UNIVERSITÉ CÔTE D'AZUR

ÉCOLE DOCTORALE STIC

SCIENCES ET TECHNOLOGIES DE L'INFORMATION ET DE LA COMMUNICATION

THÈSE

pour l'obtention du titre de

Docteur en Sciences

Mention Informatique

présentée et soutenue par

Pierre LECA

Combining active object and BSP programs

Thèse dirigée par Françoise BAUDE

Soutenue le T.B.D.

Jury

Rapporteurs	Gul AGHA	University of Illinois, Urbana – IL, USA
	Emmanuel CHAILLOUX	Sorbonne Université – Paris, France
Examinateurs	T.B.D.	
	T.B.D.	
	T.B.D.	
Directeur de thèse	Françoise BAUDE	Université Côte d'Azur, CNRS, I3S – Sophia Antipolis, France
Encadrants scientifiques	Ludovic HENRIO	Univ Lyon, EnsL, UCBL, CNRS, Inria, LIP – Lyon, France
	Wijnand SUIJLEN	Huawei Technologies – Paris, France
	Gaétan HAINS	Huawei Technologies – Paris, France

Résumés et mots clés

Combinaison de programmes à objets actifs et BSP

Résumé

Cette thèse présente un modèle de programmation hybride entre deux modèles de programmation parallèle : objets actifs et BSP (Bulk Synchronous Parallel). Les objets actifs sont spécialisés dans le parallélisme de tâche ; ils permettent d'exécuter du code fonctionellement différent en parallèle, et d'échanger leurs résultats grâce à des futurs qui représentent ces résultats avant-même qu'ils soient disponibles. Le modèle BSP permet, quant à lui, un parallélisme assez différent de celui des objets actifs : un parallélisme de donnée. Ce parallélisme consiste à découper une tâche en plusieurs morceaux et de les traiter en parallèle pour aller plus vite. Ces deux modèles spécialisés, permettent une programmation haut niveau et ont des propriétés intéressantes telle que la facilité de programmation et le déterminisme sous certaines conditions. L'intérêt d'allier ces deux modèles est donc de permettre l'écriture de programmes combinant parallélisme de tâche et parallélisme de donnée, tout en bénéficiant des caractéristiques des deux modèles. Cette thèse étudie ce nouveau modèle d'objets actifs BSP sous un aspect théorique (grâce à une sémantique opérationelle) et pratique (grâce à une implémentation en C++ avec MPI). Un nouveau concept de futurs distribués est également defini ; ils consistent à unifier les concepts de futurs et de vecteurs distribués. Cela permet une meilleure intégration entre objets actifs et BSP : grâce à eux, nos objets actifs BSP peuvent échanger entre eux efficacement en parallèle. L'efficacité de ces futurs distribués est mon-

trée grâce à des benchmarks sur notre implémentation qui comparent les performances des futurs classiques et des futurs distribués.

Mots clés : parallelisme, modèles de programmation, parallelisme de donnée, parallelisme de tache, objets actifs, acteurs, futurs, BSP, semantique opérationelle, benchmark

Combining active object and BSP programs

Abstract This thesis presents a hybrid programming model between two parallel programming models: active objects and BSP (Bulk Synchronous Parallel). Active objects are specialized in task parallelism; they enable the execution of functionally different codes in parallel and the exchange of their results thanks to futures, which represent these results before they are available. The BSP model enables a quite different parallelism from the one provided by active objects: data-parallelism. This form of parallelism consists of cutting a task into several pieces in order to process them faster in parallel. These two specialized models enable high-level programming and provide interesting properties such as ease of programming and determinism under certain conditions. The point of combining these two models is therefore to allow the writing of programs combining task-parallelism and data-parallelism, while benefiting from the properties of the two models. This thesis studies this new BSP active object model under a theoretical aspect (with operational semantics) and a practical aspect (with a C++/MPI implementation). We also introduce a new concept of distributed future. Our distributed futures consist in unifying the concepts of futures and distributed vectors in order to represent distributed data. This allows a better integration between active objects and BSP. With our distributed futures, our BSP active objects can communicate efficiently with each other in parallel. The efficiency of these distributed futures is shown through benchmark scenarios executed on our implementation. They allow us to confirm a performance improvement of our distributed futures against classical futures.

