Convergence of HPC and Big Data in extreme-scale data analysis through the DCEx programming model

Javier Garcia-Blas, Javier Fernandez Muñoz, Jesus Carretero
Computer Science and Engineering department
University Carlos III of Madrid Leganes, Spain

Fabrizio Marozzo, Domenico Talia, Paolo Trunfio
DIMES department
University of Calabria Rende, Italy

Alberto Fernandez-Pena, Daniel Martín de Blas
Instituto de Investigación Sanitaria Gregorio Marañón
Departamento de Bioingeniería e Ingeniería Aeroespacial
Universidad Carlos III de Madrid Madrid, Spain

***Abstract*—High-level programming models can help application developers to access and use resources without the need to manage low-level architectural entities, as a parallel programming model defines a set of programming abstractions that simplify the way by which a programmer structures and expresses her/his algorithm. Early proposals of Exascale programming tools are based on the adaptation of traditional parallel programming languages and hybrid solutions. This incremental approach is too conservative, often resulting in very complex code. This paper describes the design features, the programming constructs, and the runtime mechanisms of the Data Centric programming model for Exascale systems (DCEx). DCEx is based on structuring applications into data-parallel blocks. Blocks are units of shared- and distributed-memory parallel computation, communication, and migration in the memory/storage hierarchy. Blocks and their message queues are mapped onto processes and placed in memory/storage by the DCEx runtime. Those data-parallel blocks are orchestrated by using distributed parallel patterns that simplify the development cost. DCEx aims to reach the convergence of traditional HPC programming models, mainly based on MPI, with the emerging technologies based on the data intensive paradigms. To demonstrate the potential of DCEx, we carried out an experimental evaluation developing a real-world diffusion-weighted magnetic resonance imaging data processing application in a neuroimaging research context.**

***Index Terms*—Big Data, HPC convergence, Programming model, PGAS, parallel patterns**

I. Introduction

High-performance computing (HPC) refers to the usage of aggregated computing power in order to deliver as much speed as possible to run complex parallel programs efficiently. This term is tightly related to the concept of *supercomputing*, which pushes HPC to the highest operational rate of the available technology. Nowadays, top modern supercomputers reach performance ranging hundreds of petaflops, and a machine capable of delivering one exaflop is expected to appear in 2022.

Parallel programming models can assist application developers to exploit computational resources without the need to manage low-level architectural entities, as they define a set of programming abstractions that simplify the way program structures are expressed in algorithms. Previous proposals are based on the adaptation of traditional parallel programming languages and hybrid solutions. This incremental approach is too conservative, often resulting in very complicated codes, which also generates the risk of limiting the scalability of programs on millions of cores. Load balancing is still an open question. Many current parallel programming solutions contain an implicit trade-off between simplicity and performance. Those abstracting the programmer from lower-level parallel details often sacrifice performance and scalability in the name of simplicity, and vice versa. Several current implementations of the PGAS (partitioned global address space) model are no exception to this and unfortunately, limited progress has been made in the last few years toward the implementation of simple models that can be used to program at any scale and performance. The need for scalable programming models continues and the emergence of new hardware architectures makes this need more urgent.

In this paper we present a novel data-aware paradigm called Data Centric programming model for large-scale systems (DCEx) and runtime mechanisms for data intensive applications. DCEx[1] model is based on data-parallel tasks providing a simplified API. DCEx addresses various features to enhance performance of data intensive computations in data-intensive applications, by reducing the cost of accessing, moving, and

This work was supported by the EU project "ASPIDE: Exascale Programming Models for Extreme Data Processing" under grant 801091 and ADMIRE project under grant 956748-ADMIRE-H2020-JTI-EuroHPC-2019-1. This research was partially supported by the Madrid regional Government (Spain) under the grant "Convergencia Big Data-HPC: de los sensores a las Aplicaciones. (CABAHLA-CM)". Finally, this work was partially supported by the Spanish Ministry of Science and Innovation Project "New Data Intensive Computing Methods for High-End and Edge Computing Platforms (DECIDE)" Ref. PID2019-107858GB-I00. Data were provided [in part] by the Human Connectome Project, WU-Minn Consortium (Principal Investigators: David Van Essen and Kamil Ugurbil; 1U54MH091657) funded by the 16 NIH Institutes and Centers that support the NIH Blueprint for Neuroscience Research; and by the McDonnell Center for Systems Neuroscience at Washington University.

[1]Available at https://gitlab.arcos.inf.uc3m.es/aspide/dcex.

processing data across a parallel system. DCEx provides a workflow modeling mechanism that enables to set-up a data life cycle management, allowing data locality and data affinity. The elementary workflow units allow for either placing data close to the computational node where data are processed (data locality), or distributing computation where data was previously generated to avoid data movements (data affinity). This way, the proposed solution supports application developers to access and use resources without the need to manage low-level architectural entities. In the same way, we want to provide a way to easily switch among different execution modes or policies without requiring to modify applications source code. DCEx contributes and differs to other similar frameworks in the following way:

- Its header-only implementation enables the generation of native binary code. Indeed, DCEx does not require wrappers such as PySpark or Dask [1].
- DCEx supports task-based intra/inter-node parallelism in a single framework.
- Our scheduler enables the combination of sequential and parallel tasks in an efficient way.
- DCEx enables the execution of applications in the fields of workflows, batch/stream processing, MapReduce, and scientific computation.
- Due to its C++ implementation, DCEx permits the integration with other solutions in the HPC software stack such as Intel OneAPI, and CUDA.

The paper is structured as follows. Section II presents the architectural model of DCEx and discusses how programming abstractions are mapped to a parallel architecture. In Section III, we discuss a scheduler prototype implemented for the management of data and task locality. Section IV presents the task-based parallel patterns that can be used in DCEx workflows. Section V presents the experimental evaluation of a complex application developed by DCEx. Section VI compares DCEx with other existing frameworks of the literature. Finally, Section VII concludes the paper.

II. DCEx Architectural Design

Aiming at running on massively parallel architectures, the DCEx programming model uses private data structures, exploits data locality, and limits the amount of shared data among parallel processes. The basic idea of DCEx is structuring programs into *data-parallel blocks* (*DPBs*). Blocks are the units of shared- and distributed-memory parallel computation, communication, and migration in the memory/storage hierarchy. Computation processes execute close to the data, using near-data synchronization (as shown in Section III). In the DCEx model three main types of parallelism are exploited: data parallelism, task parallelism (data–driven), and SPMD (Single-program, multiple-data) parallelism.

