Data-Centric Parallel Programming Abstractions for High Performance Computations

Domenico Talia[0000-0003-1910-9236]

University of Calabria, Via P. Bucci 41C, Rende 87036, Italy
domenico.talia@unical.it

Abstract. This short paper describes the programming paradigm and the main constructs of the DCEx programming model designed for the implementation of data-centric large-scale parallel applications. The DCEx programming paradigm exploits private data structures and limits the amount of shared data among parallel threads in HPC applications. The key idea of DCEx is structuring programs into *data-parallel blocks* mapped on computing elements and managed in parallel by a large number of parallel tasks. Data-parallel blocks are the units of shared- and distributed-memory parallel computations and communications in the memory/storage hierarchy. Tasks execute close to data using near-data synchronization according to the PGAS model. Two use cases implemented using DCEx constructs are also outlined and performance measures on different parallel machine configurations are shown.

Keywords: Parallel programming, data-parallel applications, data-centric computational science, HPDA.

1 Introduction

Computational science applications use advanced computing capabilities to model and solve complex scientific problems. To reach this goal, appropriate technologies and tools are needed. In particular, parallel computing systems and scalable data management techniques are vital. Nowadays, data-intensive scientific computing systems are widely used for many computational science applications in several domains. The ever more complex nature of the underlying computing infrastructure necessary to run large-scale use cases asks for data-oriented solutions that simplify the development, deployment, and scalable execution of complex computational tasks. Among these solutions, the scientific workflow model is a leading approach for designing and executing data-intensive applications in high-performance computing infrastructures [1].

When data-intensive applications are targeted, as occurs in high-performance data analysis (HPDA), programming frameworks need to limit task synchronization, reduce communication and remote memory access. Although traditional parallel programing tools and libraries, such as MPI, OpenMP and HPF, are being adapted to manage large datasets, we argue that the best approach is to develop parallel programming paradigms specifically designed according to a data-driven style, especially for supporting for big data analysis and machine learning on high-performance computing (HPC) systems [2]. According this approach, new languages such as X10, Legion, and Chapel, have been defined by exploiting a data-centric parallel programming approach.

This paper introduces the main features and the programming mechanisms of the Data-Centric programming model for Exascale systems (DCEx) [3] designed for the implementation of data-centric parallel applications. DCEx include programming mechanisms to improve the performance of data-intensive computations by reducing accessing, exchanging, and processing of data through the computing nodes of a parallel system. DCEx provides a workflow-based model where tasks are executed closed to input data and computation is distributed where data was generated/stored to limit data transfer overhead.

The DCEx functions are based upon data-aware operations specifically designed for data-intensive applications supporting the scalable use of a massive number of processing elements run in parallel for solving computational science applications. The DCEx model is based on private data structures and associated constructs. The goal is to exploit parallelism starting from data artifacts and limit the amount of shared data among parallel threads.

Instead of starting from parallel operations, we argue that starting from distributing data abstractions specifically defined to be operated in parallel is more appropriate in today data-intensive computations. Therefore, the basic idea of DCEx is structuring programs into *data-parallel blocks* (DPBs) that are the basic units of distributed-memory parallelism, like Resilient Distributed Datasets (RDDs) in Apache Spark, around which computation, communication, and scheduling are accomplished. Computation tasks execute close to data, using near-data synchronization based on the partitioning of data on different processing elements where tasks run in parallel. Using the data-parallel blocks, in DCEx, three main styles of parallelism are exploited: data parallelism, SPMD parallelism, and task parallelism. A prototype API based on the DCEx model has been implemented and some experimental evaluations have been performed.

The remainder of this paper is structured as follows. Section 2 presents the main features of the parallel data model used in DCEx. The parallel data block concept is introduced, and data access and processing operations are illustrated together with the associated types of parallelism. Section 3 briefly illustrate two real use cases developed by means of the programming mechanisms of DCEx, showing performance results. Finally, Section 4 concludes the paper.

