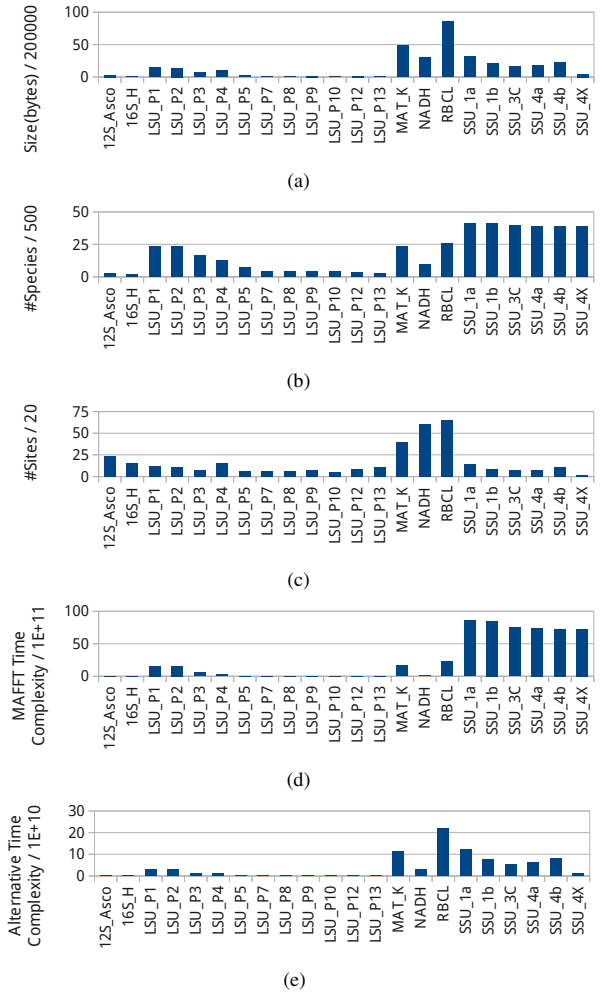
TABLE I
Eukaryote data set characterization (the file size is in bytes).

File		Taxa		File		Taxa	
Name	Size	#Species	#Sites	Name	Size	#Species	#Sites
12S_Asco	639808	1314	464	LSU_P12	328861	1723	169
16S_H	252254	752	314	LSU_P13	308854	1267	222
LSU_P1	3015948	11580	238	MAT_K	9649882	11855	792
LSU_P2	2776756	11700	215	NADH	5987809	4864	1209
LSU_P3	1422898	8357	148	RBCL	17193159	13043	1296
LSU_P4	2153641	6445	312	SSU_1a	6451254	20462	293
LSU_P5	492667	3579	116	SSU_1b	4154619	20439	181
LSU_P7	277052	2021	115	SSU_3C	3199294	19599	141
LSU_P8	306411	2017	130	SSU_4a	3504693	19552	157
LSU_P9	347477	1954	156	SSU_4b	4606681	19336	216
LSU_P10	247169	1919	107	SSU_4X	934486	19377	26

In many cases, the asymmetry of a data set generates an asymmetric set of tasks, meaning that some tasks take longer to perform than other tasks. Moreover, when heuristics are used—which happens to be the case of each phylogenetic application used throughout this paper—tasks of same size take different times to complete. One way to handle efficiently with asymmetric data sets is through the decomposition of the initial data set into smaller chunks of data. This strategy is applied with success in a wide variety of problems (e.g., in numerical linear algebra [35], or in graph partitioning [36]).

[1]A polytomy is an internal node which degree—given by the number of edges connected to the node—is higher than 3.

However, it can not be applied to many other problems due to specific restrictions. For instance, it makes no sense to split the larger file—RBCL—of Eukaryote data set into smaller files since the data of that file is related, as it is the data of any other file. Thus, the Eukaryote data set is asymmetric, either in terms of the file size, the number of species (#Species), or the number of sites (#Sites) that each file has (see Table I). Depending on the degree of asymmetry and on the size of the input data set, it may become impossible to reach a reasonable well balanced workload. Nevertheless, it becomes crucial to determine how many threads should each task use to achieve the best load balancing possible. However, quantifying this parameter—number of threads per task—is not an easy decision, since it is highly dependent on the problem itself and on the input data set. Moreover, when relying on third-party tools to exploit intra-task parallelism this decision becomes harder to reach. For all these reasons, there is no smart load balancer that is capable to cope with all these details for all domains.



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

Fig. 2. Eukaryote data set characterization.

The characterization of each specific problem becomes inevitable to estimate, as precisely as possible, the size of each task. That characterization is achieved by analyzing the data set in different dimensions. For instance, Figure 2 shows different metrics to estimate the size of each task for the same problem—MSA—using the Eukaryote data set (see Table I). In Figure 2(d), the size of each task is estimated based on MAFFT's time complexity—when input sequences are highly conserved, which is given by $O(NL) + O(N^3)$, where N is the number of species and L is the number of sites (i.e., the alignment length)—, with this view it seems that: the SSU_1a and SSU_1b files should be processed with more threads than any other; and the remaining SSU files should also be processed with more threads than the remaining files. An alternative time complexity—$O(N^2L)$ (using the same notation)—to estimate the size of each task is shown in Figure 2(e), with this view it seems that the RCBL file should be processed with more threads than any other file.

III. Multilevel Parallelism Exploitation

As seen, exploiting multilevel parallelism on asymmetric sets of tasks and when using third-party tools is highly dependent on the selected metric that is used to estimate the size of each task. To solve this kind of problems we propose: (i) a software design, shown in Figure 3, that allows to abstract the size of a task, which is estimated at runtime; (ii) Algorithm 1, which allows to estimate the number of threads per task; and (iii) Algorithm 2, a scheduling algorithm that relies on Algorithm 1.