The main DCEx entities that interact with the runtime are $Master$, $Worker$, $CNode$, $CArea$, $Data\ Parallel\ Block$, $Partition$, $Program$ and $Tasks$. The nodes that compose the system are divided into *Master* and *Worker* nodes. The *Master* node is a coordinator that receives tasks and data from a client machine and distributes them according to the current workload of each *Worker* and the static hints declared by the programmer (see for instance [at `CNode` — `CArea`] annotations). A `CNode` object conceptually represents a single computing/storage node. A `CArea` object is an aggregation of `CNode` objects, where each one is logically near to the other. A `CArea` can refer to many `CNode` objects.

A *Task* in DCEx is the main building block of a parallel *Program*. A *Task* is mainly represented by an object that stores a function pointer and some parameters. The purpose of the function stored in a task, its body, is to read data from some data sources, processing them, and writing results to the output datasets. Identifiers of data sources and output datasets can be stored in the task itself to be passed to its body function when the task is executed. A *data-parallel block* is a collection of data *Partitions* distributed among the parallel system. In general, each *Partition* can be replicated on one or more nodes. For example, we can define a `CArea` *ca*, which is a group of computational nodes, identifying five nodes where can be mapped two data parallel blocks (see Figure 1): “input” and “output”.

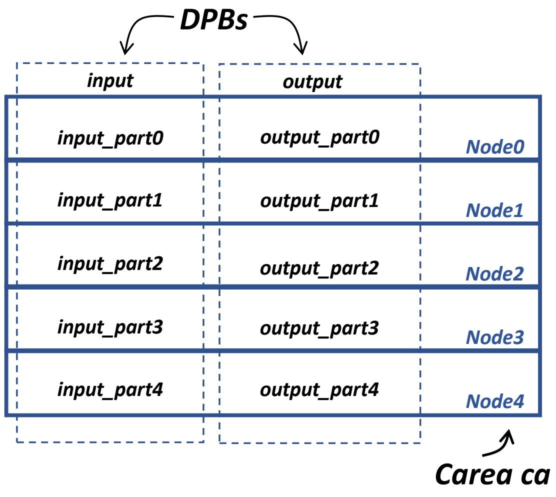


Fig. 1: Two data-parallel blocks storing two files (input and output) split on five computing nodes.

The DCEx runtime receives the code of a parallel application and registers the code that is executed on various $Workers$ by analyzing the inputs that the application must execute. When the execution of a DCEx pattern is requested, for example a pipeline, the runtime queries (to locate where partitions are located) the in object and assigns each one of its partitions to a single $Worker$ (a local task). The $executeRemoteTask$ function (see the figure above) creates a task corresponding to the pipeline and sends it to the best suitable $Worker$, which is chosen by the $partition.getCNode()$ according to the $CArea\ ca$ and to the workload of the node storing the specified partition. For example, a code that works on a single partition will be executed in parallel on each Cnode ($Cnode-0$, ..., $Cnode-n$). Therefore, in $Cnode-0$ the runtime executes a function that analyses the $partition-0$ of the input to generate the $partition - 0$ of the output.

```
class dcex_parallel_execution{
  Carea ca;  istream in;  ostream out;

  dcex_parallel_execution(Carea ca, istream in, ostream out
    ){
    this->ca = ca;  this->in = in;  this->out = out;
  }

  void pipeline(Generator && generate_op, Transformers &&
    ... transform_ops){
    Partition[] parts = in.getPartitions();
    for(part in parts){
      Cnode remoteNode = getBestNode(part.getCNode(), ca);
      executeRemoteTask(remoteNode, generate_op(part), ...
     , transform_ops[out.createPartition(remoteNode)]);
    }
  }
...
}
```

Listing 1: A DCEx code snippet from the dcex_parallel_execution class.

III. Partitioned, scalable task-based scheduler

The DCEx scheduler mainly follows the same principles of Big Data frameworks such as Apache Hadoop and Apache Spark. In that sense, the $Master$ *Cnode* acts as a global scheduler. $Master$ receives the information about pending tasks and dispatches them to *Workers* that are part of the *Carea*.

To describe the internals of the proposed scheduler, first, we define a *task* as a function that is applied to a given input data to produce an output. We distinguish between two different task types: sequential and parallel. Sequential tasks are those that represent the execution of a non-pure/*State-full* function, i.e. tasks that store any kind of state that affects the function execution. These tasks, since they need to maintain their state among executions, cannot be moved or executed over multiple *Workers* in parallel. Therefore, to be scheduled, a sequential task is assigned beforehand to a given *Worker* and all its instances run sequentially on that *Cnode*. On the other hand, parallel tasks represent pure/*Stateless* functions (i.e., functions that do not depend on state variables nor produce side-effects) when they run in parallel over different data items. Therefore, multiple instances of these tasks can be run on different *Workers* simultaneously.

To support task transfers, we have designed a scheduler as an independent thread that deals with requests coming from the *Worker* entities. This thread is also composed of three different task queues depending on the task-type and its data dependencies. Figure 2 depicts the scheduler composition.

The DCEx scheduler is composed by the following task queues; *Sequential*, comprised of a different queue for each node that stores the ready tasks that should be executed in series; *Parallel*, comprised also by a set of queues, one per node, storing the tasks corresponding to the node that has local access to the associated data; and *Not-ready*, a list of tasks that have unresolved dependencies with any other task.

To execute pending tasks, we employ a pool of workers that constantly query the scheduler for new tasks. Regarding the communication among worker threads and the scheduler, we have defined a set of messages to support task transfers and to provide task metadata to the scheduler:

- *Run*: This message is sent from the main process to the scheduler to launch tasks of a given parallel pattern. This message is acknowledged back as soon as all the tasks have finished their execution. This way, this message acts as a barrier for the main process to wait until a pattern execution finishes.
- *GetTask*: sent from a worker executor to the scheduler to request a new task.
- *SetTask*: requires the scheduler to create a new task to be scheduled.
- *Consume*: notifies the scheduler that a task has already finished. Additionally, when the scheduler receives this message, it checks the non-ready queue to annotate the solved task dependencies. If a non-ready task has all its dependencies resolved, it is moved to the corresponding ready queue (sequential or parallel).

Note that the scheduler has been designed to be an interchangeable service module that delivers pending tasks to the workers using task metadata and monitoring information for exploiting computing resources and data locality.

In order to run an application using the DCEx scheduler, we have defined three different procedures: i) initialization, which establishes the communication channels for the executing workers; ii) Task generation, which determines how task are generated in the system; iii) a task scheduling algorithm to orchestrate the execution of tasks.