Keywords : parallelism, programming model, data-parallelism, task-parallelism, acive objects, actors, futures, BSP, operational semantics, benchmark

Contents

List of Figures

Chapter 1

Introduction

1.1 Context and Objectives

Programming parallel and distributed systems is a notoriously difficult discipline that is often reserved to experts which have to be aware and careful of the many pitfalls it involves. It is also a very time consuming endeavor that requires a great deal of debugging and performance analysis in order to reach one or several of the desired benefits it can bring. These benefits are attractive; for example, parallel programming may result in solving a task quicker, solving many tasks at the same time or solving a task with a larger, distributed input that would not fit in the memory of a single computer.

In order to ease parallel programming and thus enable a greater number of programmers to bring the benefits of parallel programming into their applications, research and industries have contributed to numerous parallel programming models, languages and frameworks over the years.

With languages becoming cleaner and less complex, formal studies were introduced in order to reason on them on the theoretical level. These works include the definition of formal languages (which often represent a language executable in a production environment). For example, operational semantics is a way to define a formal language, it enables the proof of certain properties within the defined formal language. This is valuable for a programmer as it adds to the safety,

reliability, and clarity of programming languages.

However, as programming models become higher-level and easier to use, they often sacrifice flexibility and hide parallel programming concepts in order to become easier. As we will see, BSP and actors are two examples of models that make programming easier in this manner. For example, it is difficult to efficiently synchronize parallel tasks of different durations with BSP. Making parallel programming easier and safer often comes at some cost: a loss of performance or flexibility for example. A side effect is often that programming models become specialized and less suited to solve problems outside their domains.

Domain Specific Languages (DSLs) are an extreme example of specialization that can make programming so easy that the developer of an application does not need to be a programming expert; but these kind of languages are so inflexible that they can not be used in another context than the one they are specialized for. DSLs can however be a good solution in some cases, but their lack of flexibility makes them unusable in the general case.

Even general purpose programming languages often propose a very specialized parallelization methodology which could be restrictive in terms of performance. They can also restrict the potential parallelism or the expressible distribution of the computation. This is why we believe it is interesting to propose hybrid parallel programming models. The benefits of doing so include broadening the class of application for the hybrid model while keeping the ease of programming thanks to the specific constructs of the different models.

Among the programming models for parallel and distributed computing, one can identify two important families. There is *task-parallelism*, which decomposes work into functionally different parts that can be executed in parallel, and *data-parallelism*, where a set of coordinated processes perform a computation by splitting the input data to process the different pieces in parallel.

Task and data-parallel programming languages remain far less specialized than DSLs, but their specialization often makes them inappropriate for other kind of parallelism. Because task-parallelism and data-parallelism are convenient to parallelize different parts of an application, it would be valuable to combine them into one programming framework.

The goal of this thesis is to study the integration of two programming models in these categories.

We want to propose a single model able to express both kinds of parallelisms, but also this new programming model should be easy to program and feature properties that provide some form of safety and limit the occurrence of bugs.

Active objects are specialized in task parallelism. They provide a friendly object oriented manner to program parallel tasks. Active objects basically represent entities running on dedicated threads; they communicate with each other through typed asynchronous method calls and they use futures to represent results. These futures are used as placeholders for the results of asynchronous invocations. Active objects have some valuable safety properties such as the absence of data-race, and a programmability close to sequential programming and determinism under certain conditions.

On the other hand the BSP parallel programming model, and its BSPlib API, provides data-parallelism, in a relatively easy to program way. BSP makes data-parallel programming easier and safer by concisely splitting algorithms into disjoint phases of computations and communications, and limiting the mean of synchronizing processes to barrier synchronizations. This provides BSP with some interesting properties such as determinism (depending on the implementation but easy to achieve), deadlock-freedom, and programmability close to sequential programming.

Both active objects and BSP limit the concurrency mechanisms exposed to the programmer. They provide a convenient and safe way to write parallel and distributed applications, with a programmability close to a sequential program. Because of the properties and focuses both these models have in common, we believe that the active object and BSP models are good candidates for forming an hybrid model between task and data-parallelisms.

There have already been attempts to mix task and data-parallelisms, for example into parallel component frameworks. However, we believe these two parallelism aspects could be better integrated than what can be found in the literature. In any case, even in the existing hybrid models that mix data and task parallelism, one known problem appears almost systematically: how to efficiently transfer the data produced in parallel by a task, to a different task. For solving this problem, communication strategies have to be designed. In these kind of scenarios, when the programming API has to be exposed as part of these strategies, the user has to be made aware of the existence of these strategies and how they work. Our goal is also to solve this problem in an efficient and

elegant manner in the context of the integration of BSP and active objects.

