Hierarchical management of extreme-scale task-based applications

Francesc Lordan[1][0000−0002−9845−8890], Gabriel Puigdemunt[1][0000−0002−5576−9912], Pere Vergés[1,2][0000−0002−4109−1071], Javier Conejero[1][0000−0001−6401−6229], Jorge Ejarque[1][0000−0003−4725−5097], and Rosa M. Badia[1][0000−0003−2941−5499]

[1] Barcelona Supercomputing Center, Barcelona, Spain
{francesc.lordan, gabriel.puigdemunt, pere.verges, javier.conejero, jorge.ejarque, rosa.m.badia}@bsc.es
[2] University of California Irvine
pvergesb@uci.edu

Abstract. The scale and heterogeneity of exascale systems increment the complexity of programming applications exploiting them. Task-based approaches with support for nested tasks are a good-fitting model for them because of the flexibility lying in the task concept. Resembling the hierarchical organization of the hardware, this paper proposes establishing a hierarchy in the application workflow for mapping coarse-grain tasks to the broader hardware components and finer-grain tasks to the lowest levels of the resource hierarchy to benefit from lower-latency and higher-bandwidth communications and exploiting locality.
Building on a proposed mechanism to encapsulate within the task the management of its finer-grain parallelism, the paper presents a hierarchical peer-to-peer engine orchestrating the execution of workflow hierarchies with fully-decentralized management. The tests conducted on the MareNostrum 4 supercomputer using a prototype implementation prove the validity of the proposal supporting the execution of up to 707,653 tasks using 2,400 cores and achieving speedups of up to 106 times faster than executions of a single workflow and centralized management.

Keywords: distributed systems, exascale, task-based, programming model, workflow, hierarchy, runtime system, peer-to-peer, decentralized management

1 Introduction

Systems targeting exascale computing are becoming more and more powerful by interconnecting a growing number of nodes equipping processors and accelerators with an increasing number of physical cores and novel memory hierarchies. The extreme scale and the heterogeneity of these systems increment the overall complexity of programming applications while exploiting them efficiently. On the one hand, developers have to identify enough parallelism inherent in the application to employ all the compute devices; on the other hand, they have to face the heterogeneity of the system and deal with the specifics of each device (i.e. architectures with a different number of physical cores, memory sizes and hierarchies, network latency and bandwidth, and different programming models to interact with the device). This results in system-tailored applications that can not be ported to other systems without a significant performance loss.

Programming models overcome this development difficulty by providing an infrastructure- and parallelism-agnostic mechanism to describe the logic of an application. Then, at execution time, a runtime engine automatically handles the inherent parallelism to exploit the host infrastructure. Task-based programming models are a popular approach because of their high development productivity and their flexibility to adapt to the infrastructure. They build on the concept of task: an asynchronous operation processing a collection of input values to generate a set of output values. Tasks can take values generated by other tasks as input; hence, establishing data-dependency relationships among them. These dependencies define the workflow of the application and determine its inherent task-level parallelism; runtime engines orchestrate the parallel execution of all the tasks of an application guaranteeing the fulfilment of these dependencies.

Tasks encapsulate logic operations; however, the actual implementation carrying them out can change depending on the available hardware or the current workload of the system. Thus, the runtime engine can select an implementation leveraging a specific accelerator, running a multi-threaded implementation on multi-core processors, or a distributed version using several nodes. Runtime engines usually centralize the parallelism and resource management in one single node of the infrastructure (the orchestrator); on extreme-scale computers, the large number of tasks and nodes converts this management into a bottleneck.

Leveraging this task implementation versatility, applications can organize their parallelism hierarchically embedding finer-grain tasks within intermediate tasks to distribute the parallelism management overhead. Hence, the orchestrator node handles only the coarsest-grain parallelism and passes on the burden of managing the finer-grain parallelism along with the execution of the task. The node running a task decides whether to execute the tasks composing the inner workflow locally or offload them onto other nodes distributing the management workload in a recursive manner. For that to succeed, each node of the infrastructure must be aware of the computing devices equipped on the node and the amount of resources available on the other nodes of the infrastructure. This deprecates the orchestrator node approach in favour of a peer-to-peer model.

This paper contributes with an analysis of what are the requirements to bundle the fine-grain parallelism management within a task and the description of the necessary mechanisms to implement in a runtime system to support it. Besides, the article presents the results of evaluation tests using a prototype implementation conducted on the MareNostrum 4 supercomputer running two different applications (GridSearch and Random Forest) achieving higher degrees of parallelism and reducing the management overhead drastically.