A. Initialization

In order to support task communication among multiple *Workers* deployed in a *Cnode*, firstly it is necessary to assign communication ports for all necessary queues. *Workers* acts as port servers. Port servers employ a user-defined port. Then, each worker has to publish the listening port on the port server, while any process that wants to connect to a server must get the listening port by means of the same port server. Next, after deploying the port server, *Workers* deploy a local scheduler that performs the following operations:

1) Each local scheduler establishes its working *Carea* by receiving the list of hostnames of the *Carea* nodes as part of the scheduler creation arguments.
2) The local schedulers belonging to the same *Carea* establish a connection to the global scheduler.
3) Local schedulers on each *Cnode* deploys a set of working threads acting as a pool of execution entities. These execution entities are in charge of requesting pending tasks to the scheduler and execute those tasks assigned by the scheduler.

B. Task generation procedure

To create all the required tasks, we first register the functions to be executed that are classified as:

- Parallel tasks: Those functions that are implicitly defined as a parallel pattern are registered as parallel task. Therefore, the scheduler may assign different task instances to different *Workers* so they can be executed in parallel; and

- Sequential tasks: Those functions that are not defined as a parallel pattern are registered as sequential. Then, these functions are pre-assigned to a given worker that runs sequentially all the task instances of that function.

Additionally, in order to avoid the congestion by creating an excessive number of pending tasks at the same time, we also distinguish two kinds of tasks:

- Generator tasks: Those tasks that can generate two or more new tasks after finishing their execution are considered generator tasks. The execution of those tasks is stopped if the scheduler reaches the maximum number of pending tasks at a given point.
- Non-generator tasks: Tasks that generate one or zero new tasks after finishing are classified as non-generator. Therefore, a non-generator task reduces the total number of pending tasks by one after finishing its execution.

C. Task scheduling algorithm

At the execution time, worker threads ask the scheduler for new tasks to be executed. Then, the scheduler decides which task should be delivered to that worker thread following the next algorithm.

First, the scheduler assigns to the requesting thread a sequential task mapped to that *Worker*. Notice that, if any thread of that worker is running an instance of the same task, other task's pending instances are marked as not-ready.

Second, if there are no sequential tasks pending for that *Worker*, the scheduler looks for a parallel task whose associated data is locally stored on the *Cnode* of the requesting *Worker*. To do so, we take advantage of the data location information, specified by the DCEx model, which is defined by the list of Cnodes that store data. Therefore, when a parallel task is created, it is enqueued on the parallel queue corresponding to the local *Cnode* where its related data are stored. If there are multiple Cnodes with local access to that data (e.g., data replication in HDFS), the task is stored on all of them as a unit. Thus, when one of the replicas is consumed, all copies are removed from all the queues. Therefore, when a worker requests a new task and there are tasks stored on its corresponding queue, the scheduler assigns to that worker the first task on that queue in order to ensure that data for the task is locally stored on that *Cnode*. Otherwise, if there are no tasks ready to be executed due to data are locally stored on the resulting *Cnode*, the scheduler does the best to assign another task that has its data stored elsewhere and it moves the data from the remote node to the local one. If data associated with a given task is temporal (i.e., it is handled by the temporary data service), data can be freely moved between the nodes belonging to the *Carea*. However, if data are associated with a DPB, the scheduler checks the *Carea* assigned to that data. If both task and DPB are part of the same *Carea*, data are moved/forwarded between the remote and the local node. Otherwise, data are moved only if the DPB policy is defined as $SOFT$.

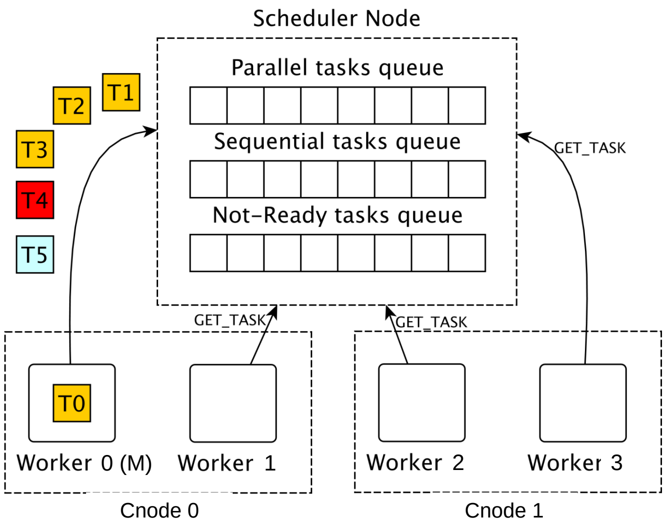


Fig. 2: Task generation sequence.

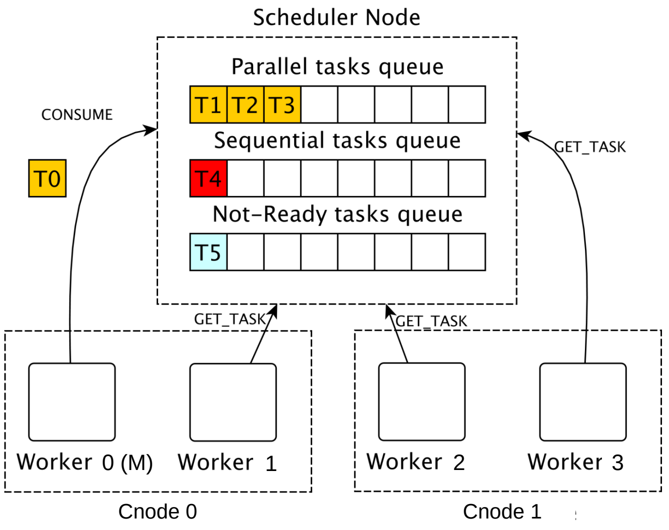


Fig. 3: Task finalization sequence.

If no task can be assigned to the requesting *Worker*, using the aforementioned criteria, the *Worker* waits until a new task is created. Basically, the scheduler annotates that this *Worker* is pending for new tasks. Note that, if a task is a task generator, before assigning it, the scheduler checks if the maximum number of pending tasks has been reached. In that case, those tasks are not assigned until there is space for the new ones, which is generated after its execution.

Finally, as soon as the workers end the execution of the received tasks, they communicate that event to the scheduler using the `CONSUME` message. When this message is received, the scheduler checks if the finished tasks resolve data dependencies of some non-ready tasks. If so, those dependencies are marked on the dependent task, and as soon as all the dependencies of a non-ready task are resolved, those tasks are moved to the corresponding ready queue.

