Project no. 801091

ASPIDE

Research & Innovation Action (RIA)
EXASCALE PROGRAMING MODELS FOR EXTREME DATA PROCESSING

Run-time support for programming extreme data application D2.4

Due date of deliverable: 14th August 2020

Start date of project: June 15^{th}, 2018

Type: Deliverable
WP number: WP2

Responsible institution: University of Calabria
Editor and editor's address: Domenico Talia, University of Calabria

Version 1.7

Project co-funded by the European Commission within the Horizon 2020 Programme		
Dissemination Level		
PU	Public	√
PP	Restricted to other programme participants (including the Commission Services)	
RE	Restricted to a group specified by the consortium (including the Commission Services)	
CO	Confidential, only for members of the consortium (including the Commission Services)	

Revision History

Rev.	Date	Author(s)	Description
1.0	17/03/20	Javier Garcia-Blas (UC3M)	Initial Version.
1.1	28/03/20	Fabrizio Marozzo (UNICAL), Domenico Talia (UNICAL), Salvatore Giampá (UNICAL)	Added integration chapter.
1.2	06/04/20	David del Rio (UC3M) , Javier Fernandez Muñoz (UC3M)	Added runtime chapter.
1.3	23/06/20	Javier Garcia-Blas (UC3M)	Restructured document.
1.4	05/08/20	Javier Garcia-Blas (UC3M)	Added conclusions.
1.5	10/08/20	Javier Garcia-Blas (UC3M)	Added figure in executive summary.
1.6	14/08/20	Fabrizio Marozzo (UNICAL)	Added extended conclusions.
1.7	14/08/20	Fabrizio Marozzo (UNICAL)	Added new code versions and figures in Chapter 2.

Main Contributors

David del Rio (UC3M)
Domenico Talia (UNICAL)
Fabrizio Marozzo (UNICAL)
Javier Fernandez Muñoz (UC3M)
Javier Garcia-Blas (UC3M)
Salvatore Giampá (UNICAL)

Executive Summary

The DCEx programming model designed in WP2 must be supported by 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. A major activity will consist of the construction and management of the internal run-time representation of the application as dynamic patterns including computational components and data dependencies between components. The execution is based on data-driven, dynamic scheduling mechanisms.

This deliverable describes the main concepts of the integration of the DCEx programming model into our parallel pattern programming approach. In particular, the main entities of DCEx will be shown in a more detailed way and a new DCEx execution model will be proposed.

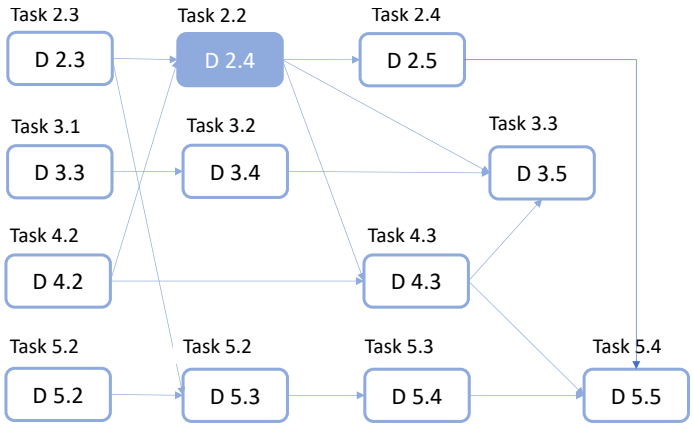


Figure 1: Deliverable dependencies of D2.4.

Contents

Chapter 1

Introduction

As data intensive scientific computing systems become more widespread, there is a necessity of simplifying the development, deployment, and execution of complex data analysis applications for scientific discovery [1]. The scientific workflow model is the leading approach for designing and executing data-intensive applications in high-performance computing infrastructures.

Current scheduling schemes are data locality agnostic. In this mode, tasks are first placed in the work queue of the node where they are created. If no work stealing events are triggered, each task remains there until a worker attached to the server obtains the task. If another server steals the task, it can be executed on any worker in the system. Thus, this scheme is not predictable, but it allows total flexibility to the load balancer. However, the application of this scheme in a data-intensive use case would result in a great increase of network traffic since locality is not applied.

As depicted in Deliverable D2.1, programming paradigms traditionally used in high performance computing 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 nodes [2].

The DCEx programming model designed in WP2 must be supported by 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 [3]. A major activity consists in the construction and management of the internal run-time representation of the application as dynamic patterns including computational components and data dependencies between components. The execution is based on data-driven, dynamic scheduling mechanisms.

DCEx is a novel programming model for data-intensive applications. It includes data parallel blocks and data-driven parallelism for the implementation of scalable

algorithms and applications. In a recent paper [4] we presented the main DCEx features and some use cases. The implementation of the DCEx framework is an ongoing activity.

GrPPI [5] is a generic and reusable parallel pattern interface for both stream processing and data-intensive C++ applications. It provides many high-level parallel patterns, such as Map, Reduce, Divide&Conquer, Pipeline and so on. GrPPI uses existing runtimes for obtaining the implementation of the parallel patterns. The Generic Parallel Pattern Interface (introduced in Deliverable D2.2) acts as an intermediate layer between the applications and different programming frameworks. Two types of parallel patterns are identified. First, data patterns are those in which the input data is available at the beginning of its execution and the number of data items is known in advance; Second, stream patterns are those in which the data size its not known and the data items comes from a given source at runtime. One of the goals of the ASPIDE project is to provide a task-based back-end that exploits data locality in data intensive applications. Actually, GrPPI implements its patterns on five different runtimes, that are exposed as execution models. In this deliverable we explain how the GRPPI model/framework has been extended for implementing the DCEx features.

ASPIDE aims to extend GrPPI to support a MPI+X model, adapted for distributed memory systems at large scale. In this project, we propose a new task-based back-end for distributed environments employing different communication channels (queues) for moving tasks and data among the different nodes comprising a given distributed architecture.