2 A Data-Centric Parallel Model

Scientific and business applications are becoming more and more data intensive; therefore, data are increasingly playing a centrale role in complex applications. For this reason, the function of data is considered fundamental in the DCEx programming model. In fact, as mentioned in the introduction, the data-centric model used in DCEx is based on the *data-parallel block* (DPB) concept, which defines data structures that are partitioned and distributed on different computing nodes where they are handled in parallel.

Data-parallel blocks are more general data-oriented mechanisms and provide a higher abstraction level with respect to Spark RDDs.

2.1 Data-Parallel Blocks

Data blocks and their message queues are mapped onto tasks to be managed in parallel and are placed in memory/storage units by the DCEx runtime. A DPB is manipulated for managing a composite data element in the main memory of multiple computing nodes. Decomposing a program in terms of block-parallelism, instead of process-parallelism, enables mapping blocks during the program execution among different processors in a parallel computer and execute tasks where data partitions are. This is the main idea that lets us integrate in- and out-of-core programming in the same model. In particular, a DPB `datapb` can be composed of multiple partitions:

```
datapb = [part0][part1][part2]...[partn-1]
```

where each partition is assigned to a given computing node.

Notation `datapb[i]` refers to the i-th partition of DPB `datapb`. However, when a DPB is simply referred by its name (e.g., `datapb`) in a computing node (e.g., the i-th node), it is intended as a reference to the locally available partition (e.g., `datapb[i]`).

A DPB can be created using the `data.get()` operation, which loads into main memory some existing data from secondary storage. This operation is specified as follows:

```
datapb = data.get(source, [format], [part|repl]) at [Cnode|Carea];
```

where the data source, its format, the optional partitioning or replication on a computing node (`Cnode`) or, better, on a computing area (`Carea`) composed of a set of computing nodes, are its parameters.

DCEX also includes the `data.declare()` operation, used to declare a DPB that will be created at run time as a result of a task execution. A DPB can be also written from main memory of processors/cores, where the block has been managed, to secondary storage using the `data.set()` operation, which syntax is as follows:

```
data.set(datapb, dest, [format]);
```

In this case, all partitions are collected in parallel form the processor memories and moved to compose the secondary storage object.

Using these three basic operations, partitions can be mapped on different processing nodes where each task will work in parallel on a given partition. This approach allows computing nodes to manage in parallel the data partitions at each core/node using a set of operations or library APIs that hide the complexity of the underlying actions.

2.2 Parallel operations

The two main types of computing abstractions defined in DCEx to be coupled with parallel data blocks are:

- *computing nodes* and *computing areas* that specify single processing elements or regions of processors of a parallel machine where to store data and run tasks.
- *tasks* and *task pools* that embody the units of parallelism in the model.

The corresponding DCEx constructs defined to refer to these abstractions are: `Cnode` for a single computing node and `Carea` for a region including a set of computing elements. For instance, a `Cnode` can be declared as follows:

```
nod = Cnode([hardware annotation parameters]);
```

whereas an example of `Carea` composed of a two-dimensional array of 30x40 computing nodes can be defined as follows:

```
nar = Carea(30,40);
```

where 30 is the number of rows and 40 is the number of columns of the computing area. A single element of this area can be referred as `nar[8][10]` and a sub-area can be also defined like, for example, `na2 = Carea(nar,6,6);` which extracts a 6x6 matrix of computing nodes from the `Carea` defined by the variable `nar.`
Concerning tasks management constructs, the programming model allows for expressing parallelism using two concepts: *Task* and *Task pool*. A task can be defined as follows:

```
t=Task(func,func_params) [at Cnode|Carea] [on failure ignore|retry];
```

to execute a function on a given computing node or on a node of a computing area. The `on failure` is an optional directive for specifying an action (for instance, `ignore` or `retry` with it) to be performed in case of task failure. Task pool abstraction is defined to implement SPMD parallelism representing a set of tasks that execute the same function. The basic syntax for declaring a pool of tasks is as follows:

```
tp = Task_Pool([size]);
```

where `tp` identifies the task pool and `size` is an optional parameter specifying the number of tasks in the pool. This statement declares a task pool but does not spawn its execution. Tasks in the pool can be activated explicitly using a `for` loop as in the following example:

```
N = 40;
nodes = Carea(N);
for (i=0; i<N; i++) {
    func_param_1 = x;
    tp[i] = Task(func_name, func_param_1) at nodes[i];
}
```