1.2 Contributions

The work of this thesis is centered around bringing task and data-parallelisms closer with each other, through a unified model between BSP and active objects.

The main contributions for this thesis are:

- **An hybrid BSP active object model**. This thesis is centered around this BSP active object model; we define a programming model which allows active objects to process a request in parallel through BSP.
- **A core formal BSP active object language and its operational semantics**. First, a formal language and clearly defined semantics precisely specify the behaviour of our model. Second, it provides the basis for a formal study of our model in order to prove theoretical properties.
- **The definition of distributed futures**. Adding data-parallelism inside active objects implies that they can produce distributed results in parallel. Providing such a functionality highlighted that futures are not appropriate for representing distributed data. In order to better integrate task and data-parallelisms, we designed distributed futures; this allows us to return distributed data from BSP active objects and communicate results of parallel tasks between active objects while optimizing the communications. While we designed distributed futures for our BSP active objects, our contribution is broader as distributed futures could be exported outside this context.
- **The design of a data transfer strategy for distributed futures**. We describe our strategy for synchronizing a distributed future and communicating the associated distributed data efficiently between parallel entities.
- **An API for creating, accessing, and manipulating distributed futures**. We extended our BSP active object model with primitives enabling the programmer to manipulate

distributed futures in our hybrid task and data-parallel environment.

- **The description of our BSP active objects and distributed futures implementations**. We implemented BSP active objects and distributed futures in a C++/MPI environment. We describe the interesting aspects of this implementation.

- **Benchmarks evaluating the performance of BSP active objects and distributed futures**. We evaluated our implementation of BSP active objects and distributed futures. We designed several benchmark scenarios to evaluate the performance of our BSP active object implementation. In our largest benchmark, we focus on the performance comparison of classical futures against distributed futures.

1.3 Overview

This document is organized as follows:

- In Chapter 2, we provide an overview of existing parallel programming models, with a focus on a task-parallel model: active objects, and a data-parallel model: BSP. All along, we point out existing works that attempts to bring the other kind of parallelism into each of them.

- Chapter 3 presents our BSP active object model.

- In Chapter 4, we present a formal language for BSP active objects.

- Chapter 5 presents distributed futures, an unification of futures and distributed vectors. We introduce distributed futures in the context of our BSP active objects.

- In Chapter 6, we describe in further details our implementation of BSP active objects and distributed futures.

- Chapter 7 provides an experimental evaluation of our implementation of BSP active object and distributed future.

- Chapter 8 summarizes the contributions of this thesis and gives concluding remarks on our work. Perspectives for future works are also presented.

Chapter 2

Background and State Of the Art

2.1 Introduction

In the context of this thesis, we are interested in two kinds of parallel programming approaches: task-parallelism and data-parallelism. More precisely, we are interested in a unifying model combining the two approaches; not as a general parallel programing model, but as a hybrid one. We picked a model of each category, for their properties that include safety and ease of programming, BSP in the domain of data-parallelism and active objects in the domain of task-parallelism. In this chapter, we give the state of the art on parallel programming in general, on the BSP and active object models, and on parallel data communication techniques, which are useful in a hybrid model that efficiently combines task and data parallelism.

This chapter is organized as follows: first we are going to give an overview of the state of parallel programming models and languages in Section 2.2. Then, we take a closer look at task-parallelism and data-parallelism by focusing on a family of each: BSP for data-parallelism in Section 2.3, and actors, active objects and futures for task-parallelism in Section 2.4. We then describe existing techniques for communicating data in parallel in Section 2.5.

2.2 Survey of existing parallel programming models, frameworks and languages

In this section, we discuss parallel programming in general, we start with low-level and well-established parallel programming approaches in Section 2.2.1. We then discuss in Section 2.2.2 why and how higher-level programming languages emerged, following with a brief survey of some of these high-level languages and frameworks for specialized parallel computing needs.