A. Software Design

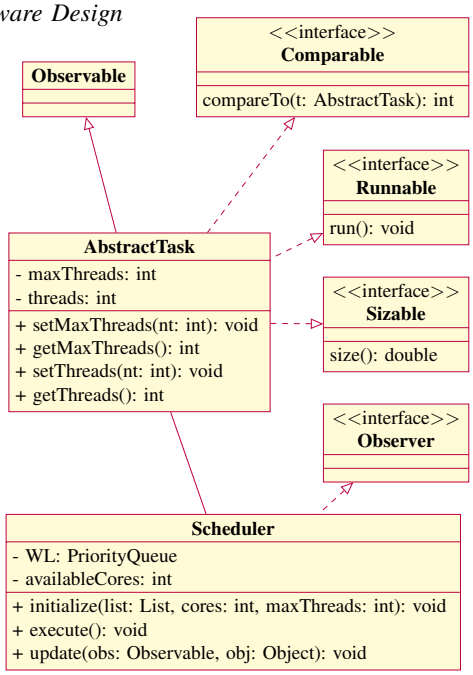


Fig. 3. Simplified UML class diagram of proposed software design.

The proposed software design (see Figure 3) is grounded on the size of a task (i.e., on interface `Sizable`). The `AbstractTask` class does not implement the `size` method. Thus, the proposed software design is agnostic about the

implementation of the `size` method and, therefore, can be applied to any problem. As an example, when using MAFFT to solve a MSA problem, a concrete implementation of `size` method could be based on a time complexity given by MAFFT, for instance, $O(NL) + O(N^3)$ (see Listing 2).

The proposed software design relies on Observer Pattern [37], however, it can be easily implemented using a different programming model (e.g., MPI [2]). The scheduler—an instance of `Scheduler` class—observes each task of its worklist—`WL` (we use the concept of *worklist* in the same way that it is used in [22][23][25]). Each task—an instance of `AbstractTask` class—upon completion notifies its observer (i.e., the scheduler), this happens when the `run` method finishes its execution.

```
public abstract class FASTATask
   extends AbstractTask {
  private File fasta;        // the input file
  protected int species;     // N
  protected int sites;       // L
  ...
  // returns the command used to run this task
  public abstract String command();
}
```

Listing 1. Implementation excerpt of `FASTATask` class.

```
public class MAFFTTask extends FASTATask {
  ...
  public double size() {  // O(NL) + O(N^3)
    return species * sites
       + species * species * species;
  }
}
```

Listing 2. Implementation excerpt of `MAFFTTask` class.

B. Number of Threads per Task Algorithm

The size of a task (see Section III-A) is extremely important to compute the weight that the task has on the entire `WL` of the scheduler. The weight would then enable to set the number of threads per task, as shown in Algorithm 1.

Algorithm 1: Number of Threads per Task.

Data: ***list*** $= \{t_0, \ldots, t_i, \ldots, t_{N-1}\}$ such that:
$0 \leq i \leq N-1, |list| = N \wedge N \geq 1$; and
t_i is the i^{th} element—task—of the list.
cores such that: $1 \leq cores \leq$ total number of cores.
maxThreads such that: $1 \leq maxThreads \leq cores$.

```
1  Procedure setNumberOfThreads(
         List list, int cores, int maxThreads)
2      double totalSize ← 0.0
3      for task in list do
4          totalSize ← totalSize + task.size()
5      for task in list do
6          double weight ← task.size()/totalSize
7          int threads ← round(cores * weight)
8          threads ← max(1, threads)
9          threads ← min(maxThreads, threads)
10         task.setThreads(threads)
```

In Algorithm 1, the first loop (line 3) iterates over the elements (i.e., the tasks) of the input `list` to compute the sum of the size of each individual task. Then, in the second loop (line 5), the weight of a task is computed (line 6), which in turn is used to compute the number of threads that will be used when processing the task. In line 8, Algorithm 1 guarantees that at least one thread (i.e., one core) will be used to process the task. The maximum number of threads—a limit that may be useful, for instance, due to a third-party tool limitation—is controlled in line 9. Finally, it is set the number of threads of the task (line 10).

Algorithm 1 could be improved using a lookup table to avoid repeat computations for elements of the input `list` that have equal size. For the sake of simplicity, we decided to not incorporate this improvement here, though it is highly recommended.

C. Scheduling Algorithm

As we have seen, in parallel computing it is necessary to decide, wisely, how tasks get assigned to cores (i.e., how many threads should be used to process each task). In irregular applications that is a hard decision since the amount of parallelism is not predictable at compile time and may be subject of considerable variations at runtime [23]. To complicate even more that decision, in order to get an acceptable solution within a reasonable amount of time heuristics are used to solve some problems (e.g. Maximum Likelihood implemented in [19], [20], or Maximum Parsimony implemented in [21]), which means that, most likely, tasks of the same size may significantly differ in the amount of time required for completion. Each of these facts turn extremely difficult the job of a scheduler, no matter the selected load balancing algorithm. However, things can get even more complicated when it is inevitable to launch processes by calling an external tool which in turn provides support for multithreading. Once the external process gets launched there is no chance for a scheduler to readapt (i.e., reconfigure) task execution.

To help to mitigate this problem we propose the scheduling algorithm shown in Algorithm 2. The `initialize` procedure (line 3) is called by the runtime system (i.e., the main thread) to perform the needed initialization. Algorithm 2 relies on Algorithm 1 (line 4) to set the numbers of threads per task of the input `list`. In line 5, all elements of the input `list` are added to the worklist—`WL`—of the scheduler. The `WL` is a priority queue, where each element—task, an instance of `AbstractTask`—is sorted in a descending manner according to its `size`. The `execute` procedure (line 7) is called, *for the first time*, by the runtime system. Then, for each iteration of the while loop (line 8): (i) one `AbstractTask` get removed from the worklist—`WL`—(line 9); (ii) the number of available cores—`availableCores`—is decreased according to the (estimated) number of threads that is required to process the task (line 10); (iii) an observer—the scheduler—is added to the task to be processed (line 11); (iv) the task is executed—implies spawning a thread—(line 12); and (v) these steps—from line 9 to line 12—get repeated while the `WL` is not empty

and there is at least one core available. When each task finishes its execution it will notify its observer—the scheduler—, this action will make the `update` procedure (line 13) to get called. The `update` procedure will increase the number of available cores (line 15) and, extremely important, it will call the `execute` procedure (line 16).

Algorithm 2: Scheduling Algorithm.