By exploiting the parallel abstractions of DPBs, tasks, and computing areas, DCEx implements three main types of parallelism that can be combined to develop complex parallel applications: *Task parallelism*, *data parallelism*, and *SPMD parallelism*.

Task parallelism is exploited when different tasks that compose an application run in parallel. It is data driven since data dependencies are used to decide when tasks can be spawn in parallel. As input data of a task are ready its code can be executed. Data parallelism is achieved when the same code is executed in parallel on different data blocks or on partitions of a data block. In SPMD parallelism, differently from data parallelism, tasks cooperate to exchange partial results during execution. In DCEx, these three types of parallelism are combined with the features of the Partitioned Global Address Space (PGAS) model [5] that supports the definition of several execution contexts based on separate address spaces that compose a global address space. For any given task, this allows for the exploitation of memory locality and affinity, providing programmers with a well-define way to distinguish between private and shared memory blocks.

To give a quick example of how locality can be exploited in DCEx using data parallel blocks and computing areas, we can define the computing area `ca`, which includes 8 computing nodes where two data parallel blocks (`df1` and `df2`), storing data coming from two files `f1` and `f2`, can be mapped by splitting them in four partitions each (see Figure 1).

3 Two Real-world Use Cases

Different real-world applications have been developed by using the DCEx abstractions integrated with the GrPPI language [4], such as urban computing dynamics, parallel neuroimaging, and deep learning for anomaly detection in electric vehicles [6]. The two we discuss here have been developed to analyze diffusion-weighted magnetic resonance imaging data and trajectory discovery from social data analysis.

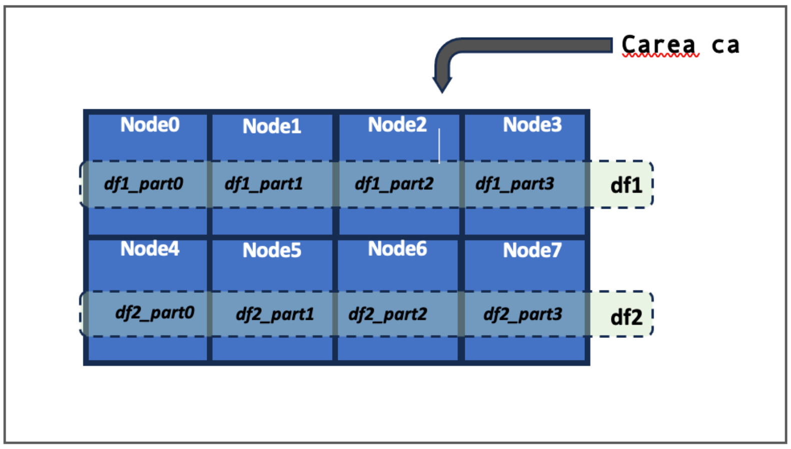


Fig. 1. Two data-parallel blocks storing two files (***f1*** and ***f2***) partioned on 8 computing nodes.

Diffusion-weighted magnetic resonance imaging (DWI) aims to obtain unique metrics for the study of brain white matter microstructure and structural connectivity. DWI data are four-dimensional images composed from tens to hundreds of three-dimensional brain images. Each image acquisition is composed of the signal of around 1 million volumetric pixels (voxels). This leads to DWI images ranging from hundreds to thousands of MB for each patient. The DWI workflow has been implemented using the DCEx abstractions to orchestrate the execution of different parallel processing steps consisting of Python scripts [4]. The processing steps of the DCEx workflow included processing commands from different state-of-the-art neuroimaging toolboxes.

