In: Euro-Par 2019 Workshops, Gottingen 2019. August 2019.

A Novel Data-Centric Programming Model for Large-Scale Parallel Systems

Domenico Talia[1]✉*, Paolo Trunfio[1], Fabrizio Marozzo[1], Loris Belcastro[1], Javier Garcia-Blas[2], David del Rio[2], Philippe Couvée[3], Gael Goret[3], Lionel Vincent[3], Alberto Fernández-Pena[4], Daniel Martín de Blas[4], Mirko Nardi[5], Teresa Pizzuti[5], Adrian Spătaru[6], and Marek Justyna[7]

[1] University of Calabria, Rende, Italy
[2] University Carlos III of Madrid, Madrid, Spain
[3] Atos BDS R&D Data Management, France
[4] Instituto de Investigación Sanitaria Gregorio Marañón, Madrid, Spain
[5] INTEGRIS, Rome, Italy
[6] Institute e-Austria Timişoara, Timişoara, Romania
[7] PSNC, Poznan, Poland

Abstract. This paper presents the main features and the programming constructs of the *DCEx* programming model designed for the implementation of data-centric large-scale parallel applications on Exascale computing platforms. To support scalable parallelism, the *DCEx* programming model employs private data structures and limits the amount of shared data among parallel threads. The basic idea of *DCEx* is structuring programs into data-parallel blocks to be managed by a large number of parallel threads. Parallel blocks are the units of shared- and distributed-memory parallel computation, communication, and migration in the memory/storage hierarchy. Threads execute close to data using near-data synchronization according to the PGAS model. A use case is also discussed showing the *DCEx* features for Exascale programming.

Keywords: Large-scale parallelism, Exascale systems, Data-centric applications.

1 Introduction

High-level parallel programming models assist designers accessing and exploiting high-performance computing (HPC) resources abstracted from physical entities such as storage, memory, and cores. Their main goal is facilitating the programming task, increasing programmer productivity, achieving scalability, and improving software portability. Exascale systems refers to highly parallel computing systems capable of at least one exaFLOPS. Therefore, their implementation represents a big research and technology challenge. The design and development of Exascale systems is currently under investigation with the goal of

* Corresponding author: talia@dimes.unical.it

building by 2020 high-performance computers composed of a very large number of multi-core processors expected to deliver a performance of 10^{18} operations per second. Programming paradigms traditionally used in HPC systems (e.g., MPI, OpenMP, OpenCL, Map-Reduce, and HPF) are not sufficient/appropriate for programming software designed to run on systems composed of a very large set of computing elements [1]. To reach Exascale size, it is required to define new programming models and languages that combine abstraction with both scalability and performance. Hybrid models (shared/distributed memory) and communication mechanisms based on locality and grouping are currently investigated as promising approaches. Parallel applications running on Exascale systems will require to control millions of threads running on a very large set of cores. Such applications will need to avoid or limit synchronization, use less communication and remote memory, and handle with software and hardware faults that could occur. Nowadays, no available programming languages provide solutions to these issues, specially when data-intensive applications are targeted. In this scenario, the EU funded Horizon 2020 project *ASPIDE* is studying models for extreme data processing on Exascale systems, starting from the idea that parallel programming paradigms must be conceived in a data-driven style especially for supporting for Big Data analysis on HPC systems.

This paper introduces the main features and the programming constructs of the *DCEx* programming model designed in the *ASPIDE* project. *DCEx* is based upon data-aware basic operations for data-intensive applications supporting the scalable use of a massive number of processing elements. The *DCEx* model uses private data structures and limits the amount of shared data among parallel threads. The basic idea of *DCEx* is structuring programs into data-parallel blocks, which are the units of shared- and distributed-memory parallel computation, communication, and migration in the memory/storage hierarchy. Computation threads execute close to data, using near-data synchronization based on the *Partitioned Global Address Space* (PGAS) model, which assumes the memory is partitioned into a global shared address space and a portion that is local to each process [3]. In the *DCEx* model, three main types of parallelism are exploited: data parallelism, task parallelism, and *Single Program Multiple Data* (SPMD) parallelism. A prototype API based on that model will be implemented.