The article continues by casting a glance over related work that can be found in the literature. Section 3 introduces the concepts of the proposed solution and Section 4 describes how runtime systems must handle data, resources and tasks to adopt it. To validate the proposal, Section 5 contains the results of the evaluation of a prototype implementation in two use cases. The last section concludes the article and identifies potential lines for future work.

2 Related Work

Previous work has addressed the support for hierarchical or nested parallel regions, especially in shared memory systems such as multi-core architectures. Most of the widely-adopted shared memory programming models – e.g., OmpSs [5], Cilk [19] or OneAPI TBB [10] – support nested parallel regions or tasks, and the OpenMP standard also supports nested tasks since version 4.5 [16]. The management of nested parallelism focuses on handling dependencies between different nested parallel regions to allow their correct concurrent processing. Having shared memory simplifies this management since data regions can be directly identified by their memory addresses, and the different threads processing these regions have shared access to the control data. The solution presented in this article targets distributed systems where application and control data are spread across the infrastructure; thus, memory addresses no longer uniquely identify data regions and control data is not shared among all the computing nodes or devices making the parallelism management more complex.

In distributed systems, nested parallelism is typically achieved by combining different programming models, one supporting the distributed system part and another dealing with the execution within each shared memory system. This is the case of the hybrid MPI + OpenMP model [18], StarSs [17] or the COMPSs + OmpSs combination [6]. Since the runtime systems supporting these models do not share information, developers must master several models and manage the coordination of different levels of parallelism. The proposal of this article uses a single programming model to support parallelism at all granularity levels.

Current state-of-the-art workflow managers have done some efforts to enable nested parallelism. Most of them allow the explicit sub-workflow definition (e.g. Snakemake [15], NextFlow [4] and Galaxy [1]), or even allow the definition of external workflows as modules (e.g. Snakemake), to enable the composition of larger workflows. However, they rely on the dependency management of the underlying queuing system (e.g., Slurm [21]) and submit each task as an individual job with the required job dependencies. These systems are centralised and this methodology leads to floods of jobs. Alternatively, Dask [3] allows launching nested tasks within the same job by allowing the creation of clients that connect to the main Dask scheduler to spawn a child task. Unfortunately, this approach has the same essence as previous workflow managers since it also relies on a centralized scheduling system; besides, it is considered an experimental feature. The methodology described in this work differentiates from these solutions by following a decentralized approach to deal with this management. This feature has also been explored by dataflow managers (e.g., Swift/T [20] and TTG [8]); however, their management approach cannot be applied onto workflow managers.

3 Workflow Management Encapsulation

Task-based approaches with support for nested tasks are a good-fitting model for extreme-scale systems because of the flexibility of the task concept. Tasks are asynchronous operations with a defined set of input and output data – in the context of this article, the term data refers to individual files and objects. The definition of a task establishes an operation to carry out but specifies nothing about its implementation. Thus, a task can run sequentially or create new tasks to open additional parallelism (nested task detection). This establishes relationships among tasks. All the tasks created by the same task are child or nested tasks of the creating task; inversely, the creating task is the parent of all the tasks created by it. Tasks sharing the same parent are siblings.

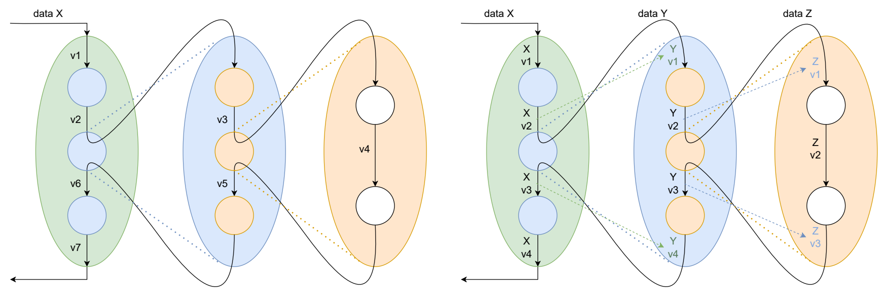
It is during a task execution that nested tasks are discovered; thus, tasks never start executing earlier than their parent. When it comes to finishing a task, a parent task must wait until all of its nested tasks have been completed before it can finish its execution. This is because the output of the parent task relies on the outputs of its child tasks.

Task-based programming models build on data access atomicity to convert an application into a workflow by establishing dependency relations among tasks where the outputs of a task (task predecessor or value producer) are the inputs of another (task successor or value consumer). By detecting nested tasks, each task has the potential to become a new workflow, and thus, applications evolve from being a single workflow into a hierarchy of workflows. The workflow of a task can define data dependencies among its nested tasks based on the access to its input data or newly created intermediate data. However, beyond the scope of the task, only those values belonging to the output on the task definition are significant; hence, all intermediate data is negligible and can be removed.