Data: ***list*** $= \{t_0, \ldots, t_i, \ldots, t_{N-1}\}$ such that:
$0 \leq i \leq N-1, |list| = N \wedge N \geq 1$, and
t_i is the i^{th} element—task—of the list.
cores such that: $1 \leq cores \leq$ total number of cores.
maxThreads such that: $1 \leq maxThreads \leq cores$.

```
1  PriorityQueue WL                          // the worklist
2  double availableCores

3  Procedure initialize(List list, int cores, int maxThreads)
4      setNumberOfThreads(list, cores, maxThreads)
5      WL.addAll(list)
6      availableCores ← cores

7  Procedure execute()
8      while (not WL.isEmpty() and availableCores > 0) do
9          AbstractTask task ← WL.poll()
10         availableCores ←
               availableCores − task.getThreads()
           // add this scheduler as the observer
11         task.addObserver(self)
12         task.run()

13 Procedure update(Observable obs, Object obj)
14     AbstractTask task ← (AbstractTask) obs
15     availableCores ← availableCores+task.getThreads()
16     execute()
```

Algorithm 2 enables the exploitation of inter-task—coarse-grained—parallelism and intra-task—fine-grained—parallelism, at the same time. However, due to the implicit use of Algorithm 1, Algorithm 2 gives preference to the exploitation of inter-task—coarse-grained—parallelism. Thus, one should expect that with fewer cores intra-task—fine-grained—parallelism may end not being exploited.

IV. Results

A. Experimental Design

We used in our evaluations one computing node at Stampede supercomputer [38]. 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 PAUP* 4.0b10 [21], MAFFT 7.017, RAxML 8.0.22, FastTree 2.1.7, and SuperFine [27]. MAFFT, RAxML and FastTree 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. FastTree was compiled with support for SSE3, and with support for SSE3 and OpenMP. 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 three programs:

- **MSA Program**: a program to perform multiple sequence alignment, which calls MAFFT—a third-party tool;
- **Gene Trees Program**: a program to perform inference of gene trees, which calls RAxML—a third-party tool; and
- **Supertree Refinement Program**: a program to refine a supertree, which uses SuperFine and calls FastTree—a third-party tool.

All these programs can be executed in the following modes:

- **sequential**: no parallelism get exploited;
- **intra-parallel**: intra-task parallelism get exploited, the number of cores is given through a command line argument, tasks are processed one after the other, and each task is processed with a number of threads that is equals to the number of cores;
- **inter-parallel**: inter-task parallelism get exploited, the number of cores is given through a command line argument, tasks are processed in parallel, larger tasks are processed in first place, each task is processed with one thread; and
- **hybrid-parallel**: inter-task and intra-task parallelism get exploited, Algorithm 2 and, implicitly, Algorithm 1 are used, the number of cores and the maximum number of threads per task are given through command line arguments.

In our studies, we used the following biological data sets:

- Eukaryote, 22 files (as an example, the largest file—RBCL—has 13043 taxa), studied originally in [34];
- CPL (Comprehensive Papilionoid Legumes), 2228 taxa, 39 source trees, studied originally in [39];
- Marsupials, 267 taxa, 158 source trees, studied originally in [40];
- Placental Mammals, 116 taxa, 726 source trees, studied originally in [41];
- Seabirds, 121 taxa, 7 source trees, studied originally in [42]; and
- THPL (Temperate Herbaceous Papilionoid Legumes), 558 taxa, 19 source trees, studied originally in [43].

We used the 1000-taxon simulated data set, studied originally in [27], to measure the accuracy of results.

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

B. Performance of MSA Program

The MSA Program takes as input a set of files—the input data set—, each with multiple unaligned DNA sequences (MUDS), and generates a set of files with multiple aligned DNA sequences (MADS), one MADS file per each MUDS file. To generate a MADS file from a MUDS file a task of the MSA Program calls MAFFT—a third-party tool—to perform the multiple sequence alignment operation.

We measured the running times of MSA Program when executed in intra-parallel mode (i.e., while exploiting intra-task parallelism), for the Eukaryote data set as input and using 1 to 16 threads/cores, results are shown in Table II. The performance of running the MSA Program in intra-parallel

TABLE II
RUNNING TIMES (IN SECONDS) OF MSA PROGRAM WHEN EXECUTED IN INTRA-PARALLEL MODE USING THE EUKARYOTE DATA SET AS INPUT.