Figure 1.1 shows the DCEx programming framework and its components. In particular, the execution flow of a DCEx application, as implemented in the system prototype, is composed by the following steps:

1. The application is defined by the programmer by using a high level language (for example GrPPI) extended with the defined DCEx abstractions.
2. Internal representations of the application, tasks, and tasks graph are created by exploiting DCEx abstractions.
3. Tasks are sent to the Runtime for their execution.
4. The Runtime asks the Intelligent Data Manager (IDM) where is the data.
5. The Runtime asks the Scheduler to execute the tasks.
6. The IDM informs both the Scheduler and the Runtime where is the data.
7. The Monitor provides analysis to the Scheduler, so that intelligent decision can be done where to execute the task.
8. If this is not first task/application, information from the previous runs is used by the Auto-tuner to provide input parameters for the execution of future

applications, i.e., improving the application performance.

An application may consist of multiple types of tasks spanned across different *Careas*, which can share data through files and communication mechanisms. When an application is requested to be deployed, the Data Manager can be queried to find the nodes which hold (parts of) the input data. If needed, the Data Manager can be asked to replicate or distribute the data to a given number of nodes. Finally, the tasks dependent on this data can be started on these nodes. Tasks that are dependent on other task's output data will wait for this dependency to be met.

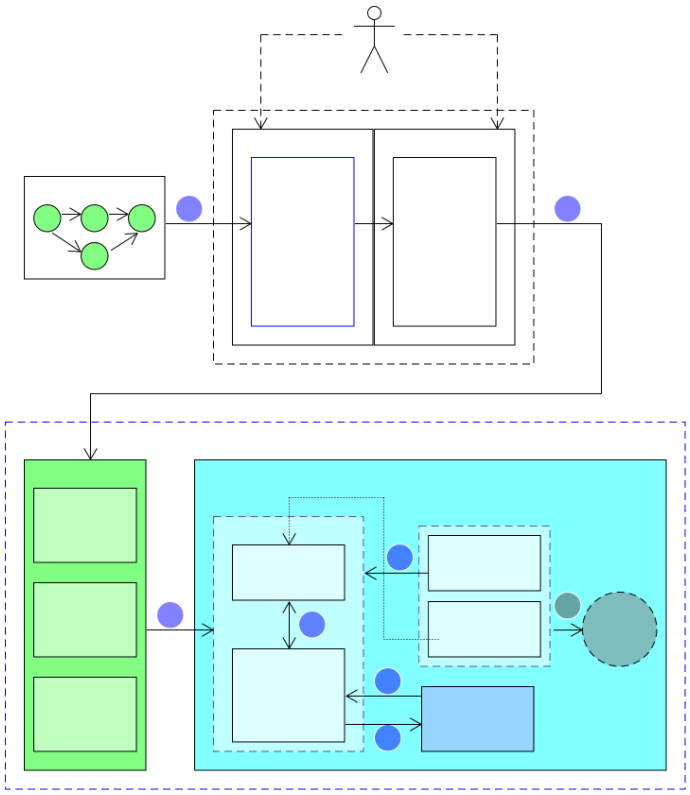


Figure 1.1: DCEx programming framework and its components.

This deliverable describes how the GrPPI runtime has been extended to support the execution of the programming constructs and mechanisms of DCEx and the functioning of the runtime system. Some code examples will show how a new GrPPI+DCEx appliction can be defined and executed. Moreover, the runtime support will be described together with the communication mechanisms for tasks scheduling and temporary data, the functioning of the scheduler and the managing of tasks. The approach is focuses from two points of view. First, we present a centralized scheduler based on multiple queues, which are specialized depending on the task nature (sequential or parallel). Second we detail a distributed and

decentralized scheduler that aims to run under large applications in HPC clusters.

Chapter 2

DCEx programming model integration in GrPPI

This chapter describes how the GrPPI language and it runtime have been extended to support the use and execution of the programming constructs and mechanisms introduced by the DCEx programming model. In the first section of this chapter, we introduce the main entities that interact with the runtime, i.e.,as $Master$ and $Workers$ entities, $CNode$ and $CArea$, $Data\ Parallel\ Block$ and $Partition$, $Task$ and $Program$. The second section deals with the currently defined execution models in GrPPI and the new execution model proposed to integrate GrPPI with DCEx. The DCEx runtime receives the code of a parallel application and generates the code that it will execute on the various workers by analyzing the inputs of the application. The third section show how to design a GrPPI+DCEx application that implements a data analysis applications.

The remainder of this chapter is organized as follows. Section 2.1 describes the main DCEx entities used by the runtime. Section 2.2 describes the behavior of the data-aware runtime and the interaction of tasks with data. Section 2.3 presents the pseudo code of a use case application.

2.1 DCEx entities

The main DCEx entities that interact with the runtime are shown in Figure 2.1: $Master$, $Worker$, $CNode$, $CArea$, $Data\ Parallel\ Block$ (DPB) and $Partition$, $Program$ and $Tasks$. The nodes that compose the system are divided in *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). There is only a master node for each application. 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 others. A `CArea` can refer to many `CNode` objects.

A *Task*, in DCEx, is the main buiding 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, process them, and write 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* (*DPB*) 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*, that is a group of computational nodes, identifying five nodes and two data parallel blocks (see Figure 2.2): "input" and "output".

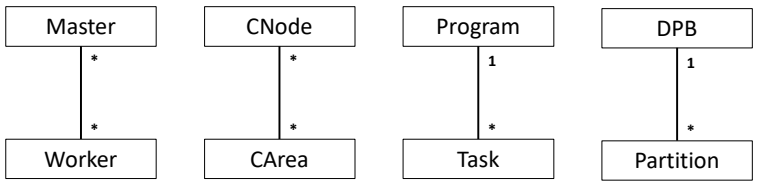


Figure 2.1: Main entities.

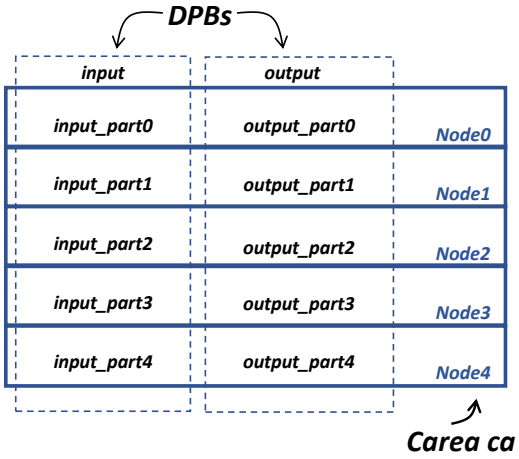


Figure 2.2: Data Parallel Block on two files (input and output) split on five nodes.