The trajectory discovery application has been developed for extracting frequent patterns from large volumes of geotagged data gathered from social media. The main steps of the application are as follows: (a) parallel crawling, (b) parallel data filtering, (c) automatic keywords extraction and data grouping, (d) Regions of Interest (RoIs) extraction through a parallel clustering algorithm, and (e) Trajectory mining. This final step is based on a highly parallel versions of the FP-Growth (frequent itemset analysis) and Prefix-Span (sequential pattern mining) algorithms.

3.1 Experimental evaluations

The experiments carried out to evaluate the DWI application have been performed on an Intel-based cluster with Xeon processors with 128 GB of RAM memory each. Data have been shared using NFS and GlusterFS filesystems with a 10 Gbps network. Performance results are calculated by averaging five consecutive runs and are compared with a baseline implementation based on the use of the Python package Nipype [7], the SLURM cluster and multithreaded execution.

Table 1 summarizes the execution times using four different parallel configurations ranging from 48 to 648 cores. The DCEx results are compared with those obtained with the Nipype/Slurm implementation. The DCEx based solutions are significantly faster than the one based on Nipype under Slurm.

Table 1. DCEx/ Nipype execution time (in minutes) using different parallel machine setting.

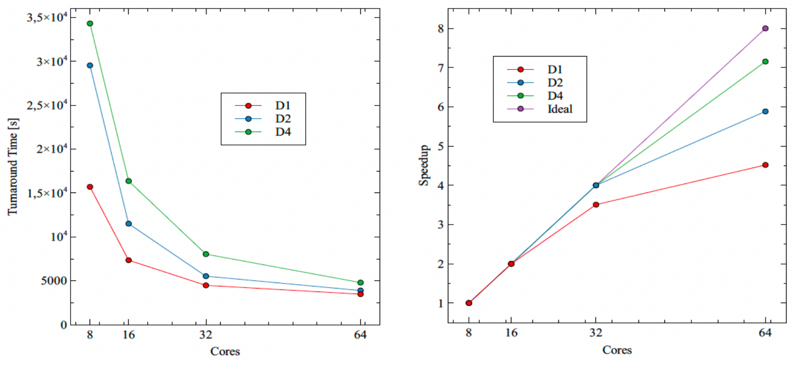
Cores	Nypype/NFS	Nypype/GlusterFS	DCEx/NFS	DCEx/GlusterFS
48	74.00	74.12	57.50	54.37
192	18.87	18.56	12.41	11.29
336	12.72	12.53	8.89	7.13
684	8.21	8.09	6.74	5.38

From this performance experiments, we also noted the effect of data locality in DCEx under NFS and GlusterFS. Strong data caching at data nodes benefits the overall execution time of a workflow execution of the same subject. In the best case, data locality results on 25% of improvement using 684 cores.

Concerning the trajectory discovery application, a set of experiments has been run for evaluating turnaround time and speedup using different number of cores and different sizes of datasets to be analyzed. Figure 2 shows the main results.

Figure 2(a) shows the turnaround times (in seconds) of the application for the three datasets we considered (D1, D2, D4), using from 8 to 64 cores. For the smallest dataset (D1) that contains 1.2GB of data, the turnaround time decreases from 4h 10m using 8 cores to 58 minutes using 64 cores. For D2 (2.4GB) the turnaround time decreases from 6h 21m to about 1h 5m minutes. For D4 (4.8GB) the turnaround time decreases from 9h 32m to 1h 20m.

Figure 2(b) reports the speedup obtained by analyzing the different datasets from 8 up to 64 CPU cores. For dataset D1 the speedup is close to linear up to 16 cores, while it slightly decreases to 3.5 and 4.6 with 32 and 64 cores, respectively. For datasets D2 and D4, the application reached a better speedup, which is close to linear. It is worth to note that the speedup increases as the size of the dataset increases. This means that CPU cores are better used when larger datasets are analyzed.



