A Cluster-As-Accelerator approach for SPMD-free Data Parallelism

Maurizio Drocco, Claudia Misale, Marco Aldinucci
Computer Science Department
University of Torino
Torino, Italy
{drocco, misale, aldinuc}@di.unito.it

***Abstract*—In this paper we present a novel approach for functional-style programming of distributed-memory clusters, targeting data-centric applications. The programming model proposed is purely sequential, SPMD-free and based on high-level functional features introduced since C++11 specification. Additionally, we propose a novel cluster-as-accelerator design principle. In this scheme, cluster nodes act as general interpreters of user-defined functional tasks over node-local portions of distributed data structures. We envision coupling a simple yet powerful programming model with a lightweight, locality-aware distributed runtime as a promising step along the road towards high-performance data analytics, in particular under the perspective of the upcoming exascale era. We implemented the proposed approach in SkeDaTo, a prototyping C++ library of data-parallel skeletons exploiting cluster-as-accelerator at the bottom layer of the runtime software stack.**

***Index Terms*—skeletons, cluster computing, data-centric, parallel programming, skedato, exascale**

I. Introduction

Programming distributed-memory clusters for high performance data-centric applications is a big challenge nowadays. To exploit large-scale clusters massive computational capabilities, user is typically provided with parallel programming models based on a *Single-Program-Multiple-Data* (SPMD) execution model, in which the same program is executed by different processes (i.e. cluster nodes). Message-Passing Interface (MPI) [1] has become the *de-facto* standard for SPMD programming of distributed environment. In a MPI application, parallelism is explicit and the programmer is responsible for designing and implementing applications from scratch according to MPI constructs.

Algorithmic skeletons [2] have been proposed as an abstraction aiming at capturing common parallel programming patterns (e.g. ForAll, MapReduce, Divide&Conquer, etc.) making them available as high-level programming constructs. Skeletons abstract all concurrency exploitation, mapping or scheduling aspects, by moving them from the application to tools runtime level. Several skeleton-based approaches targeting distributed systems and relying on the SPMD programming model have been proposed [3], [4].

Further along the road traced by skeletons, tools for big data analysis such as Spark [5] are gaining popularity, also by changing the way users implement data-centric applications on distributed clusters. The key aspect is very simple: expose to the programmer an abstract datatype on which to apply data-parallel, functional-style operators while writing sequential code. Allocations and parallelism execution is completely hidden and managed by the runtime.

Partially inspired by these frameworks, we propose to overcome the SPMD programming model by providing a model in which user has visibility exclusively on the sequential execution flow defining the sequence of transformations to be applied, while the parallel execution is managed by an optimized distributed runtime. This approach results in pure sequential user codes, endowed with a skeleton library for applying element-wise kernels to distributed data collections.

Moreover, mentioned high-level platforms typically rely on heavyweight runtimes, inhibiting them from meeting high performance requirements. We propose a dual architecture shaped as a thin stack on top of a lightweight C++ runtime, by artificially overcoming some well-known C++ limitations. We furthermore enrich our vision with the idea of using clusters as external-memory hardware accelerators, which allows both decoupling the user memory and execution models from the cluster, and integrating low-level communication libraries (e.g. MPI, POSIX sockets, libfabric verbs) in form of pluggable software components. We prototyped the proposed approach in SkeDaTo, a C++ data-parallel skeletons.

II. Related Works

Various approaches have been proposed for simplifying distributed-memory systems programming.The Partitioned Global Address Space (PGAS) model provides a global memory space, still exposing the underlying memory partitioning in order to permit locality-based optimizations. Language-level PGAS formalisms have been proposed in form of either language extensions (e.g. UPC [6]) or standalone languages (e.g. Chapel [7], X10 [8]). A notable exception is Global Arrays [9], a Fortran framework providing one-sided primitives and atomic operations over distributed-shared data structures.