The rest of the paper is structured as follows. Section 2 presents principles and features of the data model used in *DCEx*. The data block concept is presented and data access and management operations are discussed. Section 3 introduces the principles and the kinds of parallelism exploited in *DCEx*. Section 4 presents a use case designed using the programming mechanisms of *DCEx*. Finally, Section 5 outlines related parallel models and languages recently proposed for scalable applications on Exascale systems.

2 The DCEx Data Model

The role of data management and processing is central in the *DCEx* programming model. The data model used in the *DCEx* is based on the *data parallel block*

(DPB) abstraction. DPBs are the units of shared- and distributed-memory parallel computation, communication, and migration. Blocks and their message queues are mapped onto processes and placed in memory/storage by the ASPIDE runtime. Decomposing a problem in terms of block-parallelism (instead of process-parallelism) enables migrating blocks during the program execution between different locations in the hardware. This is the main idea that lets us integrate in- and out-of-core programming in the same model and change modes without modifying the source code.

A DPB is used for managing a data element in the main memory of one or multiple computing nodes. In particular, a DPB d can be composed of multiple partitions:

```
d = [ part 0 ][ part 1 ][ part 2 ][ part 3 ]...[ part n-1 ]
```

where each partition is assigned to a specific computing node.

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

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 by the following syntax:

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

where:

- `d`: is the DPB created to manage in main memory the data element read from secondary storage;
- `source`: specifies the location of data in secondary storage (e.g., an URL);
- `format`: is an optional parameter specifying the format of data;
- `part|repl`: is an optional parameter, which should be specified only if the optional *Carea* directive is included (see below). If *part* is used, d must be partitioned across all the computing nodes in *Carea*. If *repl* is used, d must be replicated in all the computing nodes of the *Carea*;
- the ellipsis indicate further parameters to be defined;
- `Cnode|Carea`: is an optional directive to specify how d should be mapped on a single computing node or on an area of computing nodes. In particular, if a *Cnode* is specified, d is loaded in the main memory of that specific computing node; if a *Carea* is specified, d is partitioned (if the *part* flag is used) or replicated (if the *repl* flag is used) in the main memory of the computing nodes included in that area.

In addition to *data.get*, it is also possible to use the *data.declare* operation, which declares a DPB that will come into existence in the future, as a result of a task execution. Here is an example of DPB declaration:

```
d = data.declare();
```

The use of *data.declare* in association with task operations allows to store the output of a task.

A DPB can be written in secondary storage using the *data.set* operation, which is defined as follows:

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

where:

- `d`: is the DPB to be stored in secondary storage;
- `dest`: specifies where of data must be written in secondary storage (e.g., an URL);
- `format`: is an optional parameter specifying the format of data.

3 The DCEx Parallelism Model

In the *DCEx* model, data are the fundamental artifact and they are processed in parallel. In particular, *DCEx* exploits three main types of parallelism for managing the data parallel blocks: data parallelism, data-driven task parallelism, and SPMD parallelism.

To simplify the development of applications in heterogeneous distributed memory environments, large-scale data- and task-parallelism techniques can be developed on top of the data-parallel block abstractions divided into partitions. Different partitions are placed on different cores/nodes where tasks will work in parallel on data partitions. This approach allows computing nodes to process in parallel the data partitions at each core/node using a set of statements/library calls that hide the complexity of the underlying operations. Data dependency in this scenario limits scalability, thus it should be avoided or limited to a local scale.

Some proposals for Exascale programming are based on the adaptation of traditional parallel programming languages and on hybrid solutions. This incremental approach is conservative and often results in very complex codes that may limit the scalability of programs on many thousands or millions of cores. Approaches based on a partitioned global address space (PGAS) memory model appear to be more suited to meeting the Exascale challenge [5]. PGAS is a parallel programming model that assumes a global memory address space that is logically partitioned. A portion of the address space is local to each task, thread, or processing element. In PGAS the partitions of the shared memory space can have an affinity for a particular task, in this way data locality is implemented. For these reasons PGAS approaches have been analyzed and adopted in the *DCEx* model for partitioning of the address space using locality to limit data access overhead.