As with the implementation, the task definition does not specify which resources should host its execution. Any node with access to such values can host the execution of the task; thus, by transferring the necessary input data, the workload of a task-based application can be distributed across large systems and run the tasks in parallel.

Ensuring that data has the expected value when passed in as a task input is crucial to guarantee that applications produce their expected results while making the most of the underlying infrastructure. To identify more parallelism between tasks, it is possible to maintain a duplicate of every value the data holds throughout the execution. These replicas allow any task to read the expected value even if another task has already updated it; thus, false dependencies are no longer considered. The counterpoint of this method is the additional storage. To orchestrate the parallel execution of tasks, it is crucial to keep track of the values held by a data, the location of their duplicate copies, and the pending-execution tasks reading each value. This tracking enables not only identifying dependency-free tasks but also detecting obsolete values – i.e, old values with no tasks reading them – that can be removed to free storage capacity.

Fig. 1a illustrates the different values held by a data (*data X*) that enters as an input value of a task and is updated by three nested workflows. *Data X*



(a) Global data versioning (b) Encapsulated data versioning

Fig. 1: Example of data versioning where all tasks belonging to a three-level hierarchy of nested workflows update one data.

enters the coarsest-grain task (green oval) as an input value (*v1*) and is updated by the three-task workflow nested in the task (tasks within this workflow are blue). The first blue task reads and updates the input value by generating a new version (*v2*). This new version is passed in as the input of the second blue task, and, during the execution, another three-task nested workflow is detected (tasks within this workflow are depicted in orange). The first task of this finer-grain workflow updates the data (*v3*), and the second task gets it as input. Again, at run time, this second task becomes a workflow with two inner tasks (white) modifying the data and, thus, generating two subsequent versions (*v4* and *v5*). Upon completing the second white task, the whole workflow in the second orange task is completed; *v5* becomes the output value of the second orange task, and *v4* becomes an irrelevant intermediate value. The third orange task takes the *v5* value and updates the data generating *v6*. At this point, the whole workflow within the second blue task is completed; *v6* becomes the output value of the data for the blue task, and *v3* and *v5* become deprecated because they are intermediate values within the second blue task. Finally, the third blue task can be executed taking *v6* as input value to generate *v7*, which becomes the output value of the green task deprecating *v2* and *v6*.

Determining an incremental version number at task discovery time, as done in the previous example, is unfeasible. On the one hand, the execution of the different tasks is distributed across the whole system; maintaining this version record to ensure the proper handling of the dependencies requires a centralized entity or implementing consensus. Both solutions entail a significant communication overhead. On the other hand, versions generated by a nested workflow are detected at task execution time and not while the coarser-grain workflow is detected. Thus, data versions generated by tasks in a parent workflow would be detected earlier than the versions from its nested tasks. Hence, the data value discovery would not match the incremental order of the versions.

To workaround these difficulties and overcome both problems, this work proposes registering the intermediate values as versions from a different data and linking the corresponding versions to share the same copy of the value as depicted in Fig. 1b. When a new task generates a workflow, all its input data values are registered as the first version of a new data; intermediate values are considered versions of that data. Thus, in the same case of the previous example, when the green task starts executing, it only detects four versions of *data X* (the input version: *v1*, *v2* and *v3* as intermediate values, and the final version: *v4*). When the second blue task starts executing, the system registers the first version (*v1*) of a new data *data Y*, and links *v1* of *data Y* with *v2* of *data X*. The versions generated by the orange tasks are registered only as versions of *data Y*. At the end of the execution of all orange tasks, *v4* of *data Y* is linked to *v3* of *data X*. Since *data Y* will no longer be available, all its versions are declared deprecated. Thus, all the copies of the intermediate versions of *data Y* (*v2* and *v3*) can be removed. Still, the input and output versions (*v1* and *v4*) are kept because they are accessible through the versions of *data X*.

Following this proposal, the nested workflow management is encapsulated within the task creating it. Once a node starts running a task, it can spawn the nested tasks and orchestrate their execution independently from the execution of other workflows. As with the computational workload, the management overhead gets distributed to reduce the management bottleneck of centralized approaches.

4 Runtime System Architecture

The hardware of Exascale computers is already organized hierarchically. Systems are composed of thousands of nodes physically in racks interconnected by switches; internally, each node can have several processors with multiple cores and accelerators attached. This hierarchy can be put to use and define the different domain levels described to distribute the resource management. Thus, coarser-grained tasks can be mapped to the broader domains of the infrastructure, and finer-grain tasks, where the bulk of parallelism is, achieve higher performance by exploiting data locality and lower-latency and higher-bandwidth communications offered within the lowest levels of the resource hierarchy.