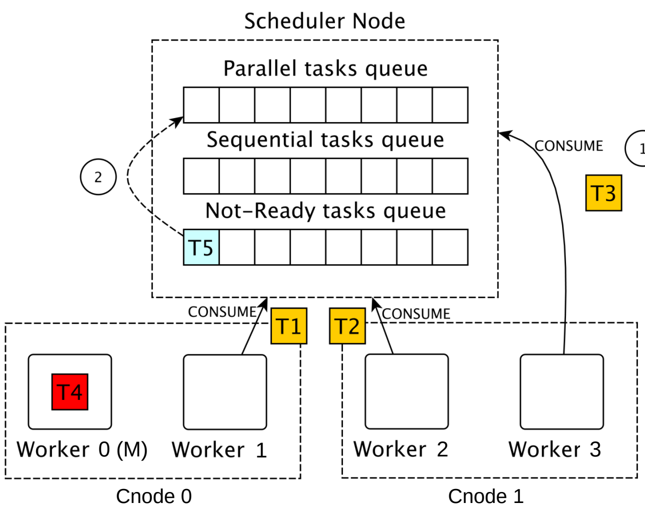


Fig. 4: Dependency resolution.

As an example, Figures 2, 3 and 4 graphically represents the scheduler execution process. Master worker is located in *Worker* 0. The example topology consists of four *Workers* distributed into two *Cnodes*. First, the application launches the initial stage that will be a task generator. That task is assigned to a single worker that may generate several new tasks, which will be introduced in the corresponding scheduler queues. Figure 2 depicts how the execution of the initial task generates 5 new tasks that are communicated to the scheduler using the `SET_TASK` message. Those new tasks are allocated in the corresponding queue depending on the nature of each task. On the other hand, the worker communicates to the scheduler that the task `T0` has finished its execution (see Figure 3). As soon as these tasks arrive, they will be dispatched to the inactive workers following the algorithm presented above. In this example, T5 has unresolved dependencies, thus it cannot be assigned to a worker until those dependencies are satisfied. On the other hand, T4 is a sequential task that is assigned to the *Cnode 0* as it may have some status information that prevents multiple instances of the task from running in parallel. Tasks T1-T3 can be freely assigned to any worker and, using the data location information, they are allocated to the most suitable *Cnode*. Figure 4 shows the two steps followed for resolving data dependencies of a non-ready task. First, *Workers* 2–4 notify the completion of Tasks T1–T3 using the `CONSUME` message. Those messages are received by the scheduler that tags T5 data dependencies as resolved. Then, as a second step, Task T5 is moved from the non-ready queue to the parallel task queue, as all its dependencies are satisfied.

D. Temporary data service

To communicate internal temporary data, which are not backed up by the data, we have designed a temporary data service. This service is only in charge of storing in-transit data among tasks. In other words, only temporary data that may need to be used by a task and can be transferred to a different node will be stored in this temporary data service. Each *Cnode* will run a thread acting as the data server to support this data communication. This data server is composed of a remote-access indexed array that stores the data serialized. Thus, data can be easily transferred to a different *CNode* if it is necessary. Basically, a temporary data server behaves as a DPB by providing the *data.get* and *data.set* functions. At the beginning of the application execution, a temporary data server is deployed on each *CNode* and these servers establish a communication channel with all the other processes involved in the execution.

Depending of the parallel pattern employed, it is necessary to exploit the benefits of DPBs. For example, this is the case of the parallel Map-Reduce pattern. More precisely, this parallel pattern requires an intermediate shuffle stage that enable the data exchange between participant tasks. In the Big Data domain, this problem is addressed by using temporal intermediate files that are access by the reduce stage, as it is done in popular frameworks like Apache Hadoop. DCEx copes with this problem in a similar way. Internally, *map* and *reduce* stages are communicated using temporal files, which are abstracted by using DPBs. End-users can determine the location of the temporal files. In contrast to explicit intermediate containers, implicit data are removed once the pattern is completely executed, as data are considered temporal.

E. Execution model

The DCEx framework is supported by *MPI+X* framework. DCEx is fully compatible with MPI implementations like MPICH or OpenMPI. The following example shows how to execute a DCEx-based application in a classical cluster platform:

```
$ mpiexec -n 40 ./string2bin_mpi
  hdfs://dataset file://output
```

This command example shows how we take advantage of MPI to deploy 40 processes. In the current version of the prototype, computational resources and their topology are obtained from a configuration file. This file has the same format that those employed in *MPICH* software. Each line of the file represents a computational node. Optionally, it is possible to define the number of cores in each node. This format determines the topology and process distribution done at execution time. It is important to note that end-users do not need to implement MPI-basic communication routines in developing parallel code. MPI facilitates the deployment of parallel processes by using environment variables that are obtained at execution time. Other alternatives have been considered, but MPI is the most popular solution usually available in supercomputers. Additionally, DCEx also supports the *Slurm* workload manager.

IV. Parallel programming abstractions/patterns

In order to provide a generic interface to parallel patterns, in a previous work we designed GrPPI, a generic and reusable parallel pattern interface for C++ applications [2]. This interface takes full advantage of modern C++ features, metaprogramming concepts, and generic programming to act as a switch between parallel programming models such as OpenMP or TBB. Its design allows developers to make use of the aforementioned execution frameworks from a unified and compact interface. Furthermore, their modularity permits combining them to arrange more complex constructs (as shown in Section V). In this work, we provide a distributed task-based execution policy that enables the parallel execution of applications relaying on a MPI-based environment. This section describes task-based parallel patterns like `Farm`, `Task`, `Split-join`, `Critical` and `Path generator` designed to provide application developers with structured constructs for developing application workflows in the DCEx model. DCEx also includes other patterns not described in this paper such as `Pipeline` and `MapReduce`. In particular, the `MapReduce` pattern in DCEx works in a similar way to Apache Spark.

A. Farm

This distributed parallel pattern is designed to identify a task as parallel across multiple workers. A farm pattern used as a stage of a pipeline represents a function that can be applied over different data items in parallel without side-effects. This pattern receives as argument a function or another pattern that transforms the input data item to produce a given output data item. When used as part of a pipeline, the function will be registered as a parallel function, and therefore multiple *Workers* can execute this function over different data items.

B. Command

This pattern is designed to provide a way to introduce external programs, as part of DCEx pipeline stages (i.e., executes any external application, script, or command as a task of the pipeline). This pattern interface receives as an argument a single function to generate the external process to be executed for each task. Listing 3 depicts an example of a DCEx pipeline using the command pattern as one of its stages. This basic application generates a sequence of numbers from 0 to the value of the `last` variable. Then, those numbers are passed as input to the next stage that generates the command to be executed. In this case, the function provided by the user for the task pattern will print the number received as argument and will execute the hostname call. Therefore, the result of executing each task will print a number and the hostname of the node that executes the task. Additionally, the task pattern is composed with a farm pattern determining that each task has no data dependencies apart from the data generated in the previous stage. That means that multiple instances of the task can be executed in parallel. In a nutshell, this pattern allows application developers to define application flows and scheduling the execution of each task.

```
pipeline(
  [&i,last]()->optional<int> {
     if(i<last) return i++;
  else return{};
  },
  split_join(duplicate{},
  [](int id){
     return id+1;},  //Branch 1
  [](int id){
     return id+2;}), //Branch 2
  [](int id){
     cout<<id<<"After join"<<endl;}
)
```

Listing 2: Split-join pattern example.

```
pipeline(
  [&i,last]()->optional<int> {
       if(i<last) return i++;
       else return{};
  },
  task([](int i){
       return "printf '" + to_string(i) + "' && hostname";}
)
```

Listing 3: Command pattern example.