2.2.1 Well-established models for programming parallel computers

Several parallel APIs have been introduced to provide uniform interface to parallel programming. Indeed, parallel computers used to be specialized machines that each require their own programming library. The Message Passing Interface (MPI) is an attempt at providing a common API for programming parallel computers. Parallel computer providers can still provide dedicated libraries for programming their machines, but by providing an MPI implementation, that can be built from their specialized libraries, they enable programmers to build programs that are portable to other parallel computers.

With MPI, all processes start by entering the same function at the beginning of a program. This style of parallel programming is called Single Program Multiple Data (SPMD), because all processes execute the same program, usually from the same main function, with each process having its own control flow. This is in contrast with the Single Instruction Multiple Data (SIMD) style, which has only one control flow.

MPI provides message passing primitives in many variants, the basic ones are the blocking point-to-point message sending and receiving primitives, respectively `MPI_Send` and `MPI_Recv`. MPI also provides collective operations such as broadcast, gather and scatter, that each may perform on a group of processes that can be controlled by the programmer through a communicator concept. Inspired from other library interfaces such as BSPlib, later MPI versions added Remote Memory Access (RMA) primitives. While difficult to use and with many constraints, these RMA primitives allow the writing of data into registered remote memory areas. The remote processes are not aware

about exactly where data is read or written on their memory, but they still have to enter barrier-like primitives in order to allow these operations to happen.

One common source of bug with MPI is the misunderstanding and misuse of its primitives, possibly leading to deadlocks. According to the MPI specification, the sending primitive *may* block until the message is received by the target process, which must call the receiving primitive in order to do so. However, MPI implementations often optimize the sending of small messages by sending them right away in a non-blocking manner, which can be confusing for the programmer.

Calling the receiving primitive blocks until a matching message is received. While there are variants of sending and receiving primitives which may be mistaken to be asynchronous such as the `MPI_Irecv` and `MPI_Isend` primitives, they are in fact not guaranteed to be asynchronous by the MPI specification. Sending and receiving messages involves a lower-level protocol that may vary between different MPI implementations. The MPI standard recognizes this and chose to leave flexibility in the implementation of primitives for performance reasons. While some primitives may be mistaken to be asynchronous, other primitives must be called from time to time in order to make sure the data is being sent.

While MPI became a standard for programming parallel computers through the message passing model, shared memory computers can also be programmed through shared memory programming. OpenMP [22] later became a successful standard for programming shared memory computers. OpenMP took a higher level approach than MPI or other shared memory programming standards such as POSIX threads or Win32 threads. It consists of a set of compiler directives for C, C++ and Fortran. These directives can be used, among other things, to automatically parallelize loops. Shared and private variables are also specified within compiler directives. The way processes are used is also different from MPI. Here by default, the execution is sequential and the context becomes explicitly parallel when a parallel compiler directive is called, for the duration of the scope for which the directive is called. This follows the fork-join model of parallel programming; while MPI also provides means to program in a fork-join style, it is not as convenient to do so as with OpenMP.

While the C language is one of the preferred languages for writing high-performance applications, it does not natively integrate aspects of a parallel computer, or provide the means to program it.

While C libraries such as MPI C implementations provide means to program parallel computers, the Unified Parallel C (UPC) [17] extends the C language itself, which can then be implemented using, for instance, message passing or shared memory libraries. New keywords are added in order to distinguish between local and remote addresses and exploit data-locality. Pointers and arrays can be declared to be shared among parallel processes. The internal structure of shared variable includes a process number that indicates where the data is stored. The items of an array of shared data may or may not point to local data. Pointer arithmetic looks up element as cyclically distributed data. Block distribution is mentioned to be achievable either manually, by having a shared array of blocks of data, or as an UPC implementation choice. The memory consistency model may also be chosen by the user, with the choice between strict or relaxed models. Synchronization of processes is achieved through locks, fences and global barriers; the strict shared memory consistency model may also be exploited in order to synchronize processes.

2.2.2 High-level programming models

Because parallel programming, especially at lower level, is difficult and time consuming due to its increased complexity compared to sequential programming, numerous research works were dedicated to providing higher level languages and frameworks to simplify development work and make parallel programming accessible to programmers who are not parallel programming experts. Such work often comes with compromises on different levels, including performance and expressiveness, with goals varying from ease of programming, cost analysis, and verifiability.