To fully achieve their potential performance, task computations require exclusive access to the resources running their logic to reduce the issues of concurrent execution on shared resources such as increasing the number of cache misses or memory swapping. Runtime systems monitor the resource occupation to orchestrate the task executions and grant this exclusivity. An orchestrator node handling a large number of task executions on many workers becomes a management bottleneck in extreme-scale infrastructures. Given the management independence provided by workflow hierarchies, peer-to-peer architectures arise as a promising architecture to efficiently support the detection of nested tasks and distribute the management overhead. In this approach, each node hosts an autonomous process (Agent) that establishes a collaborative data space with other nodes and handles the execution of tasks.

Each Agent controls the computing devices equipped on the node to allocate task executions and monitors the resources from neighbouring nodes with the

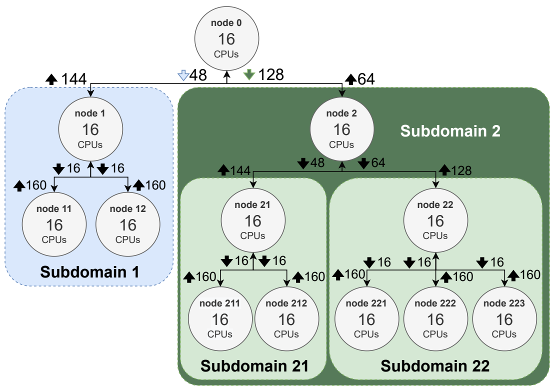


Fig. 2: 12-node cluster with a total of 176 cores divided into a hierarchy of domains. From the point of view of node0, the infrastructure is composed of two domains accessible via node1 (subdomain1) and node2 (subdomain2).

purpose of offloading tasks onto them. Despite not being a limit on the number of remote nodes, the more nodes being monitored, the larger the management overhead and the complexity of scheduling task executions. To distribute the management, the resources can be grouped into disjoint domains, each managed by one of the nodes within it. Instead of monitoring the state of many nodes, The orchestrator node only distributes the workload among a few resource-rich domains interacting with the manager node within each. In the example depicted in Fig. 2, a cluster is divided into two domains. The orchestrator node (node0) considers only 3 options to host the execution: its 16 local CPU cores, 48 CPU cores available in Domain1 through node1, or 128 CPU cores in Domain2 through node2. The resources within a domain can still be too many to be handled by a single node. To that end, domains can be subsequently divided into several subdomains establishing a resource hierarchy as depicted in Domain2 of Fig. 2 where node2 considers 3 options: hosting it in its local 16 CPU cores, delegating it to one of its subdomains (Subdomain21 with 48 cores or Subdomain22 with 64 cores) pushing it down the hierarchy, or offloading out from the domain ascending through the hierarchy (64 cores available through node0).

Agents comprise five main components as illustrated in Fig. 3. The *Agent API* offers users and other Agents an interaction interface to submit task execution requests and notify task completions. The *Data Manager* establishes a distributed key-value store used by Agents to register data values and share their values. The *Task Scheduler* monitors the data dependencies of the workflows generated by the tasks running on the node and decides the best moment to start a task execution or offload it onto a domain. The *Local Execution* and *Offloading Engine* respectively handle the execution of tasks on the local devices and their offloading onto remote Agents. An internal API allows tasks implemented with *task-based programming models* to notify the detection of nested workflows and request the execution of their child tasks.

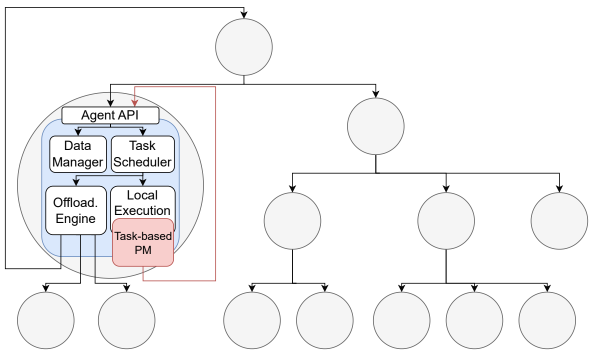


Fig. 3: Components of the Agent deployed in node1 from Fig. 2.

Tasks arrive to the Agent through an API indicating the operation to perform and its parameters (input data values and the expected results). Upon reception, the Agent registers each parameter as a new piece of data and binds the first version of all the input values with the corresponding version, as described in Section 3. Then, the task becomes part of a pool of pending workload; the Agent schedules the execution of these pending tasks considering the availability of the local or remote resources, aiming for an optimal distribution while providing resource exclusivity to the tasks. Arrived the time if the Agent decides to offload the task onto a remote node, it reserves the resources required by the task in the corresponding subdomain, submits via the API of the main Agent of the subdomain, and waits asynchronously for its completion to release the resources.