3.1 Basic Features

As mentioned before, the *DCEx* model for managing a very large amount of parallelism exploits three main types of parallelism: Data parallelism, task parallelism, and SPMD parallelism. Those forms of parallelism are integrated with PGAS features taking into account computing areas and other data and computing locality features.

Data parallelism is achieved when the same code is executed in parallel on different data blocks. In exploiting data parallelism, no communication is

needed, therefore this type of parallelism allows for the independent execution of code processing in parallel different partitions of data without suffering of communication or synchronization overhead.

Task parallelism is exploited when different tasks that compose an application run in parallel. The task parallelism in *DCEx* 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. Such parallelism can be defined in two manners: *i*) explicit, when a programmer defines dependencies among tasks through explicit instructions; *ii*) implicit, when the system analyses the input/output of tasks to understand dependencies among them.

SPMD parallelism is achieved when a set of tasks execute in parallel the same code on different partitions of a data set (in our case parallel data blocks); however, differently from data parallelism, processes cooperate to exchange partial results during execution. Communication occurs among the processors when data must be exchanged between tasks that compose an SPMD computation. Tasks may execute different statements by taking different branches in the program and it is occasionally necessary for processors to synchronize, however processors do not have to operate in locksteps as in SIMD computations.

In *DCEx*, these three basic forms of parallelism can be combined to express complex parallel applications. This can be done by programming Exascale applications in a *Directed Acyclic Graph* (DAG) style that corresponds to workflow programming, where a parallel program is designed as a graph of tasks. As data-intensive scientific computing systems become more widespread, it is necessary to simplify the development, deployment, and execution of complex data analysis applications. The workflow model is a general and widely used approach for designing and executing data-intensive applications over high performance systems or distributed computing infrastructures. Data-intensive workflows consist of interdependent data processing tasks, often connected in a DAG style, which communicate through intermediate storage abstractions. This paradigm in the *DCEx* model can be exploited to program applications on massively parallel systems like Exascale platforms.

The combination of the three basic types of parallelism allows developers to express other parallel execution mechanisms such as pipeline parallelism, which is obtained when data is processed in parallel at different stages. Pipeline parallelism is in particular appropriate for processing data streams as their stages manage the flow of data in parallel [6]. As mentioned before, the types of parallelism discussed here are combined in *DCEx* with the features of the PGAS model that support the definition of several execution contexts based on separate address spaces. For any given task, this allows for the exploitation of memory affinity and data locality that provides programmers with a clear way to distinguish between private and shared memory blocks, and determine the association to processing nodes of shared data locations [7]. In fact, in the PGAS model, the computing nodes have an attached local memory space and portions of this local storage can be declared private by the programming model, making them not visible to other nodes. A portion of each node's storage can be also shared with

others nodes. Each shared memory location has an affinity, which is a computing node on which the location is local, with the effect that data access rate is higher for code running on that node. Therefore, through data affinity mechanisms a programmer can implement parallel applications taking into account local data access and communication overhead facilitating high performance and scalability.

3.2 Programming Constructs

This section introduces the main programming concepts and constructs designed in the *DCEx* model. The description is focused on the two main components of the model: *i*) computing nodes and areas that identify single processing elements or regions of processors of an Exascale machine where to store data and run tasks; *ii*) tasks and task pools that represent the units of parallelism.

Computing Nodes and Computing Areas The *DCEx* model defines two basic constructs to refer to computing nodes and computing areas:

- `Cnode` representing a single computing node, and
- `Carea` representing a region (or area) including a set of computing nodes.

In general, `Cnodes` and `Careas` are used to implement data and task locality by specifying a mapping between data loading operations and (the main memory of) computing nodes, and task execution operations and (the processors of) computing nodes.

A `Cnode` variable may be used to specify in a data loading operation the computing node that should be used to store (in its main memory) a given data element read from secondary storage. It can be used also in a task execution operation to specify the computing node on which a task should be executed.