File	Taxa		#Threads/Cores															
Name	Species	Sites	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
12S_Asco	1314	464	5.1	3.8	3.0	2.7	2.5	2.5	2.4	2.4	2.4	2.4	2.4	2.5	2.5	2.5	2.5	2.4
16S_H	752	314	2.1	1.7	1.3	1.1	1.0	1.0	1.0	1.0	1.0	1.0	1.1	1.1	1.1	1.1	1.1	1.2
LSU_P1	11580	238	40.5	27.8	22.8	20.2	18.8	17.9	17.1	16.4	16.1	15.9	15.7	15.4	15.4	15.4	15.3	15.3
LSU_P2	11700	215	41.8	28.3	23.5	20.8	19.4	18.2	17.5	17.0	16.5	16.4	16.0	15.8	15.8	15.7	15.5	15.6
LSU_P3	8357	148	33.0	24.7	23.5	26.2	26.7	27.7	28.0	28.3	25.9	27.2	27.1	28.5	28.5	26.2	27.5	27.7
LSU_P4	6445	312	40.0	28.4	23.2	21.8	22.5	21.5	22.6	22.2	20.6	21.3	21.9	21.2	21.8	22.4	22.5	22.2
LSU_P5	3579	116	6.2	5.4	5.3	5.2	5.7	5.7	5.8	5.3	5.6	5.1	5.5	5.3	5.4	5.3	5.6	5.5
LSU_P7	2021	115	2.9	2.6	2.3	2.4	2.4	2.4	2.5	2.5	2.5	2.5	2.5	2.5	2.6	2.6	2.6	2.6
LSU_P8	2017	130	3.5	3.1	2.8	2.8	2.9	2.9	2.9	2.9	3.0	3.0	2.9	3.1	3.0	3.1	3.1	3.2
LSU_P9	1954	156	4.3	4.3	4.0	3.9	4.0	4.1	4.0	3.9	3.9	3.9	3.9	3.8	3.9	3.8	3.8	3.9
LSU_P10	1919	107	2.4	2.2	2.1	2.1	2.2	2.1	2.1	2.1	2.2	2.2	2.2	2.2	2.2	2.2	2.2	2.2
LSU_P12	1723	169	3.2	3.0	2.6	2.5	2.5	2.5	2.5	2.4	2.4	2.4	2.5	2.4	2.5	2.5	2.6	2.5
LSU_P13	1267	222	2.9	2.7	2.4	2.3	2.2	2.2	2.3	2.2	2.3	2.3	2.3	2.4	2.5	2.5	2.5	2.6
MAT_K	11855	792	193.3	105.7	75.3	60.1	51.0	45.1	40.9	37.5	35.1	33.5	32.2	31.4	31.4	31.3	31.3	31.7
NADH	4864	1209	125.0	76.6	57.3	47.4	41.8	39.0	36.4	35.0	34.4	32.8	32.8	31.9	32.8	34.2	35.1	35.7
RBCL	13043	1296	455.2	237.7	167.1	130.4	109.8	98.0	88.4	80.2	75.4	71.1	67.3	64.5	62.8	61.3	61.4	61.8
SSU_1a	20462	293	164.1	105.7	86.0	76.6	70.5	66.6	63.9	61.8	60.1	59.0	58.6	57.5	57.0	56.4	56.8	56.2
SSU_1b	20439	181	113.5	82.3	71.8	66.2	63.0	61.0	59.5	58.2	57.5	57.0	56.4	56.0	55.8	55.7	55.6	55.9
SSU_3C	19599	141	86.1	63.7	55.4	51.3	48.6	47.0	46.0	45.2	44.7	44.0	43.7	43.5	43.4	43.4	43.1	43.4
SSU_4a	19552	157	55.2	39.5	34.7	32.3	30.8	29.9	29.2	29.0	28.6	28.6	28.3	28.2	28.4	28.6	28.7	29.0
SSU_4b	19336	216	87.3	60.5	52.0	47.6	45.0	43.3	42.8	41.4	40.6	40.2	40.0	39.6	39.3	39.2	39.4	39.0
SSU_4X	19377	26	30.6	26.2	24.7	23.8	23.7	23.5	23.3	23.4	23.5	23.6	23.5	23.8	23.6	23.4	24.3	24.5
	Running Time (s)		1498.2	935.9	743.1	649.7	597.0	564.1	541.1	520.3	504.3	495.4	488.8	482.6	481.7	478.8	482.5	484.1

mode with one core is barely the same as running the MSA Program in sequential mode, meaning that the overhead of running with support for multithreading is negligible. We elide to show here the results of such comparison since they bring little (to none) contribution to this discussion. The first point to notice is that the performance of MSA Program benefits from taking advantage of multithreading support provided by MAFFT and, therefore, intra-task parallelism should be exploited. The second point to notice is that with 5 cores the efficiency is roughly 50%, therefore, at a certain point—6 or more cores—there is no evident pay-off on keep increasing the number of threads/cores. The third point to notice is that the observed running times do not follow MAFFT's time complexity—$O(NL) + O(N^3)$ (when input sequences are highly conserved) to $O(NL^2) + O(N^3)$ (when the similarity among input sequences is weak).

TABLE III
RUNNING TIMES (IN SECONDS) OF MSA PROGRAM WHEN EXECUTED IN INTER-PARALLEL MODE USING THE EUKARYOTE DATA SET AS INPUT.

# Cores	1	2	4	8	16
Running Time (s)	1508.7	761.0	453.0	466.8	455.2

Then, we measured the running times of MSA Program when executed in inter-parallel mode (i.e., while exploiting inter-task parallelism), for the Eukaryote data set as input and using 1, 2, 4, 8, and 16 cores, results are shown in Table III. The first point to notice is that the performance of MSA Program benefits from taking advantage of exploiting inter-task parallelism, and, therefore, inter-task parallelism should be exploited. The second point to notice is that the running times when using 4, 8, or 16 cores are roughly the same. Therefore, from a certain point—between 5 and 8 cores—there is no evident pay-off on keep increasing the number of cores. The explanation for this behavior—good to excellent performance with fewer cores (2-4) and no increasing gain in performance with more cores (8-16)—is simple and grounded in two facts: (i) with increasing number of cores there are fewer tasks per core (for instance, with 16 cores there is a ratio of $22/16$ tasks per core, too few); and (ii) the larger task limits the performance improvement (in this case, the time required to align all sequences of RBCL file becomes the performance limit—see Table II, RCBL file, 1 thread/core).

With all this information in our hands, we customized the implementation of `size` method, based on an alternative time complexity—$O(N^2L)$—, as shown in Listing 3.

```
public class MAFFTTask extends FASTATask {
  ...
  public double size() {  // O(N^2 L)
    return species * species * sites;
  }
}
```

Listing 3. Implementation excerpt of `MAFFTTask` class.

The customized definition of the size of each task enables to set the number of threads per task as shown in Table IV. Though not perfect, with increasing number of cores this attribution of threads per task allows to execute the largest task—the task that is responsible to process the RCBL file—with more threads/cores. This is an important decision since, as aforementioned, when using the Eukaryote data set as input, the performance of MSA Program is bounded by the time that is required to process the larger task (i.e., the RCBL file).

Then, we measured the running times of MSA Program when executed in hybrid-parallel mode (i.e., while exploiting

TABLE IV
ESTIMATION OF NUMBER OF THREADS/CORES PER TASK, DONE BY ALGORITHM 1, FOR THE EUKARYOTE DATA SET (WEIGHT IS IN %).