Otherwise, if the Agent decides to host the execution locally, it allocates the corresponding local resources, fetches all the missing input values and launches its execution. If implemented following a task-based programming model, the task becomes a workflow and spawns nested tasks with dependencies among them, creating new pieces of data and new versions of the already existing parameters. As explained in the previous section, this data management, as well as the parallelism among nested tasks, can be handled within the node with no need to interact with other peers. Hence, the programming model notifies the newly detected nested tasks and their dependencies to the local Agent. It manages their execution with the parent task running them locally or offloading them onto other nodes.

Workflow executions reach synchronization points where they wait for some of their nested tasks to end producing data values. Every task that becomes a workflow reaches at least one synchronization point at its end to wait for the completion of all its nested tasks. During these waits, the resources allocated for the parent task remain idle. For better exploitation of the infrastructure, the task can release these resources so they can host another task execution; for instance, one of its nested tasks. When the synchronization condition is met and the nested task being waited for ends its execution, the execution of the parent task can continue. At this point, the runtime system has to ensure that there are enough idle resources to host the parent task execution exclusively. If there are, the task resumes its execution; otherwise, the runtime will hold the execution until other tasks release their resources because they complete their execution or they reach a synchronization point.

Regardless of whether a task has become a workflow or not, upon finishing its execution (including its nested tasks), the Agent collects all the output values, binds them to the corresponding version of its parent task data (passed in as parameters) and removes all the references to the pieces of data created for the task. At this point, the runtime system considers the task completed, releasing its resources and dependencies. If the task was detected by a parent task running in the same node, the Agent releases the local resources allocated for its execution and any data dependency with its successors. Otherwise, if the task was offloaded from another node, the Agent notifies the completion of the task to the Agent from where it was submitted to release the resources of the corresponding domain. If the notified node is the Agent where the task was detected, it also releases the dependencies; otherwise, the notification is forwarded to the Agent that sent it, repeating the process until it reaches the source Agent to release the data dependencies and continue with the execution of the parent workflow.

5 Evaluation

To validate the proposed idea, several tests have been conducted using a prototype implementation building on Colony [14]: a framework to handle task executions on distributed infrastructures organizing the resources as a hierarchical peer-to-peer network. The task-based programming model selected for defining the nested workflows is COMPSs [13], for which Colony provides native support.

All the experiments have been run on the MareNostrum 4 supercomputer: a 3,456-node (48 servers of 72 nodes) supercomputer where each node has two 24-core Intel Xeon Platinum 8160 and 96 GB of main memory. A Full-fat tree 100Gb Intel Omni-Path network interconnects the nodes which also have access to a 14PB shared disk. Each node hosts the execution of an Agent managing its 48 cores. All the Agents within the same server join together as a domain and one becomes the interconnection node for the domain; in turn, one of these nodes interconnects all the domains and receives the main task of the application.

The scheduler within each Agent is the default Colony scheduler. Upon the arrival of a dependency-free task, it attempts to assign it to an idle resource considering the locality of its input values. If there are no available resources, the scheduler adds the task to a set of pending tasks. When a task completes, the scheduler releases the used resources and the successors and tries to employ any idle resources with one of the just dependency-freed tasks or one from the pending set computing a locality score for all the combinations and iteratively selecting the one with a higher value until no task can be assigned. To avoid loops where a task is being submitted between two Agents back and forth, the scheduler dismisses offloading the task onto the Agent detecting it or any of its parents; offloading is always down the hierarchy.

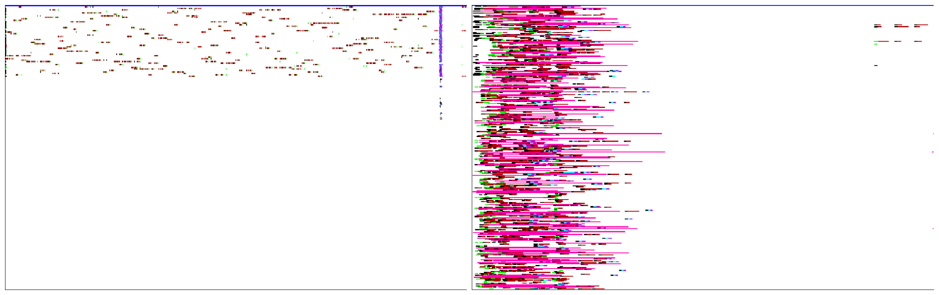
5.1 GridSearch

The first test evaluates the performance of GridSearch [11] with cross-validation: an algorithm that exhaustively looks for all the different combinations of hyper-parameters for a particular estimator. With cross-validation, it trains and evaluates several estimators for each combination (splits), and the final score obtained for a combination is the average of the scoring of the corresponding splits. GridSearch is one of the algorithms offered within dislib [2], a Python library built on top of COMPSs providing distributed mathematical and machine learning algorithms. The conducted test finds the optimal solution among 25 combinations of values for the *Gamma* (5 values from 0.1 to 0.5) and C (5 values from 0.1 to 0.5) hyper-parameters to train a Cascade-SVM classification model (CSVM) [7].