C. Split-join

This pattern allows introducing a point to diverge the execution flow into several flows and then merge it again after executing the corresponding flows. The interface of this pattern receives as arguments a split policy and the list of flows to be executed. We support the duplicate and round-robin policies that determine how the input items are segregated among the different flows. The duplicate policy creates a copy of the incoming items and initiates all the execution flows. On the contrary, the round-robin policy executes a single flow for each item following a round-robin policy. Regarding the list of flows, each of them can be a single task or a pattern composition, for instance, a given flow can be defined as a pipeline to execute a set of consecutive tasks. After executing the corresponding flow(s), the resulting data are joined to generate a tuple containing the output of all the flows. Thanks to this pattern, the developers can define kind of directed acyclic graphs.

Listing 2 shows an example of pattern compositions using the split-join. In this case, the first stage generates a sequence of numbers from 0 to the value of `last`. Afterward, the split-join pattern generates the multiple execution paths for the inputs using the *duplicate* split policy. Duplicated tasks are executed in parallel as soon as the input data becomes available and, after all of them have finished, the pattern launches the task defined after the joining point. In other words, the task after the join will have data dependencies with the corresponding flow execution.

D. DPB stream generator

This abstraction works as a stream generator for *DPB*s, which forwards a specific DPB to the next stage of the pipeline. This solution enables the simplification of dealing with complex data path structures. Currently, we provide support for the following DPB stream generators: *all*: returns all possible file/folder paths inside a container at the first level of recursion; *filter*: returns a filtered list of file/folder paths inside a container at the first level of recursion; *locale*: considers the data locality of the path and the execution *Cnode*. This path generator enables data locality by selecting coexisting data and *Cnodes*.

*DPB*s are supported by containers [3]. Containers provide a generic I/O interface that unifies the access of parallel file systems in a single I/O interface. The use of containers hide the complexity of accessing different storage patterns (single file, multiple files in a single folder, etc) by enabling data accesses using iterators. These containers are high-level abstractions that provide a common interface, with a simple set of operations, to manage data items that belong to a dataset or a collection of datasets. In addition, they are designed to expose metadata that can be used to improve data locality or perform task scheduling in runtime. They can be categorized depending on their behavior: input, those used to read data, and output, the ones for writing data. Listing 4 Line 2, we show how a new DPB is created, indicating an URI of the data source. In this case DPBs correspond to files and folders inside this folder path.

An example of the exploitation of a DPB stream generator is shown in Listing 4. In Line 3, we prepare a workflow execution by using a pipeline parallel pattern. This pipeline is composed of a farm and a simple lambda. In this case, we define as the first stage of the pipeline an all-based path generator. This stage returns each path inside the previously created binary container. It is important to note that DPB's containers provide an iterator-like vision, which simplifies the file path generation. In case of running path generators in a fully parallel runtime (i.e., MPI), each deployed process takes a subset of DPBs.

```
1 parallel_execution_dist_task exec{};
2 dpb f("file://home/aspide/data/");
3 pipeline( exec, all(f), [](string & path)  {
4         cout <<  "Created " + path  << endl;}
5 );
```

Listing 4: DPB generator for pipelines.

V. Evaluation

For the evaluation of DCEx, we have implemented a real-world use case that analyzes diffusion-weighted magnetic resonance imaging (DWI) data in a neuroimaging research context. DWI aims to characterize the water diffusion in different brain areas, allowing to obtain unique metrics for the study of white-matter microstructure and structural connectivity of the brain. DWI data are 4-dimensional images composed from tens to hundreds of volumetric (3-dimensional) acquisitions of the brain, each one providing distinct diffusion information. Each 3-dimensional acquisition is composed of the recorded signal of around one million volumetric pixels (voxels). This leads to DWI images ranging from hundreds to thousands of MB for each participant.

The pipeline proposed for the evaluation, depicted in Figure 5 and its implementation in Listing 5, is a typical DWI processing workflow. This workflow generates for each participant diffusion-tensor maps (more sophisticated white-matter microstructure maps), white-matter fiber orientation distribution functions (FODs), and an estimation of white-matter tracts (tractography). Inputs for this workflow are a preprocessed NIfTI 4D image containing the DWI data, two plain text files containing the diffusion weighting information, and a preprocessed NIfTI 3D T1w image already coregistered to the DWI data. We use neuroimaging data from the open-access Young Adults Human Connectome Project (YA-HCP) database[2] [4]. This dataset contains high-quality, high-resolution DWI and structural brain data of over one thousand healthy young adults. The high resolution of the images of this dataset translates into large image files that pose a real processing challenge. Furthermore, the YA-HCP database provides ready-to-use preprocessed data [5].

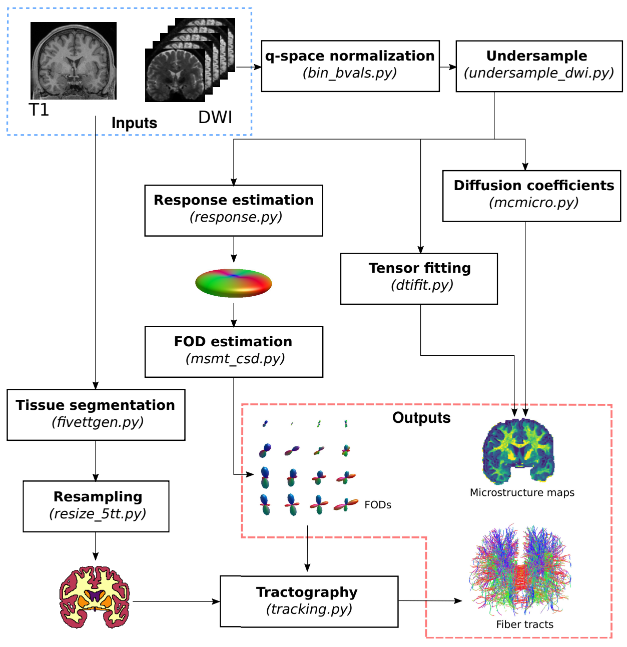
The baseline processing workflow was initially implemented, on the one hand using the Python package *Nipype*[3] [6], a library that provides interfaces to the most used neuroimaging software, simplifying the interconnection between packages, and provides several execution plugins, including SLURM cluster and multithreaded executions. On the other hand, the workflow is implemented using the DCEx to orchestrate the execution of different processing steps consisting of Python scripts. The processing steps of the proposed workflow use processing commands from different state-of-the-art neuroimaging toolboxes:

- Q-space normalization: The diffusion weighting values stored in a text file are binned and saved in a new text file. This is done using the NumPy python package.
- Undersample: The DWI data (288 volumes) is loaded; we select the first 139 and save them in a new NIfTI image. Text files with diffusion-weighting data are modified accordingly. This is done using the nibabel and NumPy packages.
- Tissue segmentation: T1w image is segmented into five tissue types [cortical grey matter (GM), subcortical GM, white matter (WM), cerebrospinal fluid (CSF), and abnormal tissue]. This is done using MRtrix3's `5ttgen` script that uses FSL's `fast` tool [7]. The result is stored in a 4D NIfTI image.
- Resampling: The tissue-type segmentation is resampled to the DWI data resolution. The result is stored in a 4D NIfTI image.
- Response estimation: Voxels representative of CSF, GW and one-bundle WM are selected from the DWI data. Then, the diffusion-weighting response of the different tissues is estimated. This is done using the *dhollander* algorithm implemented in MRtrix's `dwi2response` [8]. Results are stored in plain text files.
- FOD estimation: DWI data and DWI response for each tissue are used to estimate the FODs for every voxel employing the multi-shell multi-tissue constrained spherical deconvolution algorithm implemented in MRtrix3's `dwi2fod` tool [9]. The computation is done independently for every voxel, and the tool allows for **multi-threading**. Results are stored in the 4D NIfTI images.
- Tensor fitting: Diffusion tensors are estimated in every voxel to compute most common diffusion tensor imaging

[2]https://db.humanconnectome.org/
[3]https://nipype.readthedocs.io/

metrics. This is done using FSL's `dtifit` tool. The computation can be performed independently in each voxel, but `dtifit` does not offer a **multithreading** implementation. Results are stored in several 3D NIfTI images.
- Diffusion coefficients: DWI data is used to estimate several WM microstructure metrics. This is done using the spherical mean technique as implemented in SMT's `fitmcmicro` tool [10]. Estimation can be performed in each voxel independently and the used tool allows for **multithreading** computation. Results are stored in several 3D NIfTI images.
- Tractography: Using the estimated WM FODs, and the tissue type segmentation, one million streamlines representing the WM fiber bundles are computed. This is done using the *iFOD2* algorithm implemented in the MRtrix3's `tckgen` tool [11]. Each streamline computation can be performed independently and `tckgen` allows for **multithreading** computation. Results are stores in binary files.

The baseline processing workflow was initially implemented using the Python package *Nipype* [6], a library that provides interfaces to the most used neuroimaging software, simplifying the interconnection between packages, and provides several execution plugins, including SLURM cluster and multithreaded executions. Then, the workflow was implemented using the DCEx to orchestrate the execution of different processing steps consisting of Python scripts.



T1
Inputs
DWI
q-space normalization (bin_bvals.py)
Undersample (undersample_dwi.py)
Response estimation (response.py)
Diffusion coefficients (mcmicro.py)
Tensor fitting (dtifit.py)
FOD estimation (msmt_csd.py)
Tissue segmentation (fivettgen.py)
Outputs
Microstructure maps
FODs
Resampling (resize_5tt.py)
Tractography (tracking.py)
Fiber tracts

Fig. 5: DWI workflow.

```
auto resize_5tt = [](string path) {
  auto in  = path+"/5tt.nii.gz";
  auto out = path+"/resized_5tt.nii.gz";
  return "python resize_5tt.py "+in+" "+out;
};
...
auto inner_split=split_join(duplicate{},
  farm(task(response)),
  farm(task(par, dtifit)),
  farm(task(par, mcmicro))
);

pipeline(exec,
  all(folders),
  task([](string path){
    return "mkdir -p "+path;}),
    split_join(duplicate{},
    pipeline(farm(task(par, fivettgen)),
             farm(task(resize_5tt))),
    pipeline(farm(task(bin_bvals)),
             farm(task(undersample_dwi)),
             inner_split,
             farm(task(par, msmt_csd)))),
    farm(task(tracking))
);
```

Listing 5: Implementation of the DWI workflow. The optional tag *par* indicates that the task is intrinsically parallel (i.e., OpenMP, GPU, etc).

A. Results

The experiments were carried out in a bare metal cluster composed by 24 Intel(R) Xeon(R) Silver 4214 and 128 GB of RAM memory each. The software layer is based on Ubuntu 18.4, GCC compiler 9.3, and MPICH 3.2.0. Datasets are shared using NFS and GlusterFS filesystems with a 10 Gbps network. We have chosen a random subset of 32 subjects/patients from the YA-HCP database. The results shown in the experiments correspond to the average value of five consecutive executions. Focusing on reproducibility, Python-like scripts and source code are also provided.

Table I summarizes the execution time and lines of code (LOC)[4] with Nipype/Slurm using one single *Cnode*. We observe that our solution significantly reduces the overall execution time. It employs a higher number of resources compared with Nipype. DCEx enables the use of over-subscription, allowing more active threads than available cores. Experiments did not show a significant overhead using it. It is worth to note that in case of a single compute node, for native multi-threaded stages (i.e., *tracking*, *mcmicro*), the farm pattern has been removed in order to enable specific shared-memory scheduling. In the future, we plan to enable an adaptive mechanism to guide the scheduling process by indicating the expected number of running threads.

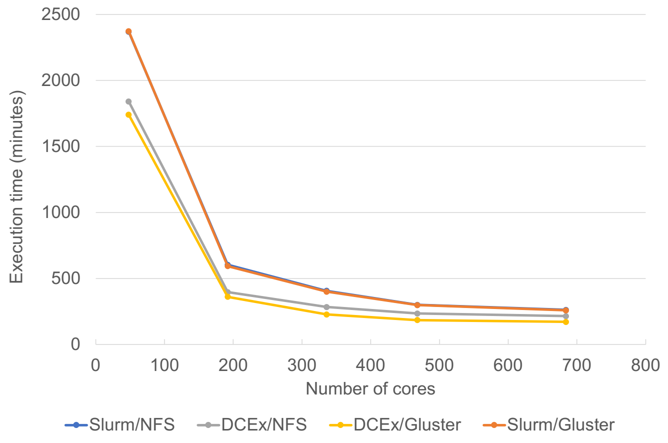
We observe that DCEx outperforms Nipype in all cases. The improvement ratio is about 25% compared to Nipype/Slurm for a single subject and 13% for 4 subjects. Additionally, our approach significantly reduces the number of lines of source code (see repository). We highlight that DCEx does not require an external workload manager (i.e., Slurm, SGE, etc.) for a distributed execution. In the case of Nipype, every stage is directly translated into a Slurm job and the resource utilization is determined by Slurm. DCEx can contribute and assist Slurm with data locality mechanisms if no other solution is offered.