A `Carea` variable may be used to specify in a data loading operation the set of computing nodes that should be used to store (in their main memory) a given data element read from secondary storage, by partitioning data on all the nodes. In a task execution operation a `Carea` is used to specify the computing nodes on which a pool of tasks should be executed. A `Cnode` is declared as follows:

```
node = Cnode;
```

where `node` is a variable used to refer to the computing node.
Through this declaration, the runtime chooses which computing node will be assigned to variable `node`. Alternatively, it may be specified by annotations to help the runtime in choosing the computing node, e.g.:

```
node = Cnode({hardware annotation parameters})
```

A `Carea` can be defined as an array of computing nodes. For instance, the example below defines nodes as an array of 1000 computing nodes:

```
nodes1 = Carea(1000);
```

Similarly, the following examples defines a two-dimensional array of 100x100 computing nodes:

```
nodes2 = Carea(100,100);
```

Referring to the last example, the following notation:

```
nodes2[10][50]
```

identifies the computing node at row 10 and column 50 in the nodes `Carea`. It is also possible to create a `Carea` as a view of a larger `Carea`:

```
nodes3 = Carea(nodes2,10,10);
```

which extracts a 10x10 matrix of computing nodes from from the `Carea` defined by `nodes2`.

Tasks and Task Pools In *DCEx*, tasks are the basic elements for implementing concurrent activities. To manage the parallel execution of multiple tasks, a task data dependency graph is generated at runtime. Each time a new task is executed, its data dependencies are matched against those of the other tasks. If a data dependency is found, the task becomes a successor of the corresponding tasks. Tasks are scheduled for execution as soon as all their predecessors in the graph have produced data they need as input. The programming model allows to express parallelism using two concepts: *task* and *task_pool*.

A task can be defined according to the following syntax:

```
t = Task(f_name,f_param_1,...,f_param_n) [at Cnode|Carea] [on failure ignore
    |retry|...];
```

where:

- `t`: is a numeric identifier to the task being created;
- `f_name`: is the name of the function to be executed;
- `f_param_i`: the i-th parameter required by the function identified by *f_name*;
- `at Cnode|Carea`: is an optional directive that allows to specify on which given computing node the *task* should be executed (if a *Cnode* is specified), or to execute the task on any computing node from a set of computing nodes (if a *Carea* is specified).
- `on failure`: is an optional directive that allows to specify the action (for instance, *ignore* or *retry* with it) to be performed in case of task failure.

According to the basic assumptions about concurrent task execution mentioned above, the *Task* keyword allows to concurrently execute a method in the future, as soon as its data dependencies are resolved (i.e., its input data are ready to be used). Moreover, the *at* directive that specifies the execution of a task on a given *Cnode* is intended as request/suggestion to runtime that can be satisfied or not, depending on available hardware resources, their status and load, and the runtime execution optimization strategy.

As an example, let assume we defined the following function:

```
partitioner(in:dataset, out:trainset, out:testset);
```

that takes as input a *dataset* and returns (by reference) a *trainset* and a *testset* extracted from the dataset. The following code shows how that function may be executed:

```
dsURL = ``some url''; trainURL = ``some url''; testURL ``some url'';
node = Cnode; ds = data.get(dsURL) at node;
train = data.declare(); test = data.declare();
```

```
t = Task(partitioner, ds, train, test);
data.set(train, trainURL); data.set(test, testURL);
```

Tasks can be used in a *for* loop to exploit data-driven task parallelism. For example, a set of tasks can be executed in parallel in such a way:

```
N = 10; vec = [];
for (i=0; i<N; i++) {
  if (cond)
    vec[i] = Task(f1, f1_par_1, ..., f1_par_n);
  else
    vec[i] = Task(f2, f2_par_1, ..., f2_par_n);
}
```

In this code example we assume functions $f1$ and $f2$ have been already defined.

To implement SPMD parallelism in *DCEx*, the *Task_Pool* abstraction is defined to represent a set of tasks. In fact, tasks in a pool are activated to execute the same function that implements the algorithm executed by the *Task_Pool* in an SPMD parallel style. The basic syntax for declaring a pool of tasks is as follows:

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

where:

- `tp`: is an identifier of the task pool being defined, it can be also used with an index to identify a single task of the pool; and
- `size`: is an optional parameter specifying the number of tasks in the pool.