The implementation of GridSearch provided within dislib – Flat – delegates the detection of the tasks to the implementation of the estimator and invokes them sequentially expecting them to create all the finer-grain tasks at a time. CSVM is an iterative algorithm that checks the convergence of the model at the end of every iteration; hence, it stops the generation of tasks at the end of each iteration. This affects the parallelism of GridSearch; it does not detect tasks from a CSVM until the previous one converges. The Nested version of the GridSearch algorithm encapsulates the fitting and evaluation of each estimator within a coarse-grain task that generates the corresponding finer-grain tasks achieving higher degrees of parallelism. Albeit both versions reach the same task granularity, the Nested version overcomes the task generation blockage enabling parallel convergence checks by encapsulating them within nested workflows.

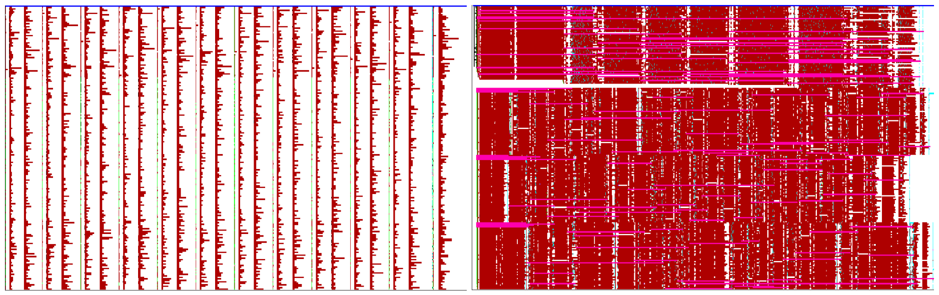
The first test studies the behaviour when training a small dataset (the IRIS dataset) using 4 Marenostrum nodes. Fig. 4 depicts an execution trace with the 192 cores when executing the Flat (Fig. 4a) and Nested (Fig. 4b) implementations. Given the small size of the dataset, the corresponding CSVM implementation does not detect many tasks to run in parallel. In the Flat version case, where CSVMs run sequentially, the infrastructure is under-utilized and takes 116.27 seconds to run. Enabling nested task detection allows running several CSVMs simultaneously; this increases the number of finer-grain tasks detected at a time, and the infrastructure hosts more executions in parallel. The overall execution time is reduced to 9.33 seconds (12$\times$ speedup).

When CSVM processes larger datasets (e.g., the Atrial Fibrillation (AT) composed of 7,500 samples with 100 features characterizing an ECG), it can detect enough parallelism to fully use the 4 nodes simultaneously as shown in Fig. 5a. However, convergence checks reduce the parallelism in every iteration and a large part of the infrastructure is under-used. By overlapping several CSVMs, the Nested version employs these idle resources to compute tasks from other CSVMs as depicted in Fig. 5b. For this experiment, the Nested version reduces the time to find the optimal solution among 25 combinations from 27,887 seconds to 5,687 (4.9x speedup). Aiming at verifying the scalability of the solution, we run a GridSearch to find the optimal solution among 50 combinations: 250 CSVMs and 707,653 tasks. When processing the AT dataset, a CSVM generates parallelism to employ up to 4 nodes. With the FLAT version, the estimated



(a) Flat: 25 combinations - 118 seconds (b) Nested: 25 combinations - 10 seconds

Fig. 4: Execution trace of an IRIS model training with 4 nodes of 48 cores



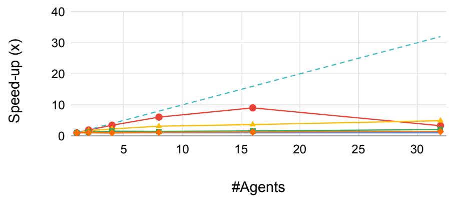
(a) Flat: 2 combinations - 2,500 seconds (b) Nested: 25 combinations - 5,700 seconds

Fig. 5: Execution trace of an AT model training with 4 nodes of 48 cores

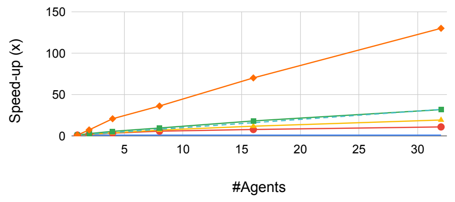
shortest execution time is 55,774 seconds. The Nested version expands this parallelism enabling the usage of more nodes. With 16 nodes, it lasts 4,315 seconds (13x). With 50, the execution already shows some workload imbalance due to the variability between CSVMs; it takes 1,824 seconds (30x).