Skeleton-based frameworks have been proposed targeting distributed systems. Muesli [4] is a C++ template library on top of MPI and OpenMP supporting multi-processor and multi-core. Muesli provides skeletons such as *map*, *fold*, *scan* and variants, applied as member functions to elements of distributed data structures such as distributed arrays, matrices, and sparse matrices. SkeTo [3] is a C++ library based on MPI

and C++ expression templates, providing skeletons for distributed arrays, matrices, and trees. SkePu [10] is a C++ open-source skeleton library for multicore CPUs and multi-GPU systems based on "smart" containers (i.e. vectors and matrices) providing automatic data partitioning for execution on multi-GPU systems both with CUDA and OpenCL. Lithium [11] and its successor Muskel [12] are Java skeleton frameworks developed at University of Pisa, handling both data and task parallelism in form of macro-data flow execution.

The Scala/Java big data analytics framework Spark [5] offers an abstraction for distributed datatypes called Resilient Distributed Datasets (RDDs), namely a distributed memory abstraction for in–memory computations on large clusters, providing a restricted form of shared memory based on coarse–grain transformations.

III. SkeDaTo Programming Model

At the highest level, a SkeDaTo program is a piece of plain *sequential*[1] C++ code. By decoupling the programming model from the runtime execution model, user can exploit all the available computational resources without being aware of the underlying parallel execution environment. Dually, the runtime can be shaped on the underlying platform, thus exploiting modern heterogeneous architectures (e.g. multi-threaded runtime for multi-core cluster nodes).

A. SPMD-free data-centric programming

We recognize at a fundamental limitation imposed by SPMD when used for programming data-centric distributed applications. The element-wise nature of such computations is not naturally captured by SPMD, which expresses the parallelism focusing on the process (i.e. a sequential execution flow) rather than the datum (i.e. the atomic element of a collection). The resulting programming model is process-centric rather than data-centric, thus motivating the proposal of several data-parallel skeleton libraries (see Section II).

In SkeDaTo, we propose a symmetric approach, namely allowing users to program just the orchestration logic by expressing application business code in terms of plain C++ sequential code, with no need to "think in parallel". Furthermore, in order to support high-level data-centric programming, user code is endowed with SkeDaTo skeletons for expressing element-wise computations over distributed collections.

B. The Data-Parallel Collection abstract data type

The core abstraction in SkeDaTo is the Data-Parallel Collection (DPC) abstract data type (ADT), provided in form of C++ template class. A DPC represents a collection of elements that can be accessed, transformed and filtered by means of user-defined data-parallel computations, that in SkeDaTo are called *DLambda*s and share the design principles underlying the skeletal approach for parallel programming.

[1]here and in the following, we use the term *sequential* for denoting a piece of code that is run by a single processor in a multi-processor execution environment, such as a distributed-memory cluster of nodes.

```
1 int main() {
2   ...
3   skedato::DLambdaInMap<myPair> mapL(
4     [] (myPair x) {
5       return myPair(x.a, x.a + x.b);
6     });
7
8   skedato::init();  //SPMD-free from here
9
10  skedato::DPC<myPair> dpc(LEN);
11  ...
12  dpc.map_inplace(mapL);
13  ...
14 }
```

Listing 1. A sample basic SkeDaTo application

1) Data-parallel skeletons: In SkeDaTo, operations over DPC collections are expressed in terms of functional data-oriented computations. Let **map** $f[a_0, a_1, \ldots a_{n-1}] = [f(a_0), f(a_1), \ldots f(a_{n-1})]$ and **reduce** $\oplus\ [a_0, a_1, \ldots a_{n-1}] = a_0 \oplus a_1 \oplus \ldots a_{n-1}$, where $f : T \to T$ is the *elemental function*, $\oplus : T \times T \to T$ the *combinator* (i.e. a binary associative operator) and $a = [a_0, a_1, \ldots a_{n-1}] \in T^n$ an array of atomic elements having type T. They are well-known examples of data-parallel patterns, since the elemental function of a map can be applied to each input element independently from the others, and the combinator is applied to different pairs in the reduction tree independently, thus naturally inducing a parallel implementation. The operations that complement the ADT definition of DPC collections are classical data-parallel skeletons (e.g. map, stencil, reduce).

