Distributed futures for efficient data transfer between parallel processes

Pierre Leca, Wijnand Suijlen, Ludovic Henrio, Françoise Baude

Distributed futures for efficient data transfer between parallel processes

Pierre Leca
Wijnand Suijlen
Huawei Technologies
Boulogne Billancourt, France
{firstname.lastname}@huawei.com

Ludovic Henrio
Univ Lyon, EnsL, UCBL, CNRS, Inria, LIP – Lyon, France
ludovic.henrio@cnrs.fr

Françoise Baude
Université Côte d'Azur, CNRS, I3S
Sophia Antipolis, France
francoise.baude@univ-cotedazur.fr

ABSTRACT

This paper defines distributed futures, a construct that provides at the same time a data container similar to a distributed vector, and a single synchronization entity that behaves similarly to a standard future. This simple construct makes it easy to program a composition, in a task-parallel way, of several massively data-parallel tasks. The approach is implemented and evaluated in the context of a bulk synchronous parallel (BSP) active object framework.

CCS CONCEPTS

• **Software and its engineering → Parallel programming languages**; **Concurrent programming structures**.

KEYWORDS

Parallel and distributed programming, Futures

ACM Reference Format:
Pierre Leca, Wijnand Suijlen, Ludovic Henrio, and Françoise Baude. 2020. Distributed futures for efficient data transfer between parallel processes. In *The 35th ACM/SIGAPP Symposium on Applied Computing (SAC '20), March 30-April 3, 2020, Brno, Czech Republic.* ACM, New York, NY, USA, Article 4, 4 pages. https://doi.org/10.1145/3341105.3374104

1 CONTEXT AND INTRODUCTION

Two of the most common parallel abstractions are *task-parallelism*, which decomposes work into different parts that can be executed in parallel and can be functionally different, and data-parallelism, which splits the work by distributing the data. Because task parallelism and data parallelism are convenient to parallelize different parts of an application, it is valuable to mix them into one programming framework. In data-parallel models, synchronization and data exchanges between the tasks are quite restricted while task parallelism is generally very flexible on these aspects. This is why several interaction patterns exist in task parallel models, we focus here on futures, a programming construct that both serves to synchronize tasks and to exchange data. While futures are convenient for task-parallel models, they are not very well integrated in data-parallel models. To enable a better interaction of these two programming models, this article designs and shows how to implement *distributed futures*.

A *future* is a placeholder for a value being computed by a task. While the task is not finished, the future is *unresolved*, automatically, the future gets filled by the value computed by the task when it finishes (the future is *fulfilled*). Futures can also be accessed by trying to *get* their value, synchronizing the current task with the resolution of the future. Several mechanisms exist for accessing futures, either blocking on the result or registering a continuation to be executed when the future is resolved. All these mechanisms provide a convenient and safe way to write parallel and distributed applications, with a behavior close to a sequential program.

Here, we choose actors as the task-parallel model, and BSP (Bulk Synchronous Parallel [13]) as the data-parallel programming framework. The *Actor* [1] model is a task-parallel paradigm. Actors prevent data-races by enforcing that asynchronous message passing is the only interaction between processes: actors communicate with each other by putting messages in their mailboxes. In this article we use active objects [4], which are objects that are at the same time actors. In active objects, a method call to an active object creates a message that reaches its mailbox and a future is used to represent the result returned by such an asynchronous method invocation. Data-parallel programming abstractions like BSP are better suited to parallel computations on large amounts of data. BSP algorithms are defined as a sequence of supersteps, each made of three phases: computation, communication, and synchronization. BSP is limited in terms of application elasticity or loose coupling of computing components as it relies on the strong synchronization of all computing entities.

The scope of our contribution is broader than the strict context of BSP and active objects. As soon as one wants to compose task-parallel and data-parallel programming models, the question of the interaction between the synchronization mechanisms arises. In this article, we focus on *futures*, which are synchronization artefacts frequently used in task-parallel applications. However, because a future encapsulates a single piece of data on which synchronization is possible, it is not suited to the context of data-parallelism that requires data to be spread over different processes. This is why we design, implement, and evaluate *distributed futures*: futures *representing* data distributed over multiple processes.