5.2 Random Forest

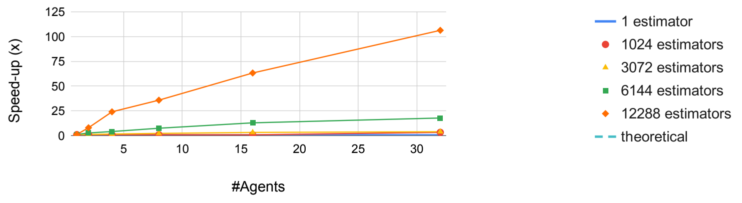
The second experiment consists in training a classification model using the RandomForest algorithm [9], which constructs a set of individual decision-trees – estimators –, each classifying a given input into classes based on decisions taken in random order. The model aggregates the classification of all the estimators; thus, its accuracy depends on the number of estimators composing it. The training of an estimator has two tasks: the first one selects 30,000 random samples from the training set, and the second one builds the decision tree. The training of an estimator is independent of other estimators. The test uses two versions of the algorithm: one – Flat – where the main task directly generates all the tasks to train the estimators and the other – Nested – where the main task generates intermediate tasks grouping the training of several estimators. In the conducted



(a) Speedup for Flat version



(b) Speedup for Nested version



(c) Speedup Nested vs Flat

Fig. 6: Strong scaling results for a 1-, 1024-, 3072-, 6144-, 12288-estimator random forest model training

tests, each batch trains at least 48 estimators, and if the number of estimators allows it, the number of intermediate tasks matches the number of Agents.

Fig. 6 depicts the speedup obtained when running a strong scaling test with each of the versions when training 1, 1024, 3072, 6144 and 12288 estimators. The results for the flat version (Fig. 6a) show the scalability limitation due to the workload imbalance when a parallelism hierarchy is not established (seen in the 1024-estimator case with not enough parallelism to exploit the 1536 cores in 32 agents). In addition, this alternative suffers from the delay produced by generating the tasks sequentially and from a scheduling overhead that grows exponentially with the number of pending tasks.

Nested tasks diminish the impact of the latter two. Several coarse-grained tasks can run at a time and generate finer-grain tasks in parallel; the faster tasks are detected, the faster the runtime system can submit their execution and better exploit the resources. Besides, the runtime system can distribute the scheduling of these tasks; hence, its overhead is drastically reduced as the infrastructure grows. As shown in Fig. 6b, mitigating these two issues allows a 130 times faster training of a 12,288-estimator model when using 32 times more resources.

Fig. 6c compares the execution times obtained with both algorithms when training the same model using the same amount of resources. The larger the model and the infrastructure are, the higher the benefit of establishing a parallelism hierarchy is. In the largest test case, training a 12,288-estimator model using 32 nodes, the Nested algorithm achieves an execution time 106 times faster than the Flat. The experiments using a single node, where tasks are detected sequentially and the scheduler handles the same amount of tasks, do not reveal any significant overhead due to the handling of the additional parent task.

6 Conclusion

This manuscript describes a mechanism to organize the parallelism of task-based applications in a hierarchical manner and proposes a mechanism to encapsulate the management of the nested workflow along with the task to enable the distribution of the management overhead along the infrastructure. Matching the application parallelism, the article also proposes a hierarchical approach for organizing the resources of the infrastructure; thus, the scheduling problem reduces its complexity by handling fewer tasks and resources. The article also describes the architecture of a runtime system supporting it.

The paper validates the proposal with two tests on a prototype implementation running on the MareNostrum 4 supercomputer. The results reveal that, by establishing a task hierarchy, applications can achieve a higher degree of parallelism without undergoing an in-depth refactoring of the code. Encapsulating the finer-grain parallelism management within tasks to distribute the corresponding overhead is beneficial for the application performance; results achieve a speedup of up to 106 times faster than executions with centralized workflow management.