2) DLambdas: A data-oriented application is characterized by manipulations performed on input data. In functional terms, element-wise functions (i.e. the parameters of data-parallel skeletons) represent the business code of the application. In SkeDaTo, element-wise functions are template C++ functors called DLambdas. They can be either *dynamic* – taking the element type as template argument and any functor as argument of the class constructor – or *static* – taking both element type and a global function as template arguments. Dynamic DLambdas offer more flexibility at the cost of some runtime overhead, while static DLambdas offer the best performance at the price of a slightly more verbose user interface.

C. A sample SkeDaTo application

Listing 1 shows the C++ code of a sample SkeDaTo application. The application creates a fixed-length DPC collection of integer pairs and applies the transformation $g(\langle a, b\rangle) = \langle a, a + b\rangle$ to each pair. Main components of SkeDaTo are shown, namely: the first section of the `main` function is the definition of a dynamic DLambda object (representing the element-wise transformation g), SkeDaTo runtime is initialized by calling `skedato::init` function, a collection named `myDPC` with fixed length `LEN` is created via `DPC` constructor (line 10) and transformed by applying the in-place transformation `DPC::map_inplace` with argument `mapL` (line 12).

IV. SkeDaTo Architecture

We implemented SkeDaTo in such a way that it is easy to maintain a clear separation among all components involved. Moreover, this modularity – together with the intrinsic component decoupling provided by cluster-as-accelerator design – led to the creation of a framework in which low-level components can be easily replaced with different architecture-dependent implementations. SkeDaTo architecture is a stack of five layers: 1) User-level layer, 2) Runtime as user-level DPC manager, 3) Virtual Accelerator as interface to the cluster, 4) Stub for Virtual Accelerator tasks offloading and 5) Remote cluster accessible via workers Stubs. In the following we describe each level of the architecture more in detail.

1) User-level code, the first level of abstraction: This layer is the only one exposed to the programmer. As illustrated in section III, the programmer writes sequential code describing the sequence of transformations applied to the dataset.

2) Runtime, a user-level interface to the accelerator: the runtime purpose is to make the upper layer see the cluster as a black-box. As intermediate layer, the runtime has the main responsibility of accepting requests coming from the user level and redirecting them to the cluster via the Virtual Accelerator primitives. Requests are issued by operations applied to DPC collections declared and called by user's source code.

3) Virtual Accelerator, the front-end to the Cluster-As-Accelerator: The Virtual Accelerator layer is responsible for both DPC data partitioning and effectively offloading tasks to the cluster. It takes requests from the runtime via a light interface offering offload and get results routines. The VA has the complete knowledge about how dataset is partitioned among cluster executors, hence it can issue tasks execution to workers with messages sent through the Stub.

4) Stub, an object for communication logic: Object responsible for communication between the runtime and the cluster, From the cluster side, there exists a Stub for each worker, acting as the backend that gathers requests from the VA. Therefore, the Stub is responsible for: 1) Initiating the communication towards the cluster nodes, 2) Sending VA messages parametrized with both the task to execute and the DPC partition, 3) Marshall and unmarshall outcoming and incoming messages, and 4) Informing the VA that the remote call is completed.

5) Cluster-level, the last layer of the stack: As partially described, executors in the cluster communicate with the VA (as well as with the runtime) by means of messages mediated by the Stub.

DLambda VTable: The DLambda VTable is an important piece of the architecture and it is reported in Figure 1. In order to make all DLambdas accessible by remote workers within the same executable, just addressing those via function pointers is not enough because of dynamic relocation. To overcome this problem, we implemented a class containing a pair of tables making DLambdas addressable both by an index and by their names. We called this class *DLambdaVtable* as the idea is to emulate virtual tables.

2^{30} `double`/`int` DPC collection ($4 + 8 = 12\,\mathrm{GB}$)				
nodes	init (ms)	zip (ms)	reduce (ms)	speedup
sequential implementation				
1	6202.60	3102.12	1688.31	1.00
SkeDaTo implementation				
1	6705.09	3097.54	1812.16	0.95
2	3474.70	1642.60	962.26	1.81
4	1743.08	822.41	482.44	3.61
native MPI implementation				
4	1583.85	822.91	481.60	3.81

TABLE I
PERFORMANCE OF DOT-PRODUCT ON THE CLUSTER.