In particular, it has been needed to define in GrPPI the new classes `CNode`, `CArea`, and DPB. In this way the new runtime handles these abstract concepts used in DCEx applications and implements the needed mechanisms. In particular the DPB class includes methods like data.get, data.declare and data.set, explained in previous deliverables [4].

2.2 Runtime and tasks

Each pattern of GrPPI receives an execution model, that imposes an execution strategy for that pattern when it is invoked.

GrPPI [6] already exposes a set of execution models. Each currently exposed model, is implemented basing on specific runtimes:

- *sequential_execution*: implements each pattern of GrPPI in a simple sequential execution, so the used runtime is the same as the main program;
- *parallel_execution_native*: the runtime is given by POSIX threads, that offer a simple way to exploit the multicore capabilities of a single machine;
- *parallel_execution_omp*: uses the well known OpenMP [7] software technology.
- *parallel_execution_tbb*: implements the patterns by using the programming model and runtime of the homonymous software technology, Intel Threading Building Blocks (TBB).
- *parallel_execution_ff*: as the previous one, uses another programming model and runtime, given by Fast Flow [8], a distributed execution runtime based on parallel patterns.

The integration of DCEx into GrPPI required to bind the patterns to the new runtime. To achieve that goal, we defined a new execution model for GrPPI, represented by the DCEx_runtime class.

The DCEx runtime receives the code of a parallel application and generates the code that it will execute on the various $Workers$ by analysing the inputs that the application must execute.

When the execution of a GrPPI pattern is requested, for example a pipeline, the runtime queries (to know 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 node ($Node - 0$, ..., $Node - n$). Therefore, on $Node - 0$ the runtime will execute a function that analyses the $partition - 0$ of the input to generate the $partition - 0$ of the output, and so on.

```
1 class dcex_parallel_execution{
2 Carea ca;
3 istream in;
```

```
4 ostream out;
5
6 dcex_parallel_execution(Carea ca, istream in, ostream out){
7 this->ca = ca;
8 this->in = in;
9 this->out = out;
10 }
11
12 void pipeline(Generator && generate_op, Transformers && ...
     transform_ops){
13 Partition[] parts = in.getPartitions();
14 for(part in parts){
15 Cnode remoteNode = getBestNode(part.getCNode(), ca);
16 executeRemoteTask(remoteNode, generate_op(part), ... ,
     transform_ops[out.createPartition(remoteNode)]);
17 }
18 }
19 ...
20 }
```

Listing 2.1: DCEx code snippet from the dcex_parallel_execution class.

2.3 Code examples

In this section we show a GrPPI+DCEx application implementing a code section of the trajectory mining use case and show how the GrPPI+DCEx runtime manages data partitioning and distribution on the computing nodes.

We execute a pipeline pattern on an input DPB to generate an output DPB. In the following figure we show the GrPPI+DCEx application implementing some patterns (pipeline and reduce).

Listing 2.2 shows an example of a pipeline application. If GrPPI has not an internal description of the hardware nodes, we must include the method $Comp.loadConf$ that loads a configuration file listing the cluster nodes and how they are connected to each other (line 1). Then, the operation `Carea ca(5)` selects 5 nodes (line 2). After that we create two containers specifying in which `Carea` they will be stored (lines 3-4). Then we define a pipeline of tasks specifying that they will be executed in the `Carea` ca (lines 5-19). As you can note, we are proposing to explicitly include a `Carea` as a parameter of containers so the programmer can specify where the data will be stored and where the tasks will be executed.

```
1 Carea.loadConf(fileConf);
2
3 Carea ca(5);
4 grppi::Container_text in("folder/", ca);
5 grppi::Container_text out("result/", ca);
6
7 grppi::pipeline(grppi::dcex_parallel_execution{ca},
8 in,
```

```
grppi::farm([](auto x) -> json {
return json::parse(x);
}),
grppi::keep(isGeoragged), //GRPPI existing parallel pattern
grppi::keep(hasTags), //GRPPI existing parallel pattern
grppi::farm([](json x) -> pair<Cell, unordered_map<string,int>>{
   // Assuming that both Cell and unordered_map<string,int>
   classes have a tostring function implementation.
static const int cellSize = 200;
return make_pair(getCell(x, cellSize), getTagFreq(x));
}),
out
);
```

Listing 2.2: Code of a pipeline application.

The next example (Listing 2.3) shows how can be created two `Careas`, $ca1$ and $ca2$. Then, two pipelines of tasks are created. The first pipeline executes a set of tasks in $ca1$. The second one executes a set of tasks in $ca2$. Finally, a new `Carea` $ca3$ is created, which is the union of $ca1$ and $ca2$. A set of reduction tasks is executed on $ca3$.

```
Carea.loadConfNodes(fileConf);

Carea ca1(5);
grppi::Container_text in1 ("folder1/", ca1);
grppi::Container_text out1 ("result1/", ca1);

Carea ca2(5);
grppi::Container_text in2 ("folder2/", ca2);
grppi::Container_text out2 ("result2/", ca2);

//first pipeline
grppi::pipeline(grppi::dcex_parallel_execution{ca1},
in1,
grppi::farm([](auto x) -> json {
return json::parse(x);
}),
...
out1
);

//second pipeline
grppi::pipeline(grppi::dcex_parallel_execution{ ca2},
in2,
grppi::farm([](auto x) -> json {
return json::parse(x);
}),
...
out2
);

Carea ca3 = ca1 + ca2;
grppi::Container_text out3 ("result3/", ca3);

//final reduce
grppi::reduce(grppi::dcex_parallel_execution{ca3},
```

```
out1+out2,
grppi::farm([](auto x) -> json {
return json::parse(x);
}),
...
out3
);
```

Listing 2.3: Creation of two `Careas`.