The statement above declares a task pool but does not spawn its execution. Each task in the pool must be activated explicitly using a *for* loop as in the following example:

```
N = 10; nodes = Carea(N);
for (i=0; i<N; i++) {
  f_param_1 = ...; f_param_n = ...;
  tp[i] = Task(f_name, f_param_1, ..., f_param_n) at nodes[i];
}
```

If there are no dependencies among the tasks initialized in the loop, they execute concurrently. On the other hand, if a task works on some data that is not yet available, it waits until that data becomes available, according to the execution model outlined before. On a *Task_Pool tp* some operations such as the following listed here can be defined:

- `size(tp)` to access the number of tasks in a pool.
- `structure(tp)` to know how the tasks in a pool are structured (e.g., in a vector, a two-dimensional matrix, a tree).
- `zone(tp)` to know in which *Carea* the tasks of a pool are mapped.

4 Use Case

To show through a real data-intensive application how the *DCEx* constructs can be used, in the following is described a trajectory data analysis application coded in DCEx. The workflow shown in Figure 1 represents the main steps of the applications (some of them are optional):

A. *Crawling*: multiple crawlers are instantiated and run in parallel for gathering data from social media. If data have already been downloaded and stored in files, a specific crawler (FileCrawler) is used to load the data.
B. *Filtering*: filtering functions are run in parallel to verify if social media items meet or not some conditions.
C. *Automatic keywords extraction and data grouping*: the keywords that identify the places of interests are extracted; these keywords will be used to group social media items according to the places they refer to.
D. *RoIs extraction*: a data parallel clustering algorithm is used to extract Regions-of-Interest (RoIs) from social media data grouped by keywords [2]. RoIs represent a way to partition the space into meaningful areas; they are the boundaries of points-of-interest (e.g., square of a city).
E. *Trajectory mining*: This step is executed to discover behaviour and mobility patterns of people by analyzing geotagged social media items. Highly parallel versions of the FP-Growth (frequent itemset analysis) and Prefix-Span (sequential pattern mining) algorithms are used here.

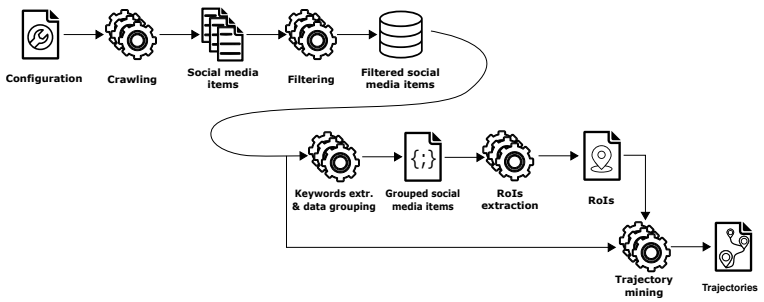

Fig. 1. Workflow of the urban computing use-case.

Listing 1.1 shows the *DCEx* pseudo-code for the trajectory data analysis use case introduced above. Initially the dataset "FullFlickrData.json" is loaded (line 2), a *Carea* of 16,000 nodes is defined (lines 3-4), then the dataset is split into 16,000 partitions and mapped onto the computing nodes (line 5). After that, filtering tasks are executed in parallel on the partitions to filter out Flickr posts that are not geotagged or do not refer to the city of Rome (lines 6-12). Filtering data is processed in parallel to extract keywords in each cell (lines 13-19). Then, such keywords are aggregated to find the top keywords in the area (lines 20-30). Afterward, filtering data have been used again and aggregated based on top keywords (lines 31-39). Finally, the RoI extraction (lines 40-48) and trajectory mining tasks (49-51) are executed concurrently.