A MPI-based implementation: The current SkeDaTo implementation is based on top of MPI 2-sided communication primitives. The MPI_Engine is the base class for the runtime in a SkeDaTo program. In the MPI implementation, after initializing the MPI environment, the execution is routed into two flows: 1) the runtime branch, which proceeds to the user main code (that is, the User Level) and 2) workers branch, in which processes are redirected to the "cluster dedicated code", from which they never exit. In order to exploit the MPI SPMD execution model, the *MPI_Stub* acts as the Virtual Accelerator, instructing workers via MPI communication facilities.

V. Experimental Evaluation

We performed some preliminary experiments in order to both showing the scalability of a simple SkeDaTo application with respect to the number of cluster nodes and – from the absolute performance perspective – comparing execution times of SkeDaTo application against a native MPI implementation.

The dot-product of two collections A and B both having size n is defined as $A \cdot B = \sum_{i=1}^{n} A_i B_i = A_1 B_1 + A_2 B_2 + \ldots + A_n B_n$ and can be implemented by means of skeletons as a sequence of a *zip* (i.e. element-wise binary combination) followed by a *reduce* skeleton. In particular, we implemented the elemental functions as static DLambdas for best performance.

All experiments were conducted on the Paradigm cluster at the Computer Science department of Torino, a small-scale representation of widespread massively parallel clusters used for data processing nowadays. It is composed of 4 low-cost low-energy nodes, each equipped with an Intel Atom C-2750 @$2.40\,\mathrm{GHz}$ and $16\,\mathrm{GB}$ of main memory with Linux x86_64.

Table I reports the execution time breakdown of the dot-product application over two DPCs containing 2^{30} double-precision floating-point and $32\,\mathrm{bit}$ integer numbers, respectively (i.e. $4 + 8 = 12\,\mathrm{GB}$). Execution times are reported for both zip and reduce skeletons, plus an *init* phase, spanning DPC creation and initialization. Reported times are averaged over 100 runs, featuring standard deviations in the order on $10\,\mathrm{ms}$. First we compare SkeDaTo execution against a plain sequential implementation, considering as a performance measure the *speedup*, defined as the ratio of sequential to

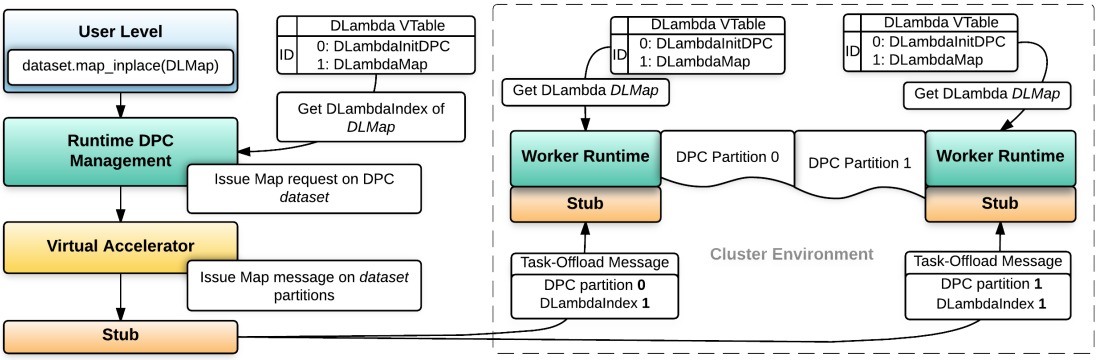

Fig. 1. SkeDaTo execution example for a Map task.

SkeDaTo total execution times. Speedups, reported for k-worker SkeDaTo execution with $k = 1, 2, 4$, result to be almost ideal in all cases. Moreover, reported speedups grow with almost linear rate with respect to k, thus showing good scalability. Finally, we compared the SkeDaTo implementation against a native MPI implementation of the same application. MPI execution, reported at the bottom of Table I for the 4-worker case, results to be just slightly faster.