(a) Turnaround time. (b) Speedup.

Fig. 2. Turnaround time and speedup of the trajectory discovery application in DCEx.

4 Conclusions

Traditional parallel programming languages are not specifically designed for developing data-intensive applications in science and engineering. With the advent of Big Data, machine learning, and generative artificial intelligence, new programming models, languages, and APIs are needed to combine parallel data abstractions with scalability and performance for extreme data processing [8].

To streamline the development of computational science applications in HPC systems, large-scale data- and task-parallelism techniques have to be developed on top of the data-parallel abstractions divided into many partitions mapped on different computing elements where local tasks process them. This approach allows for processing in parallel the data partitions at each core/node using a set of statements/library calls that hide the complexity of the underlying operations. Since data dependency in this scenario may limit scalability, data-centric abstractions help programmers to avoid or limit it to a local/neighbors scale. Scalability of large data analysis, machine learning and AI applications are closely related to the management of parallelism in the data-driven operations needed in the applications and the limitation of overhead created by data processing mechanisms and techniques.

In this paper we illustrated DCEx, as a data-aware parallel programming paradigm for data intensive computational science applications. The designed DCEx programming model includes data-parallel blocks and data-driven parallel tasks for the implementation of scalable algorithms and applications on top of HPC computers, with a special emphasis on the support of massive data analysis applications.

DCEx provides a workflow-based programming model that enables to set up a data-oriented life cycle management, allowing parallel data locality and data affinity. Moreover, the DCEx implementation offers a runtime system that controls and optimizes the execution of the component-based use-cases and applications. We described here the language features and reported on the experimental evaluation of two significant use cases. Results show that DCEx achieves good performance and scalability.

Machine learning and generative AI are going to play a key role in computational science applications as the analysis of scientific data is integrating traditional simulation approaches. AI and machine learning algorithms are becoming powerful tools to accelerate simulations, extract patterns from data, and enhance scientific discovery by bridging data-driven methods with traditional modeling strategies. To support this new trend, scalable parallel programming models and tools using a data-centric approach for exploiting parallelism in data analysis are vital.

Acknowledgments. This study was partially funded by the EU ASPIDE project (grant number 801091) and the "National Centre for HPC, Big Data and Quantum Computing", (grant number CN00000013- CUP H23C22000360005). Author also acknowledges the contribution of colleagues who worked on the DCEx model in the framework of the ASPIDE project.

Disclosure of Interests. The author has no competing interests to declare that are relevant to the content of this article.

References

1. Ejarque, J., et al.: Enabling dynamic and intelligent workflows for HPC, data analytics, and AI convergence. Future generation computer systems 134, 414–429 (2022)
2. Talia, D., Trunfio, P., Marozzo, F., Belcastro, L., Cantini, R., Orsino, A.: Programming big data applications: scalable tools and frameworks for your needs. World Scientific Press, London (2024).
3. Garcia-Blas, J., et al: Convergence of HPC and Big Data in extreme-scale data analysis through the DCEx programming model. In: IEEE 34th International Symposium on Computer Architecture and High Performance Computing (SBAC-PAD) 2022, IEEE, pp. 130-139, Bordeaux (2022).
4. Garcia-Blas, J., del Rio, D., Garcia, J. D., Carretero, J.: 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, IEEE, Larnaca, Cyprus (2019).
5. Stitt, T.: An introduction to the partitioned global address space programming model. CNX.org (2010).
6. Marchesi, A., et al: Paradigms and Models at Run-time - Final Report D2.6. ASPIDE Project (2021).
7. Gorgolewski, K., Burns, C., Madison, C., Clark D., Halchenko, Y., Waskom, M., Ghosh, S., Nipype: A flexible, lightweight and extensible neuroimaging data processing framework in python. Frontiers in Neuroinformatics 5, pp. 13 (2011).
8. Talia D., A view of programming scalable data analysis: from clouds to exascale, Journal of Cloud Computing 8, 4 (2019).