Figure 2.3 shows how data and tasks are distributed in a cluster composed by 10 computing nodes. The figure shows the nodes (rectangles in black) that compose the two Cares (*ca1* and *ca2*), the partitioned data (colored *Input-i* and *Result-i*), intermediate data (white boxes), the tasks (arrows) that are generated by the programming patterns (first and second pipeline, and final reduction).

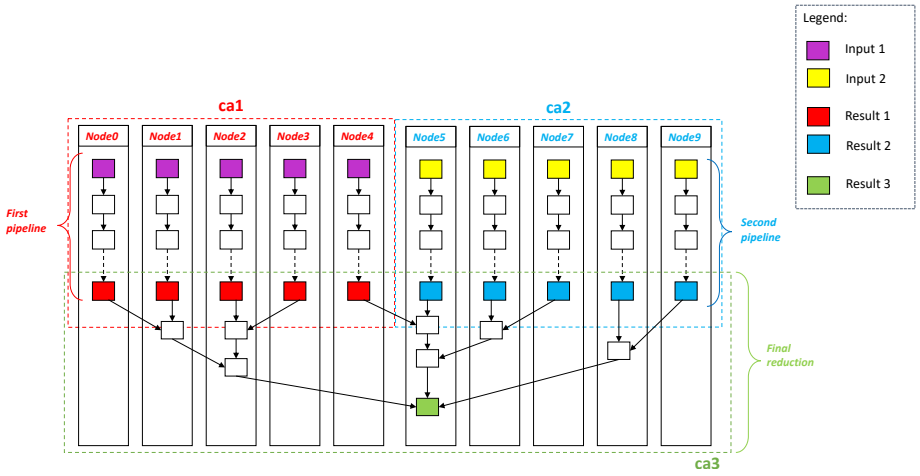


Figure 2.3: Data and tasks distributed in a cluster composed by 10 computing nodes.

Chapter 3

Runtime support

3.1 Communication mechanism for tasks scheduling and temporary data

In order to provide support to the scheduler for transferring both task and temporary data, we employ the ZeroMQ library [9], which provides an implementation of distributed queues based on sockets. In this sense, this library provide point-to-point, request-response and publisher-subscriber communication queues. Additionally, the request-response messages provide two different communication mechanisms: `i)` REQ/REP that ensures that messages are answered in the same order as the messages are requested, and `ii)` REQ/ROUTER that allows to send a response to a given client independently of the request's arrival order.

To better support both task and temporary data communication, we have preferred to use the REQ/ROUTER mechanism to communicate the working and storage nodes of the system. Basically, for each message, we store the client id and the request received from the destination node. Then, we can store multiple requests and answer them in any order, depending on the scheduling policy.

Using these queues, we have designed the two main components for temporary data and task communication.

3.2 Temporary data service

To communicate internal temporary data, which are not backed up by the data containers proposed in WP4, we have designed the temporary data service. This service is in charge of only 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 compute node will run a thread acting as the data server to support this data communication. This data server is composed by a remote-access indexed array that store the data serialized. Thus, data can be easily transferred to a different compute node if it is necessary. Basically, temporary data server behaves as a data container by providing the *data.get* and *data.set* functions defined by the DCEx model in Deliverable D2.2.

At the beginning of the application execution, a temporary data server will be deployed on each compute node and they will establish a communication channel with all the other processes involved on the execution.

We distinguish two different kind of tasks depending on their nature: sequential and parallel.

Sequential tasks are those that represents the execution of a non-pure function, i.e. the task store any kind of state that affects the execution of the function. These tasks, since they need to maintain the state among executions cannot be moved nor execute multiple instances on different data in parallel. Therefore, to be scheduled, a sequential task will be assigned beforehand to a given compute node, part of the *Carea*, and all its instances will be run in series in that node.

On the other hand, parallel tasks represents pure 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 nodes at the same time.

3.3 Task-based scheduler

A task represents a function that will be applied to a given input data to produce an output. Depending on the function definition, we distinguish two types of functions:

1. State-full functions: are those that maintains a state value among the different tasks executing the same function. Therefore, these functions cannot be executed on different nodes or in parallel.

2. Stateless functions: are those that its output only depends on the input data. Tasks executing these functions can be executed in parallel.

To execute the pending tasks in the system, we employ a pool of workers that constantly ask to the scheduler for a new task to run. The scheduler is an interchangeable service module that delivers pending tasks to the workers using task metadata and monitoring information for exploiting computing resources and data locality. Currently, we have designed two different scheduler: a centralized and a distributed version and shown in this chapter.

3.3.1 Initialization

In order to support the task communication among nodes in a given *Carea*, firstly it is necessary to assign communication ports for all necessary ZeroMQ queues. Compute nodes will act as the port servers. Port servers employ a user-defined port which is known by all the processes running the application. Then, each server process, that want to offer its services, must publish the listening port on the port server, while any process that want to connect to a server must get the listing port by means of the same port server.

Next, after deploying the port server, each node will deploy a local scheduler that will perform the following operations:

1. Each local scheduler establish its working *Carea* by receiving the list of IPs of the *Carea* nodes as part of the scheduler creation arguments.
2. Communications between local schedulers belonging to the same *Carea* will be established depending on the communication hierarchy used:
    - Centralized scheduler: All local schedulers will have a connection to a single centralized scheduler selected by default.
    - Distributed scheduler: All local schedulers will have a connection with any other local scheduler present in the same *Carea*.
3. Local schedulers on each compute node will deploy a set of working threads acting as a pool of execution entities. These execution entities are in charge of requesting pending task to the scheduler and execute those tasks assigned by the scheduler.

In the next sections, we describe in detail both proposed schedulers versions (centralized and distributed).

3.3.2 Centralized scheduler

The first version of the DCEx-GrPPI scheduler follows a centralized strategy. In this sense, a given node will act as the global scheduler and will receive the information about pending task and they will be dispatched to the nodes that are part of the *Carea*. To support the task transfers, we have designed the scheduler as a thread that answers the requests coming from the execution entities. This thread is also composed by three different task queues depending on the task-type. Figure 3.1 shows the composition of the centralized scheduler.