Figures 6a and 6b plot the overall execution time by increasing the number of used cores from 24 to 780 cores,

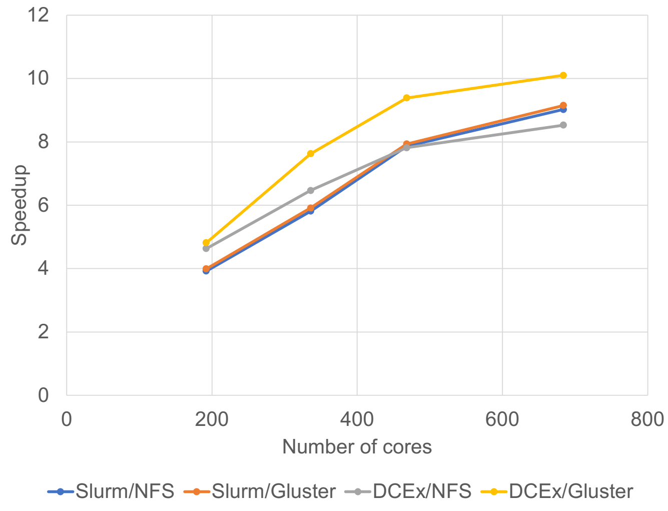
[4]Line breaks are not considered.

TABLE I: Overall execution (in minutes) and LOC comparing Nipype/Slurm (baseline) and DCEx on a single *Cnode*.

	Cores per *Cnode*	Subjects	Execution (min)	LOC
Nipype	max 48 per task	1	74	183
Nipype	max 6 per task	4	218	
Nipype	max 48 per task	4	231	
DCEx	48	1	59	102
DCEx	48	4	203	



(a) Overall execution time.



(b) Speedup.

Fig. 6: Comparison between DCEx and Nipype/Slurm under NFS and GlusterFS. We used a subset of 32 random subjects from the YA-HCP database.

comparing Nypipe/Slurm and DCEx. Figure 6b shows that the speedup trend decreases when more cores are used for all configurations, mainly because the system infrastructure based on NFS negatively affects the application performance. It is important to highlight that each subject produces 8 GB of data approximately. However, we must point out that DCEx outperforms Nipype/Slurm due to its locality policies and file system cache exploitation. We expect that a better setup based on a HPC-based filesystem, like Lustre or an in-memory filesystem [12], can result in a more positive impact on the execution time.

VI. Related work

The message passing interface (MPI) standard is the most common approach to exploit inter-node parallelism in HPC, and is the basis for numerous runtimes and workflows for scientific computing. The implementations of MPI allow the execution of standard operations comprising multiple processes on distributed memory platforms, which provides coarse-grained parallelism sufficient for petascale applications. Current HPC infrastructures have incorporated different types of accelerators to enhance the performance. Programming models have been adapted accordingly to ease the access to further finer-grain intra-node parallelism. GPGPUs are the most widely adopted accelerator in current HPC machines given their power efficiency and their many-core architecture, which pushes forward massive parallelism to the order of thousands of cores in a single chip. There are several libraries that enable the interaction with GPGPUs, such as OpenCL, NVidia CUDA, and OpenACC, supporting data offload to the accelerators, kernel operator definition, direct execution of such code on the device, and result retrieval back to the host CPU. Accelerator runtimes have also been integrated with intra-node parallelism through OpenMP [13], and inter-node parallelism via MPI [14], [15]. The mechanisms to build hybrid runtimes exploiting intra- and inter-node parallelism had major influence in subsequent advances in parallelism integration and they are expected to be present in future Exascale systems to cope with the need for adaptive hybrid programming models [16]. Examples of MPI-based MapReduce implementations are Spark-DIY [17] and Harp.

In the literature, we can find other programming models that support task-based parallelism, such as COMPSs/ServiceSs [18]. COMPSs/ServiceSs provides a programming model and an execution framework that help in abstracting applications from the actual execution environment using MPI. Our approach offers advantages over COMPSs/ServiceSs such as the use of distributed and parallel patterns.

Finally, a similar work was presented by Aldinucci et al. [19] that provided a FastFlow extension aimed at supporting distributed memory architectures. This approach differs from ours in multiple ways. First, DCEx addresses large-scale workflow-based applications following a FPGA model, which enables the elimination of explicit data sharing between computing elements. Second, the DCEx scheduler considers both sequential and parallel stages, enabling a greater range of optimization. Third, DCEx provides a user-friendly interface (as shown in Listing 5) that reduces the development complexity and facilitates the implementation of complex execution flows due to pattern's composability.

VII. Conclusions

In this paper we presented a novel data-aware paradigm called Data Centric programming model for Exascale systems (DCEx) and runtime mechanisms for data intensive applications. DCEx provides a workflow-based programming model that enables to set-up a data-oriented life cycle management, allowing data locality and data affinity. Big Data/HPC convergence is addressed in multiple ways. First, native code can take advantage of compiler optimization such as factorization and vectorization. Second, we facilitate the integration with accelerators given that the implementation is based on C++.

Moreover, the DCEx implementation offers a novel runtime system that controls and optimizes the execution of the component-based use-cases and applications. The runtime system is responsible for managing, coordinating, and scheduling the execution of an application by deciding when, where and how its constituent components should be executed by using a data-locality aware scheduling strategy for the execution of large-scale task workflows. We executed several experiments with a large medical neuroimage workflow, comparing Nipype (a tool broadly used in this area) with DCEx for performance and scalability. Results showed that DCEx outperforms Nipype for both metrics.

In the future, we plan to extend DCEx functionally by supporting well-known parallel architectures such as GPUs. In addition, it is planned to include an intermediate in-memory filesystem subsystem to store temporal data used by future stages or just for enabling data persistence (intensive traceable applications). Finally, we will investigate the cache effects over parallel file systems such as Lustre [20] and GPFS, which offer an efficient file cache at client side.

References

[1] M. Rocklin, “Dask: Parallel computation with blocked algorithms and task scheduling,” in *Proceedings of the 14th Python in science conference*, vol. 130. Citeseer, 2015, p. 136.