Task		Threads/Cores					Task		Threads/Cores				
File	Weight	1	2	4	8	16	File	Weight	1	2	4	8	16
12S_Asco	0.1	1	1	1	1	1	LSU_P12	0.1	1	1	1	1	1
16S_H	0.0	1	1	1	1	1	LSU_P13	0.0	1	1	1	1	1
LSU_P1	3.7	1	1	1	1	1	MAT_K	13.0	1	1	1	1	2
LSU_P2	3.4	1	1	1	1	1	NADH	3.4	1	1	1	1	1
LSU_P3	1.2	1	1	1	1	1	RBCL	25.8	1	1	1	2	4
LSU_P4	1.5	1	1	1	1	1	SSU_1a	14.4	1	1	1	1	2
LSU_P5	0.2	1	1	1	1	1	SSU_1b	8.9	1	1	1	1	1
LSU_P7	0.1	1	1	1	1	1	SSU_3C	6.3	1	1	1	1	1
LSU_P8	0.1	1	1	1	1	1	SSU_4a	7.0	1	1	1	1	1
LSU_P9	0.1	1	1	1	1	1	SSU_4b	9.5	1	1	1	1	2
LSU_P10	0.0	1	1	1	1	1	SSU_4X	1.1	1	1	1	1	1

inter-task and intra-task parallelism at the same time), for the Eukaryote data set as input and using 1, 2, 4, 8, and 16 cores, results are shown in Table V. We recall, that Algorithm 2 and, implicitly, Algorithm 1 and the size of a task are only used when executing the programs (in this case, the MSA Program) in hybrid-parallel mode.

TABLE V
RUNNING TIMES (IN SECONDS) OF MSA PROGRAM WHEN EXECUTED IN HYBRID-PARALLEL MODE, USING THE EUKARYOTE DATA SET AS INPUT.

# Cores	1	2	4	8	16
Running Time (s)	1505.9	768.9	454.1	240.0	141.1

The first point to notice is that the MSA Program when executed in inter-parallel or hybrid-parallel mode is faster than when executed in intra-parallel mode. The second point to notice is that the performance of MSA Program when executed in inter-parallel or hybrid-parallel mode, using 1, 2, or 4 cores, is barely the same. The third point to notice, and probably the most important, is that when the MSA Program is executed in hybrid-parallel mode using 8 or 16 cores it greatly benefits from the exploitation of intra-task parallelism (see Table IV) at the same time that inter-task parallelism is exploited. This benefit is due to the use of Algorithm 2 and, implicitly, Algorithm 1.

Figure 4 shows the speedups of MSA Program when executed in all possible modes, using the Eukaryote data set as input. As it is possible to see, the efficiency of the MSA Program when executed in hybrid-parallel mode is: 100% when using 2 cores; higher than 85% when using 4 cores; higher than 75% when using 8 cores; and higher than 65% when using 16 cores.

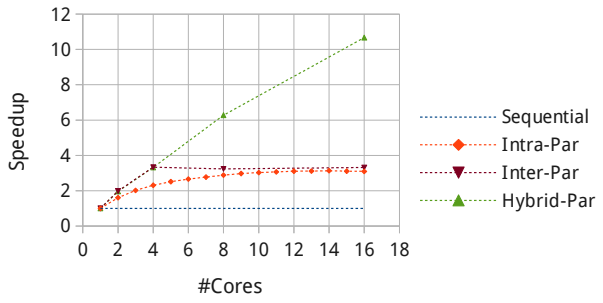


Fig. 4. Speedups of MSA program, using the Eukaryote data set as input.

C. Performance of Gene Trees Program

The Gene Trees Program takes as input a set of files—the input data set—, each with multiple aligned DNA sequences (MADS), and generates a gene tree (i.e., a phylogeny) from each file. To generate a gene tree from a MADS file, a task of the Gene Trees Program calls RAxML—a third-party tool—to perform inference over the MADS file.

Since RAxML does not provide any time complexity analysis of its algorithms, the implementation of the `size` method (see Listing 4) was based on a regression analysis that we made using the running times that were obtained when executing the Gene Trees Program in intra-parallel mode.

```
public class RAxMLTask extends FASTATask {
  ...
  public double size() {  // O(N^3.5 log(N) L^1.1)
    return Math.pow(species, 3.5)
       * Math.log(species) / Math.log(2)
          * Math.pow(sites, 1.1);
  }
}
```

Listing 4. Implementation excerpt of `RAxMLTask` class.

The definition of the `size` method shown in Listing 4 is useful when executing the Gene Trees Program in hybrid-parallel mode. Essentially, it enables to set the number of threads per task, as shown in Table VI.

TABLE VI
RUNNING TIMES (RT, IN SECONDS) OF GENE TREES PROGRAM WHEN EXECUTED IN HYBRID-PARALLEL MODE USING THE EUKARYOTE DATA SET AS INPUT (#T IS THE NUMBER OF THREADS TO PROCESS A TASK).

			#Cores									
	Taxa		1		2		4		8		16	
File	#Species	#Sites	#T	RT	#T	RT	#T	RT	#T	RT	#T	RT
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
	Running Time(RT)			**24731.5**		**12370.2**		**6286.0**		**4053.7**		**3154.9**

Then, we measured the running times of Gene Trees Program when executed in hybrid-parallel mode (i.e., while exploiting inter-task and intra-task parallelism at the same time), for the Eukaryote data set as input and using 1, 2, 4, 8, and 16 cores, results are shown in Table VI. We recall, again, that Algorithm 2 and, implicitly, Algorithm 1 and the size of a

task are only used when executing the programs (in this case, the Gene Trees Program) in hybrid-parallel mode.

The first point to notice is that, for the experimented configurations, the running time decreases as long the number of cores increases. The second point to notice is that when moving from 8 to 16 cores: the running time required to process the task that is used to process the SSU_1a file was reduced only by roughly 5% despite the number of threads has been duplicated; and the running time required to process the SSU_1b file limits the performance gain. The third point to notice, and probably the most important, is that when the Gene Trees Program is executed in hybrid-parallel mode using 8 or 16 cores it greatly benefits from the exploitation of intra-task parallelism at the same time that inter-task parallelism is exploited. This benefit is due to the use of Algorithm 2 and, implicitly, Algorithm 1.

TABLE VII
SPEEDUPS OF GENE TREES PROGRAM WHEN EXECUTED IN HYBRID-PARALLEL MODE FOR THE EUKARYOTE DATA SET AS INPUT AND USING 1, 2, 4, 8, AND 16 CORES.