1.1 Futures in programming languages

Many languages use futures because they provide a high-level synchronization paradigm. Futures are used in actors [14], active objects [4], Synchronous languages [6], but also many mainstream

SAC '20, March 30-April 3, 2020, Brno, Czech Republic

ACM ISBN 978-1-4503-6866-7/20/03.
https://doi.org/10.1145/3341105.3374104

languages like C++ or Java. Basic future usages include synchronization of a process on the availability of the future's referred data (i.e. the resolution of the future) and asynchronous reaction to the resolution of a future, i.e. registration of a continuation to be executed upon future resolution. The synchronization primitive is generally called `get` and waits for a future to be resolved while blocking the current thread of execution. Some languages like Akka [14] favour asynchronous chaining of the form `f.onSuccess(...)` that does not execute the consequence of the future resolution immediately, but registers what is to be executed when `f` becomes available.

1.2 A data-parallel and task-parallel framework: BSP Active Objects

As highlighted above, data-parallelism and task-parallelism should benefit from each other and both are often integrated in the same application. A few frameworks for coupling different parallel codes support the design of these applications.

In particular, we rely on BSP Active Objects[9], a C++ library that allows the coordination of several data-parallel tasks implemented in BSP by using an actor-based task-parallel interaction with basic (non distributed) futures. It runs on top of MPI and uses the BSPonMPI v0.2 implementation of the BSPlib specification.

A BSP active object is running on several processes, one of them is called the *head process*; this process handles the active object requests sequentially, it is able to run a parallel function by giving it as parameter of a `bsp_run` primitive. This parallel function is executed on all the processes of the active object, which communicate in a BSP manner. A class can be declared as an active object class, tagging some of the methods as "active object methods". Each BSP process has two threads. The first one is a worker thread that executes the user's code. The other one is called a *management thread*: it ensures responsiveness of the active object as it is available when the processes are inside a BSP computation. We will use this per BSP process thread to perform tasks dedicated to our distributed future management. The use of BSP active objects is illustrated in [9].

2 DISTRIBUTED FUTURES

2.1 Principle

To incorporate data-parallelism inside a task-parallel framework, the most efficient solution is to use multiple processes for each entity that handles a parallel task. This means that one needs to gather all the parts of the computed result into a single place in order to return back a result as a future even if it was distributed among processes. This gathering raises a performance issue whenever the result array is large, especially when passed to another task that scatters it again to do data-parallel processing. Consequently, it is more efficient that every data-parallel process keeps its part of the result, and transmits it directly where it is needed. The data-parallel processes thus only need to send a *description* of the part of the result it holds. We call this description a *distributed future* because it *represents* a *distributed vector* being computed in the same way a future represents a value being computed. *A distributed future is a future on which synchronization is possible, but its content is the description of a distributed vector.* Provided the distributed future is way smaller than the distributed vector, a distributed future is cheaper to pass around between tasks. Using a distributed future, any process in a data-parallel task can obtain the distributed vector parts it needs, directly from the processes that hold them. The programmer does not have to know where each part is located and on how many processes each part is distributed.

Resolving a distributed future means waiting to get the metadata, which is returned from an active object producing a distributed vector. However the meta-data is only produced after the distributed vector is produced. *This means resolving a distributed future waits for the full distributed vector be produced, but only transfer its metadata.* Every process only receives the data it is interested in, and directly from process(es) that computed these data. A synchronization on a distributed future consists in retrieving metadata necessary to access the content of the distributed future value, then the different part(s) of the effective data collection. Our design is such that requesting the value of a distributed future and making use of it to trigger effective data transfer is similar to using a *lazy* synchronization strategy with classical futures: the data is only transmitted upon need. Indeed, because data parallelism is often bandwidth-bound, we need precise control of the communication when large amounts of data may be communicated over multiple processes.

A more declarative strategy can also be envisioned: the programmer could declare a distribution policy inside each actor and this policy could be used by the different data-parallel processes hosted in this actor to pre-fetch the data before the computation is started. Such a pre-fetching strategy is outside the scope of this paper.

2.2 Implementation

Our implementation of distributed futures is based on the BSP active object library, described in Section 1.2 above. To implement the concept of distributed futures, two aspects have to be implemented: the future resolution and the future access. We review our solution for both of them below. We start by defining a data structure to represent a distributed vector, this data structure is stored in the distributed future when it is resolved.