Two common ways to parallelize an application are task parallelism and data parallelism. Task parallelism aims to decompose a problem into functionally different problems, where each is handled by a different process. Parallelism is achieved by having different problems solved at the same time. A pipeline can be formed by different processes, each handling a part of a problem instance. Data parallelism on the contrary aims to solve the same problem faster by slitting the input data and processing it in parallel on multiple processes. Sometimes, a data-parallel problem is solved without any communication. Such problems are called embarrassingly parallel problems, because of how easy their parallelization is.

One way to provide language support for task parallelism is Remote Procedure Calls (RPC), it enables sending a particular instruction to a process so that it executes a specified procedure with given parameters, all this with the language perspective of a local procedure call.

Actors, active objects and futures are concepts specialized to make task parallelism easier. On the other side Bulk Synchronous parallel is an execution model specialized into data-parallelism. Because of how close these concepts are to the work of this thesis, we will go into them into dedicated sections (Section 2.3 and Section 2.4).

The programming of stream applications is an example of task parallelism, where data flow through different components. StreamIt [63] is an example of a dedicated language for programming stream applications. It aims to ease stream programming, often programmed into low-level languages such as C, without sacrificing performance. In the original StreamIt from [63], all stream flows and filter combinations have to be known at compile time. This static structure aims to enable compile-time optimizations of streams, that had otherwise to be implemented manually by the stream application developer in a general purpose language. StreamIt uses a Java-like syntax to provide ease of programming. To provide a specialized language for streaming applications, StreamIt identifies main concepts to be integrated into the language: data stream, stream filters (computation item, or task), computation pattern (connection between filters).

Coordination languages [31] are also dedicated to task parallelism, they propose splitting programming languages into computation languages and coordination languages. The computation language part is dedicated to the specification of each task, while the coordination language is dedicated to making tasks interact with each other. Coordination languages are designed according to specific coordination needs. For example, distributed systems may make heavy use of RPC with request-reply mechanism, while parallel HPC applications can not afford to do so. An argument for the separation of coordination languages from computation languages is portability. For example, if for some reason the computation language has to change, then the coordination implementation of the different tasks can stay the same. Another argument made is that tasks made with different computation languages may be tied together in an application with a common coordination language.

The Linda [30] coordination language proposes an abstract concept of Global Tuple Space (GTS), which processes interact with in order to communicate with each other. Three base operations are described on the GTS: `out`, `in` and `read`. The `out` operation inserts a tuple of a given name along with parameters, the `in` operation reads a tuple from the GTS and removes it, while the `read` operation also reads a tuple without removing it. If an attempt is made at reading a tuple with a name that does not yet appear in the GTS, then the executing process waits for it. One feature is that the destination process is not explicitly specified when a tuple is sent using `out`, only a tuple name is given and any process can consume the tuple using `in` by specifying this name. Consequently, this set of primitives is a source of non-determinism as asynchronous processes may produce and consume tuples at different points in time. For example, when a tuple is produced and multiple processes are trying to consume this tuple by specifying its name, the processes trying to consume the tuple are actually competing with each other, because only one can obtain the tuple if it is removed from the GTS upon reading. These base operations are extended to support structured naming, allowing one name to generate a family of others. One main implementation challenge is to implement the GTS efficiently in a distributed environment. It is noted that the global aspect of the tuple space might put a heavy burden on the network, which might be a performance issue.

Tasks are sometimes encapsulated into higher level components that take care of their communicating interfaces, so that a programmer does not have to implement communication between different tasks. The Grid Component Model (GCM) [7] is a specification for implementing a component framework. It includes a way to manage components of the same kind as a single parallel component. This is useful for example for implementing data parallelism. Different tasks can be called for solving smaller problems. The result too can be viewed as a single distributed result, and passed between parallel components.

Foisy et al. studied how a get RPC call could send expressions to be evaluated by other processes [27]. The fact that processes are running the same code in parallel was exploited to allow an expression to be directly contained in a `get` call, as it would have been with a normal parameter. However the program structure is constrained as the `get` call is contained within a `sync` statement that freezes the evaluation environment so that it is not ambiguous at which point the value of the

expression has to be taken from the asynchronous processes.

Algorithmic skeletons were introduced by [21] as a high-level programming model. A skeleton aims to hide the complexity of parallelization through predefined patterns, that can be filled by the user. The user only has to program sequential aspects of the algorithms called muscles, and the difficult distributed/parallel aspects are provided by the skeleton. Different skeletons can be combined together by the user to form more complex patterns from the basic ones.