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

Table VII shows the speedups of Gene Trees Program when executed in hybrid-parallel mode for the Eukaryote data set as input and using 1, 2, 4, 8, and 16 cores. As it is possible to see, the efficiency of the Gene Trees Program when executed in hybrid-parallel mode is: 100% when using 2 cores; roughly 98% when using 4 cores; higher than 75% when using 8 cores; and roughly 49% when using 16 cores.

Finally, for the sake of brevity and due to space limitations, we decided to show only results when executing the Gene Trees Program in hybrid-parallel mode.

D. Performance of Supertree Refinement Program

The Supertree Refinement Program takes as input a supertree and tries to refine each polytomy that the supertree has (for more details see [27][28][29]). A supertree is the result of amalgamating a set of smaller trees (know as the source trees), with overlapping sets of labelled leaves, into a single tree on the full set of taxa. Refining a polytomy implies running an inference operation over a data file that represents that polytomy. Thus, to run an inference operation a task of the Supertree Refinement Program calls FastTree—a third-party tool. We used SuperFine [27] to generate supertrees for the following data sets: CPL; Marsupials, Placental Mammals, Seabirds, and THPL. An overview of the polytomies that each supertree has is shown in Table VIII.

TABLE VIII
POLYTOMIES OVERVIEW PER ESTIMATED SUPERTREE.

		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
	Median	4	4	114	6-7	4
	Mean	12.3	15.2	114.0	7.1	8.7

The implementation of the `size` method was based on the time complexity of FastTree, as shown in Listing 5.

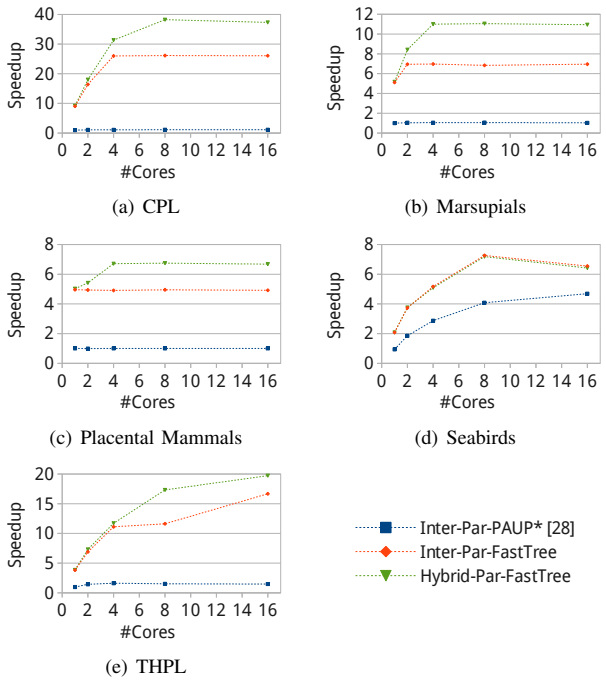
```
public class FastTreeTask extends FASTATask {
  ...
  public double size() {  // O(N^1.5 log(N) L)
    return Math.pow(species, 1.5)
       * Math.log(species) / Math.log(2) * L;
  }
}
```

Listing 5. Implementation excerpt of `FastTreeTask` class.

For the sake of brevity and due to space limitations, we decided to show only speedups that were obtained when executing the Supertree Refinement Program:

- in inter-parallel mode and using PAUP*—a third-party tool—as the inference tool (see Inter-Par-PAUP* results in Figure 5), which corresponds to the parallelization of SuperFine described in [28];
- in inter-parallel mode and using FastTree—a third-party tool—as the inference tool (see Inter-Par-FastTree results in Figure 5); and
- in hybrid-parallel mode and using FastTree as the inference tool (see Hybrid-Par-FastTree results in Figure 5), this setup uses Algorithm 2 and, implicitly, Algorithm 1.

The speedups of Supertree Refinement Program shown in Figure 5 are relatively to the running time of the baseline implementation [27].



(a) CPL (b) Marsupials (c) Placental Mammals (d) Seabirds (e) THPL

Fig. 5. Speedups relatively to the baseline implementation [27].

The first point to notice is that the Supertree Refinement Program when executed in hybrid parallel mode and using FastTree as the inference tool—Hybrid-Par-FastTree—outperforms any other setup, no matter the data set. The

second point to notice is that the Hybrid-Par-FastTree setup exhibits a good scalability, it is important to recall that the data sets used in this study have different characteristics (see Section IV-A and Table VIII). The third point to notice, and probably the most important, is the magnitude of the achieved speedups when using the Hybrid-Par-FastTree setup. As an example, on the CPL data set the Hybrid-Par-FastTree setup is more than 38X faster than the baseline implementation [27] when using 8 cores, and more than 35X faster than the Inter-Par-PAUP* setup (i.e., the parallel implementation described in [28]). In other words, on the CPL data set the hybrid parallelization enables to go from more than 700 seconds to less than 20 seconds (we do not show running times due to space limitations).

The results of the Hybrid-Par-FastTree setup, despite being excellent, are restricted due to the intrinsic limitation of FastTree parallelization—based on three OpenMP parallel sections. This fact imposes a limit on the number of threads—3 at a maximum—that can be used to process a task while exploiting intra-task parallelism. This limitation is evident on Marsupials and Placental Mammals data sets, and noticed on the CPL data set.

V. Related Work

In [30], Eldred *et al.* show that exploiting a single type of parallelism has clear performance limitations. Moreover, those limitations will impair scalability. The main conclusion of that study is that preference should be given to the exploitation of coarse-grained parallelism. This conclusion corroborates the results that we have shown in our studies. The work of Eldred *et al.* was focused on exploiting multilevel parallelism on distributed memory systems, using MPI, and, therefore, it is not tailored to the kind of problem that we address in our work.