When a distributed future is resolved, we assign it a collection of (*pid*, *local_id*, *size*, *offset*) quadruplets. Each quadruplet describes a part of a distributed vector. *pid* is the process which owns this part; *local_id* is the part identifier that is unique within the owner; *size* is the part size; *offset* is the index in the distributed vector. Fields offset and size are specified in bytes. For example, a part with offset 1 and size 1 is the second byte of a distributed vector. This structure allows storing different types of contiguous elements, including structs. The simplest data distribution is the block distribution, which stores one contiguous range per process, in this case the distributed future is made of one quadruplet per process. For example, if we have a block-distributed vector of 40 elements computed by 4 processes numbered 1, 2, 3 and 4, then the distributed future value is the list $((1, 1, 10, 0), (2, 12, 10, 10), (3, 41, 10, 20), (4, 33, 10, 30))$. Processes own consecutive parts of size 10 each.

We provide 4 main high-level primitives for manipulating `vector_distribution`, shown in Figure 1. Their following description can be followed along figure 2.

The `register_result` function stores a distributed vector part of size `size` at local address `data` into the management thread's memory of the current process; `offset` is the position of this

```
void register_result(const char *data, size_t size, size_t offset);
vector_distribution gather_vd_parts();
void broadcast_vd(vector_distribution & vd);
void get_part(const vector_distribution & vd,
              size_t offset, char *buf, size_t size);
```

Figure 1: Vector distribution primitives

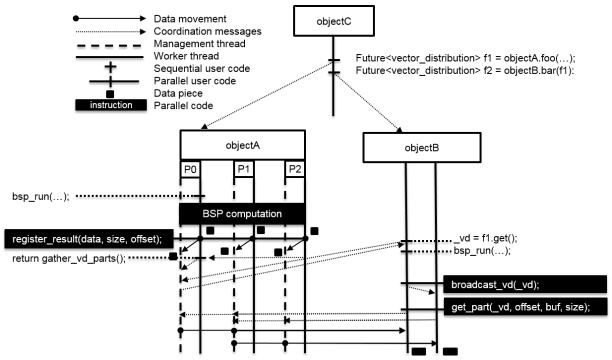

Figure 2: Distributed future API example

```
std::string path;
std::ifstream is(img_list_file); // File with list of path
while (getline(is, path)) {
    Future<distr_vector> img = object1.loadImage(path);
    Future<distr_vector> compressed = compressor.compress(img);
    Future<int> inserted = object2.insert(compressed);
}
```

Figure 3: Main part of the coordinator process

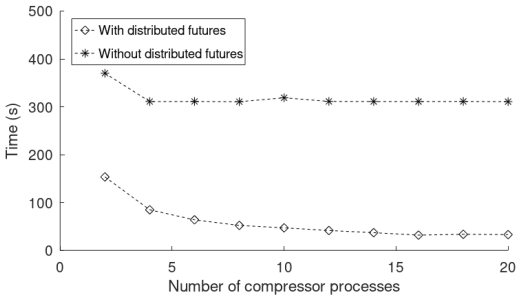

Figure 4: Execution time for inserting 1000 images as function of the number of compressor processes

part within the distributed vector. This step creates a quadruplet as defined in Section 2.2. After all parts are registered, a call to `gather_vd_parts` by the head process assembles the quadruplet list as a `vector_distribution` structure, which, when given as return value, becomes a distributed future. Resolving this future with an usual `get` returns this `vector_distribution`, after the associated distributed vector was produced. The `broadcast_vd` primitive allows a head process to send it to its other active object processes. The `get_part` primitive then enables any of these processes to request any subpart of the distributed vector, where its distribution is transparently deduced from the `vector_distribution` given as parameter. In Figure 2, the second process of `objectB` requests the second half of the distributed vector, which is deduced to be on `P1` and `P2` of `objectA`.

3 EXPERIMENTS

In this section, we demonstrate the performance gain of distributed futures when used between parallel actors. To do so, we create a pipeline of three such actors, and we focus on the middle one that receives a distributed future, work with it, and produces another. This actor is a parallel image compressor, of which the performance vary with the number of assigned processes. We configure other actors so that the compressor is the bottleneck in the pipeline. The pipeline is set-up from a coordinator process, which executes a code similar to figure 3.