The centralized scheduler is composed by the following queues:

- Sequential task queue: This queue is comprised of a different queue for each node that store the ready tasks that should be executed in series.

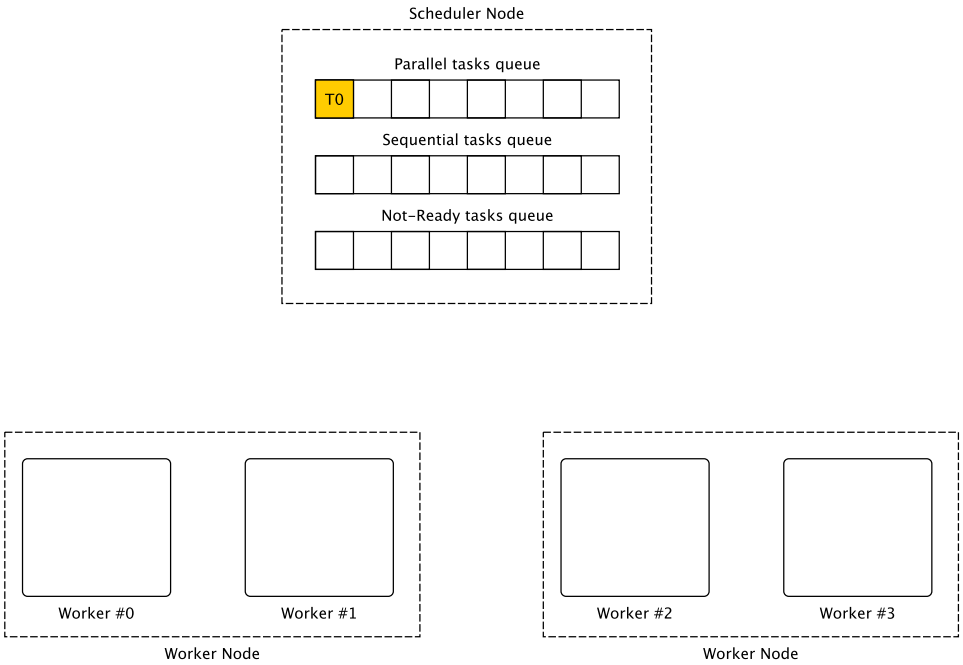


Figure 3.1: Centralized scheduler with each task queues.

- Parallel task queue: This queue is also comprised by a set of queues, one per node, and they store the tasks on the queue corresponding to the node that has local access to the associated data.
- Not-ready task queue: This queue is a list of tasks which have unresolved dependencies with any other task.

To provide support for the task transfers and scheduling, we have defined a set of messages which are sent via ZeroMQ. These messages will communicate to the scheduler with the worker threads and the main process.

- Run: This message is sent from the main process to the centralized scheduler to start launching the tasks of a given GrPPI pattern. This message will be acknowledged back as soon as all the task has finished their execution. This way, this message will act as a barrier for waiting until a pattern execution finish.
- GetTask: This message is sent from a worker thread to the centralized scheduler to request a new task.
- SetTask: This message communicate to the scheduler that it is required to create a new task to be scheduled.
- Consume: Communicate to the scheduler that a task has already finished. Additionally, when the scheduler receives this massage, 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.

Having defined the scheduler and its communication messages, in the following sections, we describe the task generation process and the task scheduling algorithm.

3.3.2.1 Task generation

To create all the tasks on the system, we first register the functions that should be executed and are classified into two groups:

- Sequential task: Those functions that are not defined as a parallel pattern are registered as sequential. Then, these functions are pre-assigned to a given compute node that will run all the task instances of that function in series.
- Parallel task: Those functions that are defined as parallel pattern are registered as parallel. Therefore, the scheduler may assign different task instances to different nodes and can be executed in parallel.

Additionally, in order to not collapse the system by creating an unlimited number of pending task at the same time, we also distinguish two kind of task:

- Generator tasks: Those tasks that can generate two or more new tasks after finishing their execution are considered generator tasks. Then the execution of these tasks will be stopped if the system reach the maximum number of pending task at a given point.
- Non generator task: Task that generates one or less new task are classified as non-generator. If a given task does not generate a new task when it finishes, it will be considered as a consumer task. Therefore, a consumer task will reduce the total number of pending tasks by one after finishing its execution.

3.3.2.2 Task scheduling algorithm

During the application execution, the worker threads will be asking to the scheduler for new work to execute. When the scheduler receives a request for a task, it will decide which task should be run on that worker thread following the next algorithm:

1. First, the scheduler assigns to the requesting thread a sequential task assigned to that node. Note that, if there is another thread of that node running an instance of the same task function, any other pending instance of that task will be stored as not-ready.
2. Next, if there are no sequential tasks pending for that node, the scheduler will look for a parallel task whose associated data is stored locally on the node running the requesting thread. To do so, we take advantage of the data

location information, specified by the DCEx model, which is defined by the list of nodes that store the data. Therefore, when a parallel task is created, it is enqueued on the parallel ready queue corresponding to the local node where its data is stored. If there are multiple nodes with local access to that data (e.g. data replication) the task is stored on all of them and linked together. So, when one of the replicas is consumed, all the copies are removed from all the queues. Therefore, when a node asks for a new task and there are tasks stored on its corresponding queue, the scheduler will assign to that thread the first task on that queue in order to ensure that the data for the task is stored locally on this node.

3. Otherwise, if there are no tasks ready to be executed whose data is locally stored on the requesting node, the scheduler will try to assign another task that have its data stored elsewhere and it will try to move the data from the remote node to the local one. If the data associated to a given task is temporary data (i.e. it is handled by the temporary data service), the data can be freely moved between the nodes belonging to the Carea. However, if the data is associated to a data container, the scheduler will try to check the Carea assigned to that data container. If both task and data container are part of the same Carea, the data will be moved between the remote and the local node. Otherwise, if they don't belong to the same Carea, the data will be only moved if the data container policy is defined as SOFT (see Deliverable D4.2).