Listing 1.1. DCEx code for the urban computing use-case.

```
1 //Crawling
2 source="/home/UNICAL/FullFlickrData.json";
3 numNodes = 16000;
4 nodes = Carea(numNodes);
5 dd = data.get(source, FILE, part) at nodes;
6 //Filtering
```

```
filterTasks = Task_Pool(nodes.size);
ddfilt = [];  f_param_0 = "IsGeotagged"; f_param_1 = "IsInRome";
for(i=0; i<nodes.size; i++){
  ddfilt[i] = data.declare();
  filterTasks[i] = Task(filteringFunc, dd[i], ddfilt[i], f_param_0, f_param_1) at nodes[i];
}
//Keywords extraction
keywordsInCellTasks = Task_Pool(nodes.size);
keywordsInCellParts = []; cell_width = "500m";
for(i=0; i<nodes.size; i++){
  keywordsInCellParts[i] = data.declare();
  keywordsInCellTasks[i] = Task(findKeywordsInCell, ddfilt[i], keywordsInCellParts[i],
       cell_width) at nodes[i];
}
keywordsInCell = groupByKey(keywordsInCellParts);
numCells = keywordsInCell.size;
topKeywordsInCellTasks = Task_Pool(numCells);
nodes = Carea(numCells);
topKeywordsInCell = []; numTopKeywords = 5;
for(j=0; j<numCells; j++){
  topKeywordsInCell[j] = data.declare();
  topKeywordsInCellTasks[j] = Task(findTopKeywords, keywordsInCell[j], topKeywordsInCell[j],
       numTopKeywords) at nodes[j];
}
topKeywords = data.declare();
aggregateKeysTask = Task(aggregateKeywords, topKeywordsInCell, topKeywords);
//Data grouping
splitDataPerKeywordsTasks = Task_Pool(numNodes);
nodes = Carea(numNodes);
dataPerKeywordsParts=[];
for(i=0; i<numNodes; i++){
  dataPerKeywordsParts[i] = data.declare();
  splitDataPerKeywordsTasks[i] = Task(assignDataToKeywords, ddfilt[i], dataPerKeywordsParts[i
       ], topKeywords) at nodes[i]);
}
dataPerKeywords = groupByKey(dataPerKeywordsParts);
//RoIs extraction
numRoIs = dataPerKeywords.size;
roiTasks = Task_Pool(numRoIs);
nodes = Carea(numRoIs);
rois=[]; eps = 50; minPts=150; splits = 32;
for(k=0; k<numRoIs; k++){
  rois[k] = data.declare();
  roiTasks[k] = Task(findRoI, dataPerKeywords[k], rois[k], eps, minPts, splits) at nodes[i];
}
//Trajectory mining
trajectories = data.declare();
trajectoryTask = Task(trajectoryMining, ddfilt, trajectories, rois);
```

5 Related Work

This section discusses a few parallel programming models and languages that have been proposed for the implementation of scalable applications on Exascale machines [9]. The approach and the main features of those models and languages are briefly discussed. To manage programming issues of data-intensive applications, different scalable programming models have been proposed [4]. Several parallel programming models, languages and libraries are currently under development for providing high-level programming interfaces and tools for implementing high-performance applications on future Exascale computers. Here we introduce the most significant proposals and outline their main features.

The programming models for Exascale systems can be classified according to four categories: distributed memory, shared memory, partitioned memory, and hybrid models. Since Exascale systems can be composed of millions of processing nodes using large distributed memory, message passing programming systems, such as MPI, are candidate tools for programming applications for such class of systems. However, traditional MPI all-to-all communication does not scale well in Exascale environments. Hence to solve this issue new MPI releases (like MPI+X) have been proposed to support neighbor collectives for providing sparse "all-to-some" communication patterns that limit the data exchange on limited regions of processors [5]. Other distributed-memory languages for Exascale are Legion[8] and Charm++[9]. On the other side, the shared-memory paradigm offers a simple parallel programming model although it does not provide mechanisms to explicitly map and control data distribution and it includes non-scalable synchronization operations that are making very challenging its implementation on massively parallel systems.