A successful example of a high-level programming framework inspired from skeletons is MapReduce [24]. MapReduce itself is merely a programming interface. It only exposes the user to two functions: `map` and `reduce`. MapReduce originates from Google where they realized that most of their computations apply a map operation to each data record to compute intermediate key-value pairs, and then applying reduce operations with all values which share the same key. Results are written to disk. Exposing only the map and reduce functions means that everything else can be programmed, optimized and made fault tolerant separately from the map and reduce performed by different applications. Portable code can also be written for different MapReduce implementations, where the user code (the map and reduce functions) can be reused in different implementations, for different architectures. One of the strong points mentioned of MapReduce is its ease of programming. Only two sequential functions have to be written by a user to write a parallel program without any knowledge of parallel programming required. Details about parallelization, fault tolerance, locality optimization and load balancing are hidden. While Google published the MapReduce concept, their implementation was kept proprietary, but open source implementations such as Hadoop exist.

Another successful high-level framework for parallel computing is Spark [69]. After each operation, MapReduce writes data to disk. This makes algorithms reusing results ineffective, in particular iterative jobs, because data has to be read and written from the disk every iteration. MapReduce applications have a rigid structure with sequence of map and reduce operations, writing to disk each time results are produced. Spark applications are more flexible in their structures as there is a driver program in charge of the control flow of the application, and with results that can stay in memory between operations. This driver program can call parallel operations that make use of distributed data called Resilient Distributed Dataset (RDD). An RDD is a read-only collection

partitioned among a set of machines. Parallel operations such as reduce, collect (send to driver), and foreach(apply user function) can be called to process the RDDs. Similarly to MapReduce, the Spark framework takes care of fault tolerance and communications, while hiding the most difficult aspects of parallel programming from the user.

As mentioned earlier among parallel programming models, this thesis rely on BSP and active objects. Because both active objects and BSP are specialized models, they are not appropriate for the kind of programs that the other is able to represent. In this thesis, we are interested in merging both into a single model in order to benefit from both task-parallel and data-parallel paradigms. We will talk about them in more details in Section 2.3 and Section 2.4. We will also see that a naive integration of both models can lead to communication patterns that could clearly be parallelized between two data parallel tasks, this is why we will describe some related works about communicating distributed data in Section 2.5.

2.3 Bulk Synchronous Parallel

In this section, we take a closer look at the BSP model of parallel programming. This model is interesting in the context of this thesis as it is a data-parallel specialized model of parallel programming. We will also see that, because the BSP model is not appropriate for task-parallelism, efforts were made in order to address this shortcoming, mainly through the proposed aspect of subset synchronization as we will later see. We picked BSP for our hybrid model between task and data parallelism because of the interesting properties we will detail in this section.

2.3.1 BSP computation model

The BSP model [66] originates from Valiant in the 90s as an attempt to provide parallel computing with an equivalent to the Von Neumann model for sequential computing. Valiant argues that what is needed for parallel computing to succeed in computationally intensive domains is what he calls a bridging model for parallel hardware and software. One main issue of parallel algorithm design was that different algorithms were required in order to solve the same problem efficiently on different

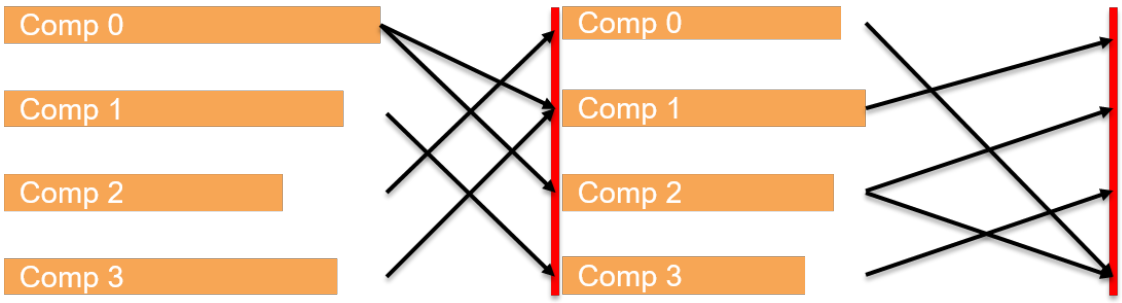


Figure 2.1 – The BSP execution model

networked parallel computers. The BSP model has the goal of splitting hardware and software concerns. It is hoped that having such a model generally accepted can lead parallel computing to lower costs and greater predictability, as software and hardware designers could keep their issues apart and focus on their actual domains. Software designers would be writing algorithms matching the BSP model and hardware/topology designers would implement the BSP abstract computer.