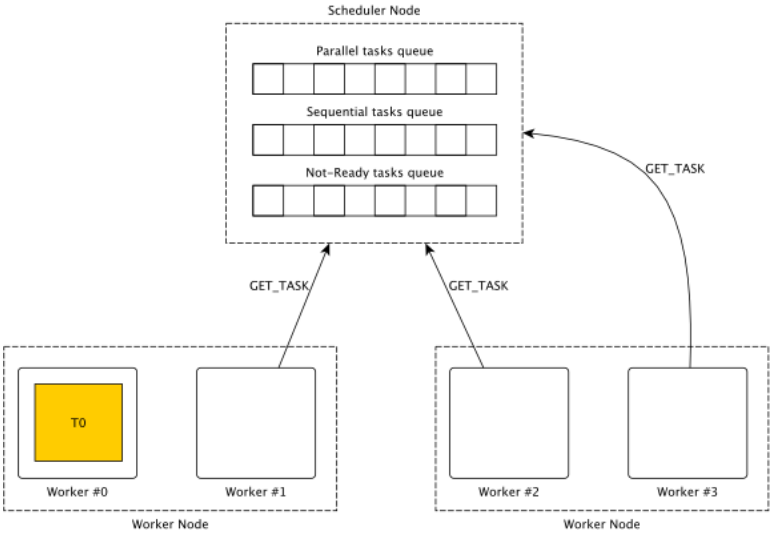
4. Finally, if no task can be assigned to the requesting thread, using the aforementioned criteria, the thread will wait until a new task is created. Basically, the scheduler will annotate that thread as waiting for a new task and it will try to assign that thread a task as soon as new tasks are available in the system.

Note that, if a task is a task generator, before assigning that kind of task, the scheduler checks the number of already existing task in the system. If there are as much task at the maximum number of pending tasks, those tasks are not assigned until there are space for the new tasks that will be generated after its execution.

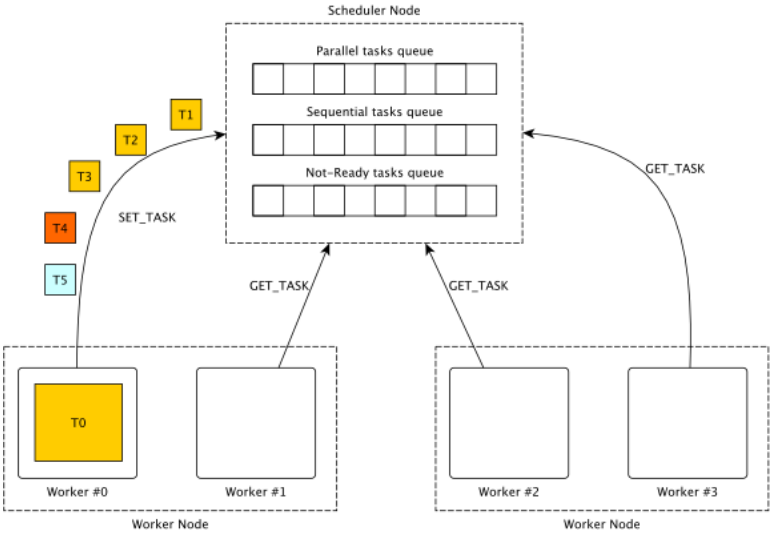
3.3.3 Distributed scheduler

Since a centralized approach is not the best option for scaling up the scheduler performance, we have designed a fully distributed version of the scheduler, shown in Figure 3.3. Similar to the centralized scheduler, each process will employ a thread as the local scheduler. Each local scheduler is comprised by the same queues as the centralized version, but, in this case each node stores a single task queue corresponding to the local node. Then, the worker threads request for new tasks to the local scheduler, and, if there are no task to run, local schedulers may communicate among them to transfer tasks among them.

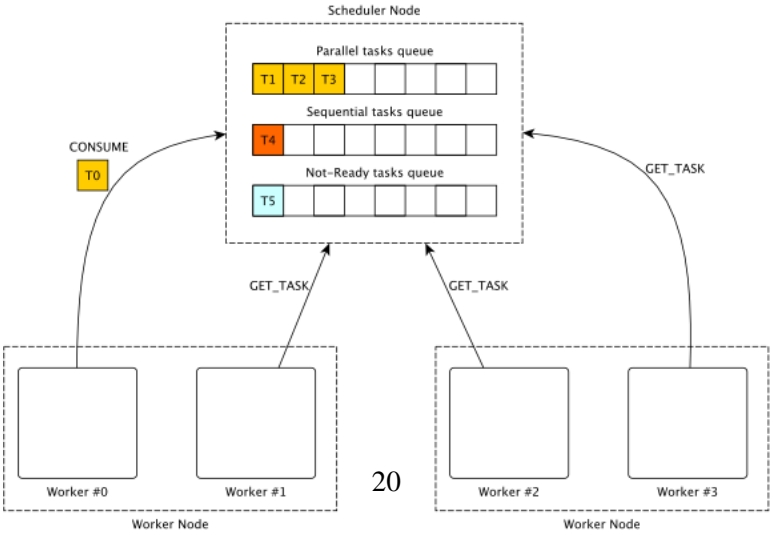
To get communication between the worker threads and the local scheduler, we have



(a) A. Task T0 is running in Worker 0.



(b) B. Task 0 creates Tasks 1 -5 and they are returned to the Scheduler Node.



(c) C. Tasks are assigned to the corresponding queue depending of the nature of the task.

Figure 3.2: Centralized schedule. Tasks assignment.