VI. Conclusions and Future Works

In this work we have proposed a novel approach for data-centric cluster computing, which contribution is twofold. First, from a usability perspective, it is based on a *sequential SPMD-free* programming model, freeing users from reasoning at the parallel coordination level and from dealing with performance-critical aspects such as data partitioning and placing, while letting them focusing exclusively on business code. Second, from high performance runtime perspective, it is based on a cluster-as-accelerator architecture, in which cluster plays the role of interpreter of user-defined The resulting architecture is highly modular, in particular the bottom runtime layer can be implemented on top of different communication libraries (e.g. MPI or libfabric verbs).

As future directions, we consider exploiting some form of Just-In-Time (JIT) compilation in order to provide inlining for a wider class of user-defined functions (i.e. dynamic DLambdas in Section III-B2) to be executed by the clustered interpreter. A partially related aspect is a more tight integration with C++11/14 specifications. Finally, SkeDaTo will be provided with more skeletons and more I/O options for SkeDaTo collections. In particular, we plan to enrich the MPI-based implementation by exploiting the improved 1-sided Remote-Memory-Access (RMA) primitives – introduced since MPI2 – for implementing data exchange between workers. This would be useful in e.g. implementing the well-known *stencil* skeleton.

Acknowledgment

This work has been partially supported by the EU FP7 REPARA project (no. 609666), the EU H2020 Rephrase project (no. 644235) and the 2015-2016 IBM Ph.D. Scholarship program.

References

[1] M. P. Forum, "Mpi: A message-passing interface standard," Knoxville, TN, USA, Tech. Rep., 1994.

[2] M. Cole, *Algorithmic Skeletons: Structured Management of Parallel Computations*, ser. Research Monographs in Par. and Distrib. Computing. Pitman, 1989.

[3] K. Matsuzaki, H. Iwasaki, K. Emoto, and Z. Hu, "A library of constructive skeletons for sequential style of parallel programming," in *Proc. of the 1st Inter. conference on Scalable information systems*, ser. InfoScale '06. New York, NY, USA: ACM, 2006.

[4] P. Ciechanowicz, M. Poldner, and H. Kuchen, "The Munster skeleton library Muesli – a comprehensive overview," in *ERCIS Working paper*. ERCIS – European Research Center for Information Systems, 2009, no. 7.

[5] M. Zaharia, M. Chowdhury, T. Das, A. Dave, J. Ma, M. McCauley, M. J. Franklin, S. Shenker, and I. Stoica, "Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing," in *Proceedings of the 9th USENIX Conference on Networked Systems Design and Implementation*, ser. NSDI'12. Berkeley, CA, USA: USENIX Association, 2012, pp. 2–2.

[6] T. El-Ghazawi, W. Carlson, T. Sterling, and K. Yelick, *UPC: distributed shared memory programming*. John Wiley & Sons, 2005, vol. 40.

[7] B. L. Chamberlain, D. Callahan, and H. P. Zima, "Parallel programmability and the chapel language," *IJHPCA*, vol. 21, no. 3, pp. 291–312, 2007.

[8] P. Charles, C. Grothoff, V. A. Saraswat, C. Donawa, A. Kielstra, K. Ebcioglu, C. von Praun, and V. Sarkar, "X10: an object-oriented approach to non-uniform cluster computing," in *Proceedings of the 20th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2005, October 16-20, 2005, San Diego, CA, USA*, 2005, pp. 519–538.

[9] J. Nieplocha, B. J. Palmer, V. Tipparaju, M. Krishnan, H. Trease, and E. Aprà, "Advances, applications and performance of the global arrays shared memory programming toolkit," *IJHPCA*, vol. 20, no. 2, pp. 203–231, 2006.

[10] J. Enmyren and C. W. Kessler, "Skepu: a multi-backend skeleton programming library for multi-gpu systems," in *Proceedings of the fourth international workshop on High-level parallel programming and applications*, ser. HLPP '10. New York, NY, USA: ACM, 2010, pp. 5–14.

[11] M. Aldinucci, M. Danelutto, and P. Teti, "An advanced environment supporting structured parallel programming in Java," *Future Generation Computer Systems*, vol. 19, no. 5, pp. 611–626, Jul. 2003.

[12] M. Aldinucci, M. Danelutto, and P. Dazzi, "Muskel: an expandable skeleton environment," *Scalable Computing: Practice and Experience*, vol. 8, no. 4, pp. 325–341, Dec. 2007.