3.1 Experimental setting

For these experiments, we are using seven Huawei RH2288v2 servers, each with two Intel Xeon E5-2690v2 CPUs that have ten cores each. We use the Intel C++ compiler version 18.0.1. Because each BSP active object process uses two threads, we put a maximum of ten processes on each of these servers. When an active object is assigned more than ten processes, it means it is distributed over multiple nodes. For example twenty processes are distributed over two. We dedicate one of our servers to the main coordinator process, so that all active objects are on remote nodes.

3.2 Results

We choose large image sizes: 36 Mega-pixel images of resolution 4912 x 7360. Each of these images amounts to about 108 MB uncompressed in pure bitmap format. We execute our pipeline for 1000 of these images, measure its performance while varying the number of compressor processes, as shown in figure 4. Because the compressor is the bottleneck in the pipeline, the whole pipeline performance improves with the compressor performance. Here the performance stops improving at sixteen processes with distributed futures and four processes with normal futures. This experiment clearly shows the advantages of distributed futures, showing the gain brought by parallel data transfers instead of gathers and scatters.

4 RELATED WORKS

Several works focus on the efficient use of futures in concurrent and distributed settings [2, 7, 12], sometimes synchronizing a group of tasks, but none of them use a single future as the abstraction of a large set of data. In the domain of parallel and distributed computing, such an abstraction is generally provided by distributed arrays. To the best of our knowledge we are the first to define futures of distributed data in the form of distributed vectors.

ParT [8] in the Encore language, provides the notion of arrays of futures that distributes data as different futures, but does not allow to synchronize parts of data as a single future. It can be used to implement speculative parallelism or barriers gathering a set of results. The set of futures is not viewed as a distributed array but rather as an array of futures. Also, the implementation is local to a single machine and there is no support for distributing a ParT

over multiple machines or the possibility to transparently allocate a range of data to a process.

The notion of *streaming futures* was defined in the context of ABS in [2]. This approach provides a solution for tailoring futures to large amounts of data and in particular data streams. Streaming futures can be accessed multiple times to obtain different data, which departs from the traditional future concept. The advantage is that such futures can create a streaming channel, and the communication pattern can then be optimized using existing streaming techniques. Our distributed futures are geared more toward high-performance computing than to data-streaming applications, because a single synchronization waits for completion of all data.

Distributed arrays, have been used as basic data structures for data parallel algorithms since the early days of parallel computing. In particular, they solve a scalability issue. On distributed memory systems, where memory is partitioned, an array that could not be stored on a single node can be stored on a distributed memory. In the context of (direct mode) BSP programming, distributed arrays are a natural way of dealing with integer-indexed data, because the programming model assumes distributed memory. For that reason, many BSP programming languages and frameworks support distributed arrays, sometimes implicitly like registered memory in BSPlib [10], and other times more explicitly like the parallel vectors in BSML [3] and coarrays in Bulk [5]. OSL [11] too is a library of data parallel algorithmic skeletons which follow the BSP model. OSL arrays are distributed but are manipulated as normal arrays by the programmer, passing them as parameter to functions such as parallel map or zip which return a new distributed array as result, that can further be passed as parameter. OSL proposes to avoid the creation of intermediate distributed arrays within a sequence of supersteps. On the contrary, our distributed futures allow a vector to be passed around to any method call without the need to delimit a sequence of calls. The use of futures triggers a synchronization but the data transfer is decoupled. Moreover, contrary to OSL where arrays must be block-distributed, we can define an unbalanced distribution of the future.

Distributed arrays require redistribution usually when two parallel programs exchange data, because, for example, the number of processes doesn't match, or the problem domain favors a different distribution. E.g. in MPI, programs can use intercommunicators to transfer data between producer and consumer tasks. However, both must know exactly how data is redistributed, something that must be specified on both ends by the programmer. In frameworks with shared-memory like OpenMP, all offset and address calculations and the synchronization can be done by the consumer task itself. Still, the programmer must know exactly which process owns what data on the producer task. Contrarily to the MPI-like approaches, our distributed futures offer a better abstraction of distribution than classical distributed arrays. The distribution information is stored with the future as meta-data, obtained upon synchronization, and the programmer doesn't need to know its content or to ask individual processes for subparts.

5 CONCLUSION

We presented the concept of distributed futures, an unification of futures and distributed arrays where a distributed future represents a distributed array. It provides synchronization capacities on the entire array and enables optimized communications by allowing processes to fetch directly the parts they need from the processes that computed them. Distributed futures as a programming abstraction makes programming easier, in particular synchronization and data transfer; it also makes the communication between data-parallel entities more efficient than with standard futures. We implemented this notion in the context of BSP active objects that allows several BSP entities to interact in a task parallel and asynchronous manner. We showed the practical benefits of this approach.