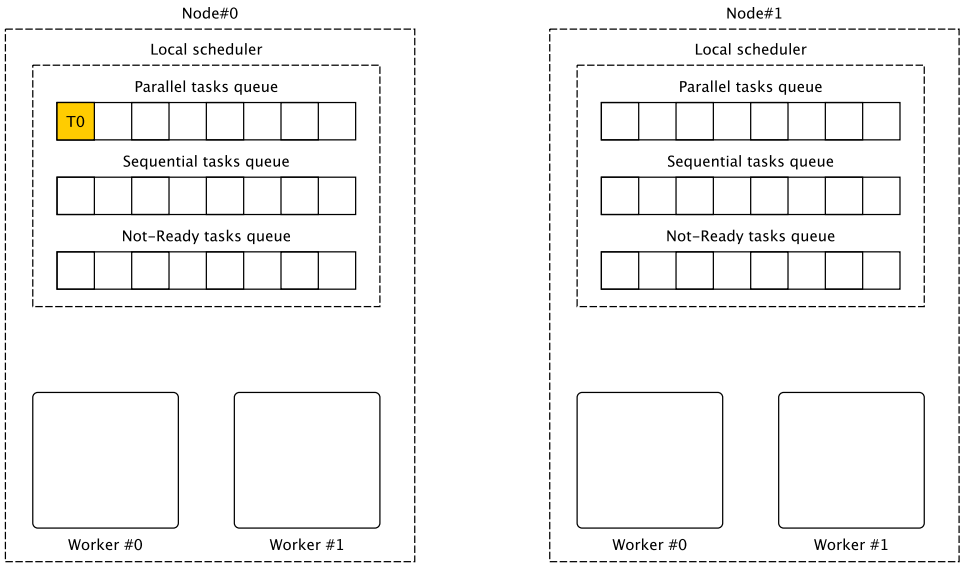


Figure 3.3: Distributed scheduler with each task queues.

defined the following messages:

- RUN: Start the scheduling process and send back an acknowledge as soon as the pending tasks have finished. This message is sent from all the processes to its corresponding local scheduler.
- GET_TASK:
  - Worker to local scheduler: Request a task to the local scheduler. After receiving this message the local scheduler will try to assign a local task to the requesting thread. If there are no ready task, the scheduler may ask to a remote scheduler for a task.
  - Local to remote scheduler: Steal a task from a remote scheduler. If a local scheduler has no task for a given thread, a scheduler can ask for a task to a remote scheduler, and, if possible, the remote scheduler will send a task to the local one.
- SET_TASK:
  - Worker to local scheduler: Requests the creation of a new task to the local scheduler. After creating the task, the scheduler may decide to delegate that task to a remote scheduler.
  - Local to remote scheduler: Delegates a new task to a remote scheduler.
- CONSUME:
  - Worker to local scheduler: Notifies to the local scheduler that a task

has finished. After receiving this message, the local scheduler will check if there are resolved dependencies on its local non-ready task. If a non-ready task has all its dependencies resolved, it is moved to the corresponding ready queue.

- NOTIFY_DEPENDENCY
  - Local to remote scheduler: After receiving a consume message, the local scheduler notifies the resolve dependencies to the scheduler holding a task with unresolved dependencies with the already finished task. Note that, each task contains the list of task that depends on its finalization and the node that stores the corresponding dependent task. Therefore, the local scheduler will send this message to all the nodes holding task with dependencies (Figure 3.5).

3.3.3.1 Task scheduling algorithm

In a similar way to the centralized scheduler, worker threads will ask to the scheduler for incoming tasks to be executed. The difference is that, in this case, each worker requests for a task to the local scheduler. Later, the local scheduler follows the next algorithm to decide which task should be executed. We follow the next principles:

- First, using the same algorithm of the centralized version, the local scheduler assigns a task stored locally (see Figure 3.4). In this way, if the task is stored locally, it means that data are local to that node, and therefore, we prioritize the access to local data. Basically, as in the centralized version, the local scheduler will check first for the sequential tasks stored locally and, then it will try with the parallel tasks.

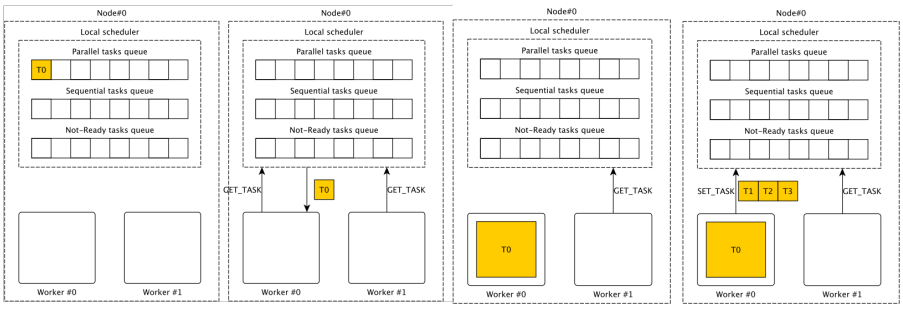


Figure 3.4: Distributed scheduler. Task T0 is initially allocated in the Parallel Task Queue. Later the task is assigned to Worker 0. Finally, after the finalization of Task T0, Tasks 1 to 3 are create and re-scheduled.

- If there are no local task that can be executed by that thread, the scheduler may ask for a new task to a remote scheduler (see Figure 3.5). To do so, we take advantage of the information provided by the monitoring system to

decide which node is the candidate to steal a task from it. This decision can be made by following different heuristics such us selecting the node with the higher CPU or memory usage. Then after asking for a new task to a remote scheduler, the remote scheduler will try to response the local with a new task following a similar criteria to that used by the centralize version for moving data. Basically, the remote scheduler will look for a task whose associated data can be moved to the local node. If its internal temporary data, the task and the data can be freely moved. On the contrary, if the data belongs to a data container, it can be only moved if both container and the scheduler configuration share the same *Carea*, or if the container configuration is set to SOFT. If none of these options can be considered, the remote node will answer to the local one with a no task message and the worker thread will wait until a new task becomes available.