[2] J. Garcia-Blas, D. del Rio, J. D. Garcia, and J. Carretero, “Exploiting stream parallelism of MRI reconstruction using GrPPI over multiple back-ends,” in *Workshop on Clusters, Clouds and Grids for Life Sciences, CCGRID-LIfe 2019, CCGRID 2019*. Larnaca, Cyprus: IEEE Explorer, 2019.

[3] P. Brox, J. Garcia-Blas, D. E. Singh, and J. Carretero, “DICE: Generic Data Abstraction for Enhancing the Convergence of HPC and Big Data,” in *High Performance Computing*, I. Gitler, C. J. Barrios Hernández, and E. Meneses, Eds. Cham: Springer International Publishing, 2022, pp. 106–119.

[4] D. Van Essen, K. Ugurbil, E. Auerbach, D. Barch, T. Behrens, R. Bucholz, A. Chang, L. Chen, M. Corbetta, S. Curtiss, S. Della Penna, D. Feinberg, M. Glasser, N. Harel, A. Heath, L. Larson-Prior, D. Marcus, G. Michalareas, S. Moeller, R. Oostenveld, S. Petersen, F. Prior, B. Schlaggar, S. Smith, A. Snyder, J. Xu, and E. Yacoub, “The human connectome project: A data acquisition perspective,” *NeuroImage*, vol. 62, no. 4, pp. 2222–2231, 2012, connectivity. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S1053811912001954

[5] M. F. Glasser, S. N. Sotiropoulos, J. A. Wilson, T. S. Coalson, B. Fischl, J. L. Andersson, J. Xu, S. Jbabdi, M. Webster, J. R. Polimeni, D. C. Van Essen, and M. Jenkinson, “The minimal preprocessing pipelines for the human connectome project,” *NeuroImage*, vol. 80, pp. 105–124, 2013, mapping the Connectome. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S1053811913005053

[6] K. Gorgolewski, C. Burns, C. Madison, D. Clark, Y. Halchenko, M. Waskom, and S. Ghosh, “Nipype: A flexible, lightweight and extensible neuroimaging data processing framework in python,” *Frontiers in Neuroinformatics*, vol. 5, p. 13, 2011. [Online]. Available: https://www.frontiersin.org/article/10.3389/fninf.2011.00013

[7] R. E. Smith, J.-D. Tournier, F. Calamante, and A. Connelly, “Anatomically-constrained tractography: Improved diffusion MRI streamlines tractography through effective use of anatomical information,” *NeuroImage*, vol. 62, no. 3, pp. 1924–1938, 2012. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S1053811912005824

[8] T. Dhollander, D. Raffelt, and A. Connelly, “Unsupervised 3-tissue response function estimation from single-shell or multi-shell diffusion MR data without a co-registered T1 image,” in *ISMRM Workshop on Breaking the Barriers of Diffusion MRI*, 09 2016.

[9] B. Jeurissen, J.-D. Tournier, T. Dhollander, A. Connelly, and J. Sijbers, “Multi-tissue constrained spherical deconvolution for improved analysis of multi-shell diffusion mri data,” *NeuroImage*, vol. 103, pp. 411–426, 2014. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S1053811914006442

[10] E. Kaden, N. D. Kelm, R. P. Carson, M. D. Does, and D. C. Alexander, “Multi-compartment microscopic diffusion imaging,” *NeuroImage*, vol. 139, pp. 346–359, 2016. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S1053811916302063

[11] J. Tournier, F. Calamante, and A. Connelly, “Improved probabilistic streamlines tractography by 2nd order integration over fibre orientation distributions,” in *Proc. of the International Society of Magnetic Resonance in Medicine*, 2010.

[12] F. Marozzo, F. Rodrigo Duro, J. Garcia Blas, J. Carretero, D. Talia, and P. Trunfio, “A data-aware scheduling strategy for workflow execution in clouds,” *Concurrency and Computation: Practice and Experience*, vol. 29, no. 24, 2017, iSSN: 1532-0634.

[13] J. Guan, S. Yan, and J. Jin, “An openmp-cuda implementation of multilevel fast multipole algorithm for electromagnetic simulation on multi-gpu computing systems,” *IEEE Transactions on Antennas and Propagation*, vol. 61, no. 7, pp. 3607–3616, July 2013.

[14] P. Rakić, D. Milašinović, Z. Živanov, Z. Suvajdžin, M. Nikolić, and M. Hajduković, “MPI–CUDA parallelization of a finite-strip program for geometric nonlinear analysis: A hybrid approach,” *Advances in Engineering Software*, vol. 42, no. 5, pp. 273 – 285, 2011, pARENG 2009. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S0965997810001286

[15] S. J. Pennycook, S. D. Hammond, S. A. Jarvis, and G. R. Mudalige, “Performance analysis of a hybrid mpi/cuda implementation of the naslu benchmark,” *SIGMETRICS Perform. Eval. Rev.*, vol. 38, no. 4, pp. 23–29, Mar. 2011. [Online]. Available: http://doi.acm.org/10.1145/1964218.1964223

[16] M. U. Ashraf, F. A. Eassa, A. A. Albeshri, and A. Algarni, “Performance and Power Efficient Massive Parallel Computational Model for HPC Heterogeneous Exascale Systems,” *IEEE Access*, vol. 6, pp. 23 095–23 107, 2018.

[17] S. Caíno-Lores, J. Carretero, B. Nicolae, O. Yildiz, and T. Peterka, “Spark-diy: A framework for interoperable spark operations with high performance block-based data models,” in *2018 IEEE/ACM 5th International Conference on Big Data Computing Applications and Technologies (BDCAT)*, Dec 2018, pp. 1–10.

[18] A. Fernández, V. Beltran, X. Martorell, R. M. Badia, E. Ayguadé, and J. Labarta, “Task-Based Programming with OmpSs and Its Application,” in *Euro-Par 2014: Parallel Processing Workshops*. Springer, 2014, pp. 601–612.

[19] M. Aldinucci, S. Campa, M. Danelutto, P. Kilpatrick, and M. Torquati, “Targeting Distributed Systems in FastFlow,” in *Euro-Par 2012: Parallel Processing Workshops*, I. Caragiannis, M. Alexander, R. M. Badia, M. Cannataro, A. Costan, M. Danelutto, F. Desprez, B. Krammer, J. Sahuquillo, S. L. Scott, and J. Weidendorfer, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2013, pp. 47–56.

[20] W. Cheng, C. Li, L. Zeng, Y. Qian, X. Li, and A. Brinkmann, “Nvmm-oriented hierarchical persistent client caching for lustre,” *ACM Trans. Storage*, vol. 17, no. 1, Jan. 2021. [Online]. Available: https://doi.org/10.1145/3404190