As shown in Figure 2.1, BSP algorithms are defined as a sequence of 3 phases. These phases are computation, communication and a global synchronization barrier between all processes (depicted by a red vertical bar on the figure). A group of these phases is called a superstep.

Formally splitting computation and communication into sequenced phases may not initially be intuitive as it implies forbidding their parallelization This choice is motivated by the fact that parallelizing them would at most improve performance by a factor of two if computation and communication steps were perfectly balanced so that their execution could perfectly overlap, while clearly separating them allows greater optimizations on communication, which we will see in Section 2.3.4.

McColl also studied and taught about the BSP model. His paper on scalable computing [53] provides a very nice overview of BSP, it represents today's general understanding about BSP. His lecture notes [52] provide a more detailed reading, from computer architecture to algorithm design.

Valiant's original hope for the BSP model is that it can be generally accepted by both the software and hardware communities so that experts can focus on their actual domains. However, while the forced barrier synchronization provides very nice properties, BSP is often criticized for only allowing this form of process synchronization. People see barrier synchronizations as slow, but they

often see the effect of communication or computation imbalance. The barrier itself can be efficiently implemented on many network topologies, often only requiring $\mathcal{O}(\log p)$ rounds of communication, which constitutes only a negligible amount in the total running time of many applications. In any non-embarrassingly parallel application, where it is required to exchange information among a group of processes, a collective communication round can be organized much more efficiently than if all processes have to synchronize and communicate using point-to-point communications with each of their peers without any sense of global schedule. Moreover, it makes accurate algorithmic cost analysis feasible, which is the main argument for always performing communications synchronously in the BSP model.

2.3.2 BSP abstract computer

A BSP computer is an abstraction which represents a parallel computer. It is very similar to the well-known distributed memory architecture, where each processor has its own memory and can communicate with other processors through a black box network. One assumption in a BSP computer is that the communication bottleneck is not in the network itself, but in the individual processes's network interfaces. A BSP computer is described by four performance indicators: its computation speed per processor s (e.g. in FLOPS), the number of processors p, the reciprocal communication throughput g and the latency L. Throughput and latency are expressed as a factor of the computation speed to indicate resource balance (How many computations are "spent" for sending each byte, or initiating communication).

The performance ratio of the components that make up a BSP computer, namely the processors with their memory and the network, can differ between systems. One system may have ten very fast processors (high s), slow synchronization (high L), but a high throughput network (low g), while the other system can have a thousand slow processors (low s), fast synchronization (low L) but a low throughput network (high g). These differing performance ratios may mean that the best BSP algorithm for, for example, a broadcast operation will also be different for these systems. The ten processor system in this example will be much better off with the single phase broadcast whose BSP cost is $T(n) = png + L$, while the thousand processor system will be likely much faster with the two

phase broadcast with cost $T(n) = 2ng + 2L$. To be able to decide more accurately, one can actually measure the BSP parameters, e.g. by using the benchmark program from BSPedupack [12], and apply those numbers to the cost formulas. The algorithm with the lowest BSP cost, will have a good chance of also being the fastest algorithm for that system.

The original BSP computer architecture is quite simple, with processor-memory pairs that communicate with each other through a shared network. However, it does not accurately represent modern computers. Valiant later proposed an alternative model [67] called Multi-BSP that could represent multi-core computers. This model can represent arbitrary combination of caches, memory, chips and processors. Instead of a flat p, g and L parameter set, a Multi-BSP computer has d levels of parameter sets, and it also captures the memory capacity m of each level. This gives $(p_1, g_1, L_1, m_1)(p_2, g_2, L_2, m_2)...(p_d, g_d, L_d, m_d)$ for a d layers Multi-BSP computer. This list represents a tree of components, where the components at each level are homogeneous, this is why this list is enough to represent a tree. Each level i has p_i components of the same kind, in other words, heterogeneous parallel computers can not be modeled. Each component of a layer forms a BSP computer with other components that have the same parent component. This BSP computer has bandwidth g_i, latency L_i and memory capacity m_i.