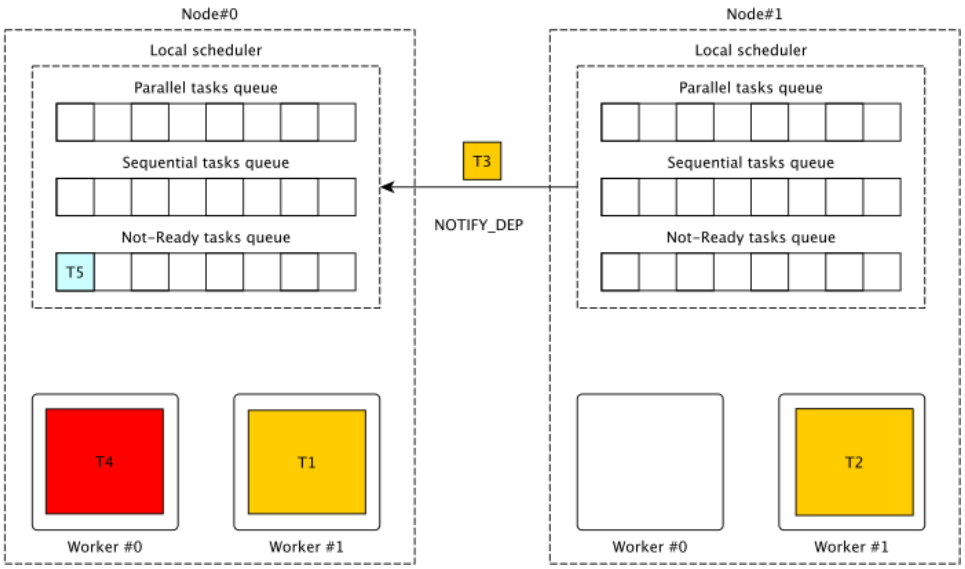


Figure 3.5: Distributed scheduler. Notify message.

3.4 Software

The algorithm and prototypes of scheduler are available at https://gitlab.arcos.inf.uc3m.es/aspide/dcex.

Chapter 4

Conclusions

The work done in this deliverable corresponds with Tasks 2.2, which aims is to Facilitate the exploitation of heterogeneous and distributed memory systems in data intensive applications.

In this deliverable we described the main concepts and the prototype of the DCEx runtime system developed by the integration of the DCEx runtime model into the GrPPI (Generic Reusable Parallel Pattern Interface) framework. The main elements and the runtime mechanisms of DCEx have been presented in a detailed way to describe the associated software prototype developed in WP2 that includes the new GrPPI-based execution model for the execution of massively parallel scalable applications.

The DCEx programming model designed in WP2 is supported by 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.

The main activities of WP2 consisted in the construction and management of the internal run-time representation of the DCEx model, integrated into the GrPPI framework, as a dynamic DAG where nodes represent computational components and edges data dependencies between components. The parallel execution of DCEx programs is based on data-driven, dynamic DAG scheduling mechanisms, where a component in the DAG will be ready for execution once all of its required input parameters have become available.

The deliverable has been organized in three main chapters. Chapter 1 introduced how the DCEx programming framework and its components are combined with the GrPPI runtime. Chapter 2 described how the GrPPI language has been extended with the programming constructs and mechanisms introduced by the DCEx programming model. In the first section of the chapter, we introduced the main entities that

interact with the runtime, i.e., Master and Workers entities, $Cnode$ and $Carea$, Data Parallel Block and Partition, Task and Program. The second section deals with the currently defined execution models in GrPPI and the new execution model proposed to integrate GrPPI with DCEx. Finally, Chapter 3 discussed details of the DCEx runtime support such as communication mechanism for tasks scheduling and temporary data, task generation and task scheduling.

The deliverable summarizes the choices done for the implementation of the runtime prototype and describes our research work that resulted in the extension of the GrPPI programming interface with the programming constructs and mechanisms of DCEx like $Carea$, $Cnode$, and DPB. Indeed, the main goal of this deliverable is to illustrate the details of runtime prototype and its the functioning in the execution of parallel applications. Some code examples have been used to show how new GrPPI+DCEx applications can be designed and executed. Moreover, the runtime support is described together with the communication mechanisms for tasks scheduling and temporary data, the functioning of the scheduler and the managing of tasks.

Bibliography

[1] Ioan Raicu, Ian T. Foster, and Pete Beckman. Making a Case for Distributed File Systems at Exascale. In *Proceedings of the Third International Workshop on Large-scale System and Application Performance*, LSAP '11, pages 11–18, New York, NY, USA, 2011. ACM.

[2] Marc Duranton, Koen De Bosschere, Bart Coppens, Christian Gamrat, Madeleine Gray, Harm Munk, Emre Ozer, Tullio Vardanega, and Olivier Zendra. The hipeac vision 2019, 2019.

[3] Fabrizio Marozzo, Francisco Rodrigo Duro, Javier Garcia Blas, Jesus Carretero, Domenico Talia, and Paolo Trunfio. A data-aware scheduling strategy for workflow execution in clouds. *Concurrency and Computation: Practice and Experience*, 29(24), 2017. ISSN: 1532-0634.

[4] Domenico Talia, Paolo Trunfio, Fabrizio Marozzo, Loris Belcastro, Javier Garcia Blas, David Del Rio, Philippe CouvÃ©, Gael Goret, Lionel Vincent, Alberto FernÃ ndez-Pena, Daniel Martin de Blas, Mirko Nardi, Teresa Pizzuti, Adrian Spataru, and Marek Justyna. A novel data-centric programming model for large-scale parallel systems. In *Euro-Par 2019: Parallel Processing Workshops*, Lecture Notes in Computer Science, pages 452–463, Gottingen, Germany, 26-30 August 2020. ISBN: 978-3-030-48339-5.

[5] David del Rio Astorga, Manuel F. Dolz, Javier Fernández, and J. Daniel García. A generic parallel pattern interface for stream and data processing. *Concurrency and Computation: Practice and Experience*, Online:e4175–n/a, April 2017.

[6] David del Rio Astorga, Manuel F. Dolz, Luis Miguel Sanchez, Javier García Blas, and J. Daniel García. A c++ generic parallel pattern interface for stream processing. In Jesus Carretero, Javier Garcia-Blas, Ryan K.L. Ko, Peter Mueller, and Koji Nakano, editors, *Algorithms and Architectures for Parallel Processing*, pages 74–87, Cham, 2016. Springer International Publishing.

[7] Rohit Chandra, Leo Dagum, David Kohr, Ramesh Menon, Dror Maydan, and Jeff McDonald. *Parallel programming in OpenMP*. Morgan kaufmann, 2001.

[8] Marco Aldinucci, Marco Danelutto, Peter Kilpatrick, and Massimo Torquati. Fastflow: high-level and efficient streaming on multi-core. *Programming multi-core and many-core computing systems, parallel and distributed computing*, 2017.

[9] *ZeroMQ: messaging for many applications*.