The NanosCompiler [31] is a source-to-source parallelizing compiler that exploits nested parallelism in OpenMP applications. The compiler uses Parafrase-2 [44] to do a symbolic dependence analysis. Then, the compiler captures: the parallelism expressed through OpenMP directives and extensions; and the parallelism discovered using the symbolic dependence analysis made by Parafrase-2. After, the compiler handles the details of parallel execution. Therefore, the NanosCompiler is tailored to exploit multilevel parallelism in applications that rely on a specific library—OpenMP. Moreover, the NanosCompiler does a static analysis and, therefore, it would be extremely difficult, if possible, to generate parallelizations tailored to handle specificities of irregular applications. As we have shown, parallelism on irregular applications is very input dependent [22].

In [32], Krishnan *et al.* used the Common Component Architecture and Global Arrays (GA) processor groups [45] to express and exploit multilevel parallelism. Therefore, the exploitation of multilevel parallelism is restricted to a programming model, which is imposed by the use of GA library. On the contrary, our work does not rely on a specific: library; framework; programming language; or programming model.

VI. Conclusions and Future Work

In this paper, we have presented a software design that allows to abstract the size of a task, which is estimated at runtime. As we have shown, the proposed software design is agnostic about the concrete implementation of the `size` method. This feature is extremely important since it enables the redefinition of the `size` method without the need of changing elsewhere on the implementation. We also have presented two algorithms—Algorithm 1 and Algorithm 2. Algorithm 1 allows to set the number of threads per task, which is extremely important when exploiting intra-task parallelism. The number of threads used to process a task is set according to the weight that the task has relatively to the weight of the other tasks and the number of cores used. The weight of a task is the ratio between the size of the task and the size of all tasks. Algorithm 2 is a scheduling algorithm that relies on Algorithm 1 and makes a careful usage of available cores on a shared-memory system, accordingly to the number of threads used to process each task.

As shown, with the proposed software design and algorithms it becomes possible to exploit, efficiently, multilevel task parallelism on asymmetric sets of tasks and when using third-party tools. To the best of our knowledge, our work represents the first solution to this kind of problem.

To assess the feasibility and benefit of applying the proposed software design along with both algorithms, we used three irregular applications from phylogenetics—real-world problems—and several input data sets—essentially, biological data sets (i.e., real data)—with different characteristics. The achieved results have shown that with the proposed software design and algorithms it is possible to reach high levels of efficiency and excellent speedups. We recall that those results were obtained using three distinct third-party tools: MAFFT, RAxML, and FastTree. Each of these third-party tools were used as black boxes. Therefore, there was no improvement on the parallelization of each of these third-party tools. Moreover, we have shown that scalability was not impaired even when the parallelization of one of these third-party tools—in the case, FastTree—is limited.

Besides the irregular applications that were used throughout this paper, our approach can be applied to several other applications from distinct domains. A good example is the N-Body codes, which are common in a variety of domains (e.g. astrophysics, biochemistry, biophysics, and 2D/3D graphics), where asymmetric sets of tasks do also appear.

We plan to turn our framework publicly available, as soon as possible. Currently, we are finishing the documentation.

In the near future, we plan to develop support for MPI to enable the use of distributed-memory systems. Thus, the levels of supported parallelism will increase and it will be possible to cope with the growth of data sets. We are also planning to add support for other features, such as checkpointing.

VII. Acknowledgments

This research was partially supported by Fundação para a Ciência e a Tecnologia (grant SFRH/BD/42634/2007).

We thank Rui Carlos Gonçalves for extremely valuable discussions and feedback about implementation issues, and, also, for reviewing an earlier version of the manuscript. We thank João Luís Sobral, Bruno Medeiros, and Rui Silva for fruitful discussions and valuable feedback. We thank Keshav Pingali for his valuable support and sponsorship to let us execute jobs on TACC machines. We are deeply grateful to Rui Oliveira, without whom it would not be possible to present this work. We are very grateful to the anonymous reviewers for the evaluation of our paper and for the constructive critics.

REFERENCES

[1] L. Dagum and R. Menon, "Openmp: an industry standard api for shared-memory programming," *Computational Science & Eng., IEEE*, 1998.

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

[3] J. Reinders, *Intel Threading Building Blocks*, 1st ed. Sebastopol, CA, USA: O'Reilly & Associates, Inc., 2007.

[4] B. Nichols, D. Buttlar, and J. P. Farrell, "Pthreads programming," 1998.

[5] D. B. Kirk and W.-m. W. Hwu, *Programming Massively Parallel Processors: A Hands-on Approach*. Morgan Kaufmann Pub. Inc., 2010.

[6] J. E. Stone, D. Gohara, and G. Shi, "OpenCL: A Parallel Programming Standard for Heterogeneous Computing Systems," *IEEE Des. Test*, vol. 12, no. 3, pp. 66–73, May 2010.

[7] D. Nguyen, A. Lenharth, and K. Pingali, "Deterministic Galois: On-demand, portable and parameterless," in *Proceedings of International Conference on Architectural Support for Programming Languages and Operating Systems*, ser. ASPLOS '14, 2014.

[8] D. T. Neves and J. L. Sobral, "Improving the Separation of Parallel Code in Skeletal Systems," in *IEEE Proceedings of Eighth International Symposium on Parallel and Distributed Computing, ISPDC'09.*, 2009.

[9] R. D. Blumofe, C. F. Joerg, B. C. Kuszmaul, C. E. Leiserson, K. H. Randall, and Y. Zhou, "Cilk: An efficient multithreaded runtime system," *SIGPLAN Not.*, vol. 30, no. 8, pp. 207–216, Aug. 1995.

[10] L. V. Kale and S. Krishnan, "Charm++: A portable concurrent object oriented system based on c++," in *Proceedings of the Eighth Annual Conference on Object-oriented Programming Systems, Languages, and Applications*, ser. OOPSLA '93, 1993, pp. 91–108.

[11] P. Charles, C. Grothoff, V. Saraswat, C. Donawa, A. Kielstra, K. Ebcioglu, C. von Praun, and V. Sarkar, "X10: An Object-oriented Approach to Non-uniform Cluster Computing," *SIGPLAN Not.*, 2005.

[12] B. Chamberlain, D. Callahan, and H. Zima, "Parallel Programmability and the Chapel Language," *Int. J. High Perform. Comput. Appl.*, 2007.