The tests also reveal some shortcomings of the prototype. The biggest concern is the limitation of the task scheduler to request task executions to higher layers of the resource hierarchy. Developing peer-to-peer scheduling strategies based on task-stealing, reactive offloading or game theory are future lines of research to improve. Also, the described work considers that the output of the task is available only at the end of its execution. However, a nested task can compute an output value of the parent task before its completion. Currently, other tasks depending on the value must wait for the parent task to end even if the value is already available. Enabling fine-grain dependency management that releases the dependency upon the completion of the nested task is also future work.

Acknowledgements and Data Availability

This work has been supported by the Spanish Government (PID2019-107255GB), by MCIN/AEI /10.13039/501100011033 (CEX2021-001148-S), by the Departament de Recerca i Universitats de la Generalitat de Catalunya to the Research Group MPiEDist (2021 SGR 00412), and by the European Commission through the Horizon Europe Research and Innovation program under Grant Agreements 101070177 (ICOS project) and 101016577 (AI-Sprint project).

The data and code that support this study are openly available in figshare [12].

References

1. Afgan, E., et al.: The Galaxy platform for accessible, reproducible and collaborative biomedical analyses: 2018 update. Nucleic Acids Res. **46**(1), 537–544 (2018)

2. Álvarez Cid-Fuentes, J., et al.: dislib: Large Scale High Performance Machine Learning in Python. In: Proceedings of the 15th International Conference on eScience. pp. 96–105 (2019)
3. Dask Development Team: Dask: Library for dynamic task scheduling (2016), https://dask.org
4. Di Tommaso, P., et al.: Nextflow enables reproducible computational workflows. Nature biotechnology **35**(4), 316–319 (2017)
5. Duran, A., et al.: Ompss: a proposal for programming heterogeneous multi-core architectures. Parallel processing letters **21**(02), 173–193 (2011)
6. Ejarque, J., et al.: A hierarchic task-based programming model for distributed heterogeneous computing. The International Journal of High Performance Computing Applications **33**(5), 987–997 (2019)
7. Graf, H., et al.: Parallel support vector machines: The cascade SVM. Advances in neural information processing systems **17** (2004)
8. Herault, T., et al.: Composition of algorithmic building blocks in template task graphs. In: 2022 IEEE/ACM Parallel Applications Workshop: Alternatives To MPI+ X (PAW-ATM). pp. 26–38 (2022)
9. Ho, T.K.: Random decision forests. In: Proceedings of 3rd international conference on document analysis and recognition. vol. 1, pp. 278–282 (1995)
10. Intel Corporation: OneAPI TBB Nested parallelism (2022), https://oneapi-src.github.io/oneTBB/main/tbb_userguide/Cancellation_and_Nested_Parallelism.html
11. Lerman, P.: Fitting segmented regression models by grid search. Journal of the Royal Statistical Society Series C: Applied Statistics **29**(1), 77–84 (1980)
12. Lordan, F., et al.: Artifact and instructions to generate experimental results for the Euro-Par 2023 proceedings paper: Hierarchical management of extreme-scale task-based applications. https://doi.org/10.6084/m9.figshare.23552229
13. Lordan, F., et al.: ServiceSs: An Interoperable Programming Framework for the Cloud. Journal of Grid Computing **12**(1), 67–91 (2014)
14. Lordan, F., et al.: "Colony: Parallel Functions as a Service on the Cloud-Edge Continuum". In: "Euro-Par 2021: Parallel Processing". pp. 269–284 (2021)
15. Mölder, F., et al.: Sustainable data analysis with Snakemake. F1000Research **10**(33) (2021)
16. Perez, J.M., et al.: Improving the integration of task nesting and dependencies in OpenMP. In: 2017 IEEE International Parallel and Distributed Processing Symposium (IPDPS). pp. 809–818 (2017)
17. Planas, J., et al.: Hierarchical task-based programming with StarSs. The International Journal of High Performance Computing Applications **23**(3), 284–299 (2009)
18. Rabenseifner, R., et al.: Hybrid MPI/OpenMP parallel programming on clusters of multi-core SMP nodes. In: 2009 17th Euromicro international conference on parallel, distributed and network-based processing. pp. 427–436 (2009)
19. Vandierendonck, H., et al.: Parallel Programming of General-Purpose Programs Using Task-Based Programming Models. In: 3rd USENIX Workshop on Hot Topics in Parallelism (HotPar 11) (2011)
20. Wozniak, J.M., et al.: Swift/t: Large-scale application composition via distributed-memory dataflow processing. In: 2013 13th IEEE/ACM International Symposium on Cluster, Cloud, and Grid Computing. pp. 95–102 (2013)
21. Yoo, A.B., et al.: SLURM: Simple Linux Utility for Resource Management. In: Job Scheduling Strategies for Parallel Processing. pp. 44–60 (2003)