As a trade-off between distributed and shared memory organizations, PGAS model [8] has been designed for implementing a global memory address space that is logically partitioned and portions of it are local to single processes. The main goal of the PGAS model is to limit data exchange and isolate failures in very large-scale systems. DASH[10] offers distributed data structures and parallel standard template library algorithms via a PGAS approach. A variant of the PGAS model, *Asynchronous PGAS (APGAS)* [7] that has been adopted by some programming languages, such as X10[11] and Chapel[12], supports both local and remote asynchronous task creation. Differently for the PGAS model, the APGAS model does not require that all processes run on similar hardware and supports dynamically spawning of multiple tasks. In fact, multiple threads be active simultaneously in a *place*, using either local or remote data. In addition, it does not require that all the places in a computation must be homogeneous [10]. PGAS-based languages proposed recently are X10, Chapel and UPC[13]. They share some concepts with *DCEx*, although they are not specifically designed for data-centric applications. In fact, in exploiting the PGAS approach, *DCEx* integrates PGAS with local communication mechanisms and data parallel blocks.

6 Conclusions

Traditional parallel programming paradigms are not appropriate for programming scalable software designed to run on systems composed of a very large set of computing nodes. Therefore, to reach Exascale size it is required to define new programming models, languages and APIs that combine abstraction with

[8] https://legion.stanford.edu/
[9] https://charmplusplus.org/
[10] https://www.dash-project.org/
[11] https://x10-lang.org/
[12] https://chapel-lang.org/
[13] https://upc-lang.org/

scalability and performance. Hybrid models (shared/distributed memory) and locality-based communication mechanisms are currently investigated as promising approaches. The main goal of the ASPIDE project is the design and development of a new Exascale programming model for extreme data applications. The designed *DCEx* programming model includes data parallel blocks and data-driven parallelism for the implementation of scalable algorithms and applications on top of Exascale computing systems with a special emphasis on the support of massive data analysis applications. We presented here the language features and a use case. The implementation of the *DCEx* language is an ongoing activity.

Acknowledgments

This work has been partially funded by the ASPIDE Project funded by the European Union's Horizon 2020 Research and Innovation Programme under grant agreement No 801091.

References

1. Belcastro, L., Marozzo, F., Talia, D.: Programming models and systems for big data analysis. International Journal of Parallel, Emergent and Distributed Systems 0(0), 1–21 (2018)
2. Belcastro, L., Marozzo, F., Talia, D., Trunfio, P.: G-roi: Automatic region-of-interest detection driven by geotagged social media data. ACM Transactions on Knowledge Discovery from Data 12(3), 27:1–27:22 (January 2018)
3. Culler, D.E., Dusseau, A., Goldstein, S.C., Krishnamurthy, A., Lumetta, S., von Eicken, T., Yelick, K.: Parallel programming in split-c. In: Proceedings of the 1993 ACM/IEEE Conference on Supercomputing. pp. 262–273 (Nov 1993)
4. Diaz, J., Munoz-Caro, C., Nino, A.: A survey of parallel programming models and tools in the multi and many-core era. IEEE Transactions on parallel and distributed systems 23(8), 1369–1386 (2012)
5. Gropp, W., Snir, M.: Programming for exascale computers. Computing in Science & Engineering 15(6), 27–35 (2013)
6. del Rio Astorga, D., Dolz, M.F., Fernández, J., García, J.D.: A generic parallel pattern interface for stream and data processing. Concurrency and Computation: Practice and Experience 29(24) (2017)
7. Saraswat, V., Almasi, G., Bikshandi, G., Cascaval, C., Cunningham, D., Grove, D., Kodali, S., Peshansky, I., Tardieu, O.: The asynchronous partitioned global address space model. In: The 1st Workshop on Advances in Message Passing. pp. 1–8 (2010)
8. Stitt, T.: An introduction to the partitioned global address space programming model. CNX. org (2010)
9. Talia, D.: A view of programming scalable data analysis: from clouds to exascale. Journal of Cloud Computing 8(1), 4–20 (2019)
10. Tardieu, O., Herta, B., Cunningham, D., Grove, D., Kambadur, P., Saraswat, V., Shinnar, A., Takeuchi, M., Vaziri, M.: X10 and apgas at petascale. In: ACM SIGPLAN Notices. vol. 49, pp. 53–66. ACM (2014)