[13] C. Lin and L. Snyder, *Principles of Parallel Programming*, 1st ed. USA: Addison-Wesley Publishing Company, 2008.

[14] W. H. Day, D. S. Johnson, and D. Sankoff, "The Computational Complexity of Inferring Rooted Phylogenies by Parsimony," *Mathematical Biosciences*, vol. 81, no. 1, pp. 33–42, 1986.

[15] W. H. Day and D. Sankoff, "Computational Complexity of Inferring Phylogenies by Compatibility," *Systematic Biology*, 1986.

[16] W. H. Day, "Computational complexity of inferring phylogenies from dissimilarity matrices," *Bulletin of Mathematical Biology*, 1987.

[17] L. Wang and T. Jiang, "On the complexity of multiple sequence alignment," *Journal of Computational Biology*, 1994.

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

[19] M. N. Price, P. S. Dehal, and A. P. Arkin, "FastTree 2 – Approximately Maximum-Likelihood Trees for Large Alignments," *PLoS ONE*, 2010.

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

[21] D. L. Swofford, "PAUP*: phylogenetic analysis using parsimony, version 4.0b10," 2011.

[22] K. Pingali, D. Nguyen, M. Kulkarni, M. Burtscher, M. A. Hassaan, R. Kaleem, T.-H. Lee, A. Lenharth, R. Manevich, M. Méndez-Lojo, D. Prountzos, and X. Sui, "The tao of parallelism in algorithms," *SIGPLAN Not.*, vol. 46, no. 6, pp. 12–25, Jun. 2011.

[23] M. Kulkarni, M. Burtscher, R. Inkulu, K. Pingali, and C. Casçaval, "How much parallelism is there in irregular applications?" in *Proceedings of the 14th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, ser. PPoPP '09, 2009, pp. 3–14.

[24] M. I. Gordon, W. Thies, and S. Amarasinghe, "Exploiting coarse-grained task, data, and pipeline parallelism in stream programs," *SIGARCH Comput. Archit. News*, vol. 34, no. 5, pp. 151–162, Oct. 2006.

[25] M. Mendez-Lojo, D. Nguyen, D. Prountzos, X. Sui, M. A. Hassaan, M. Kulkarni, M. Burtscher, and K. Pingali, "Structure-driven optimizations for amorphous data-parallel programs," in *Proceedings of the ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, ser. PPoPP '10, 2010, pp. 3–14.

[26] D. T. Neves and R. C. Gonçalves, "On the Synthesis and Reconfiguration of Pipelines," in *ARCS 2015*. VDE VERLAG GmbH, 2015.

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

[28] D. T. Neves, T. Warnow, J. L. Sobral, and K. Pingali, "Parallelizing SuperFine," in *Proceedings of the 27th Annual ACM Symposium on Applied Computing*. ACM, 2012, pp. 1361–1367.

[29] D. T. Neves and J. L. Sobral, "Towards a Faster and Accurate Supertree Inference," in *Proceedings of the 20th IEEE Symposium on Computers and Communications, ISCC '15*. IEEE, 2015.

[30] M. Eldred, W. Hart, B. Schimel, and B. van Bloemen Waanders, "Multilevel parallelism for optimization on MP computers: Theory and experiment," in *Proc. 8th AIAA/USAF/NASA/ISSMO Symposium on Multidisciplinary Analysis and Optimization, number AIAA-2000-4818, Long Beach, CA*, vol. 292, 2000, pp. 294–296.

[31] M. Gonzalez, E. Ayguadé, X. Martorell, J. Labarta, N. Navarro, and J. Oliver, "NanosCompiler: supporting flexible multilevel parallelism exploitation in OpenMP," *Concurrency - Practice and Experience*, vol. 12, no. 12, pp. 1205–1218, 2000.

[32] M. Krishnan, Y. Alexeev, T. L. Windus, and J. Nieplocha, "Multilevel Parallelism in Computational Chemistry using Common Component Architecture and Global Arrays," in *Supercomputing, 2005. Proceedings of the ACM/IEEE SC 2005 Conference*. IEEE, 2005, pp. 23–23.

[33] Y. Hu, H. Jiang, D. Feng, L. Tian, H. Luo, and C. Ren, "Exploring and Exploiting the Multilevel Parallelism Inside SSDs for Improved Performance and Endurance," *Computers, IEEE Transactions on*, vol. 62, no. 6, pp. 1141–1155, 2013.

[34] 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, Jun. 2009.

[35] E. Anderson, Z. Bai, J. Dongarra, A. Greenbaum, A. McKenney, J. Du Croz, S. Hammerling, J. Demmel, C. Bischof, and D. Sorensen, "LAPACK: A Portable Linear Algebra Library for High-performance Computers," in *Proceedings of the 1990 ACM/IEEE Conference on Supercomputing*, 1990.

[36] G. Karypis and V. Kumar, "A Fast and High Quality mMultilevel Scheme for Partitioning Irregular Graphs," *SIAM Journal on scientific Computing*, vol. 20, no. 1, pp. 359–392, 1998.

[37] E. Gamma, R. Helm, R. Johnson, and J. Vlissides, *Design patterns: elements of reusable object-oriented software*. Pearson Education, 1994.

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

[39] 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.

[40] M. Cardillo, O. R. P. Bininda-Emonds, E. Boakes, and A. Purvis, "A species-level phylogenetic supertree of marsupials," *J. Zool.*, 2004.

[41] 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.

[42] M. Kennedy and R. D. M. Page, "Seabird supertrees: combining partial estimates of procellariiform phylogeny," *The Auk*, 2002.

[43] 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.

[44] C. D. Polychronopoulos, M. Girkar, M. R. Haghighat, C. L. Lee, B. Leung, and D. Schouten, "Parafrase-2: An Environment for Parallelizing, Partitioning, Synchronizing, and Scheduling Programs on Multiprocessors," *International Journal of High Speed Computing*, 1989.

[45] J. Nieplocha, M. Krishnan, B. Palmer, V. Tipparaju, and Y. Zhang, "Exploiting Processor Groups to Extend Scalability of the GA Shared Memory Programming Model," in *Proceedings of the 2nd conference on Computing Frontiers*. ACM, 2005, pp. 262–272.