As future work, pre-fetching strategies would allow an active object to trigger transfer between BSP processes earlier. Instead of pulling data when the BSP computation starts, by invocation of a `get_part` primitive, the idea is to push the data on the BSP processes while the request is in the input (FIFO) queue of the active object, i.e. between the moment the request is sent to the active object and the moment the request is handled by the active object.

REFERENCES

[1] Gul Agha. 1986. *Actors: a model of concurrent computation in distributed systems*. MIT Press.

[2] Keyvan Azadbakht, Nikolaos Bezirgiannis, and Frank de Boer. 2017. On futures for streaming data in abs. In *Lecture Notes in Computer Science/Lecture Notes in Artificial Intelligence*. 67–73.

[3] Olivier Ballereau, Frédéric Loulergue, and Gaétan Hains. 2000. High level BSP programming: BSML and BSlambda. In *Heriot-Watt University*. Intellect Books, 29–38.

[4] Frank De Boer, Vlad Serbanescu, Reiner Hähnle, Ludovic Henrio, Justine Rochas, Crystal Chang Din, Einar Broch Johnsen, Marjan Sirjani, Ehsan Khamespanah, Kiko Fernandez-Reyes, and Albert Mingkun Yang. 2017. A Survey of Active Object Languages. *ACM Comput. Surv.* 50 (2017), 76:1–76:39.

[5] Jan-Willem Buurlage, Tom Bannink, and Rob H. Bisseling. 2018. Bulk: A Modern C++ Interface for Bulk-Synchronous Parallel Programs. In *Euro-Par 2018: Parallel Processing*, Marco Aldinucci, Luca Padovani, and Massimo Torquati (Eds.). Springer International Publishing, 519–532.

[6] Albert Cohen, Léonard Gérard, and Marc Pouzet. 2012. Programming parallelism with futures in Lustre. In *ACM International Conference on Embedded Software (EMSOFT'12)*. ACM, Tampere, Finland. emsoft12.pdf Best paper award.

[7] Kiko Fernandez-Reyes, Dave Clarke, Ludovic Henrio, Einar Broch Johnsen, and Tobias Wrigstad. 2019. Godot: All the Benefits of Implicit and Explicit Futures. In *33rd European Conference on Object-Oriented Programming (ECOOP 2019) (Leibniz International Proceedings in Informatics (LIPIcs))*, Alastair F. Donaldson (Ed.), Vol. 134. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, Dagstuhl, Germany, 2:1–2:28.

[8] Kiko Fernandez-Reyes, Dave Clarke, and Daniel S. McCain. 2016. ParT: An Asynchronous Parallel Abstraction for Speculative Pipeline Computations. In *Coordination Models and Languages*. Springer International Publishing, 101–120.

[9] Gaétan Hains, Ludovic Henrio, Pierre Leca, and Wijnand Suijlen. 2018. Active Objects for Coordinating BSP Computations (Short Paper). In *Coordination Models and Languages*, Giovanna Di Marzo Serugendo and Michele Loreti (Eds.). Springer International Publishing, 220–230.

[10] J. M. D. Hill, W. F. McColl, D. C. Stefanescu, M. W. Goudreau, K. Lang, S. B. Rao, T. Suel, T. Tsantilas, and R. H. Bisseling. 1998. BSPlib: The BSP programming library. *Parallel Comput.* 24, 14 (1998).

[11] Noman Javed and Frédéric Loulergue. 2009. Osl: Optimized bulk synchronous parallel skeletons on distributed arrays. In *International Workshop on Advanced Parallel Processing Technologies*. Springer, 436–451.

[12] Kyle Singer, Yifan Xu, and I-Ting Angelina Lee. 2019. Proactive Work Stealing for Futures. In *Proceedings of the 24th Symposium on Principles and Practice of Parallel Programming (PPoPP '19)*. ACM, New York, NY, USA, 257–271.

[13] Leslie G Valiant. 1990. A bridging model for parallel computation. *CACM* 33, 8 (Aug 1990), 103.

[14] Derek Wyatt. 2013. *Akka Concurrency*. Artima.