The representation of a parallel computer consisting of p Sun Niagara UltraSparc T1 multi-core chips connected to an external storage device is given by [67] in 2008. This parallel computer has p multi-core chips with external memory $m3$ accessible via a network of rate $g2$, where one chip has 8 core plus L2 cache, and each core has 1 processor with 4 threads plus L1 cache. With values taken from [67], this gives a Multi-BSP computer of ($p_1 = 4$, $g_1 = 1$, $L_1 = 3$, $m_1 = 8kB$)($p_2 = 8$, $g_2 = 3$, $L_2 = 23$, $m_2 = 3MB$)($p_3 = p$, $g_3 = \inf$, $L_3 = 108$,$m_3 \leq 128GB$).

After each component at a certain level finished doing its computation along with those sharing the same parent, they are able to write the result into their parent's memory. Through this model, Valiant hopes a more explicit control of caches can lead to more efficient synchronizations. This model is however broadly seen as overly complex.

2.3.3 BSP cost model

One of the goals of BSP is to make it easy to evaluate the cost of algorithms, this is why performance metrics are given to BSP computers. Because of the barriers, a BSP algorithm's cost can be decomposed as the sum of the costs of all supersteps. Each superstep's cost can then be calculated independently. Since the computations and communications are split, what is needed in order to evaluate the cost of a superstep is the cost of computations and the amount of data communicated. With the barrier, processes will be waiting for worst performing one at each phase. This inconvenience translates in simplicity for the cost model because it means the cost of each phase is simply the cost of the worst-performing process in this phase. For the computation phase, we directly take the worst case at any process of this computation. For the cost of the communication phase, we take the worst case sum of the amount of data sent and received by any process. This defines the BSP superstep as an h relation, with h being this sum. The cost of the communication (and synchronization) phase of an h-relation is then $h \cdot g + L$. Hence, the total cost of a superstep is $w + h \cdot g + L$, with w being the computational work required within this superstep; as usual, it is the worst case of any process. The total time of a BSP computation is the sum of its supersteps costs. Worst cases of h and w can be summed into respectively H and W to get the total cost within a simple formula $W + H \cdot g + S \cdot L$, with S being total number of supersteps.

2.3.4 Model variants and implementation optimizations

Skillicorn presented a listing of a few implementation optimizations that can apply [57]. Message packing is the most often quoted optimization when implementing BSP. Since the model suggests messages are going to be sent at the same time (at the end of a superstep), they can be packed into single messages when they have the same destination in order to avoid sending many small-sized messages on the network. Destination scheduling is also mentioned by Skillicorn, messages to different processes can be scheduled so that they have better chances of avoiding collisions when other processes are also transmitting to the same destination, for example using latin square techniques, which operate in sub-phases scheduled so that a process only interacts with one other

during the same sub-phase. Send rate pacing based on probed computer performances can also help to send just below the maximum throughput to avoid performance drops. Skillicorn also presented a few variants of broadcast implementations in message passing architecture, and how it is easy to use the BSP cost model to evaluate their performance. For example, the naive broadcast (called one-phase broadcast) involves a loop sending the data to each process. The BSP cost model gives this algorithm a communication time of $p \cdot n \cdot g + L$, where n is the size of the data, g is the BSP communication throughput, p is the number of processes and L is the latency cost of performing a superstep. The tree broadcast, implemented in many MPI implementations, involves every process sending the data to another process that does not have the data, at each superstep. This gives a cost for this algorithm of $log(p) \cdot n \cdot g + log(p) \cdot L$. Another broadcast technique relies on two supersteps, it is called the two-phase broadcast. In the first superstep, the owning process sends each process distinct n/p parts of the data. In the second superstep, every process broadcasts its part of the data to other processes, making maximum parallel usage of communication links, assumed to exist for every process pair. This implementation costs $2n \cdot g + 2L$. The two-phase broadcast is also implemented in many MPI implementations of the Broadcast, as a Scatter followed by an AllGather. According these theoretical costs, the one-phase broadcast is more efficient than the other algorithms in the situations of small data size, or when $p = 1$ or $p = 2$. The two-phase broadcast is better for large data volumes. however, the tree broadcast is never better than the others; it is only equivalent in performance to the two-phase broadcast when $p = 4$, but then performs worse.

Optimizations in the barrier itself can also be made, for example, if a first phase only triggers metadata exchange, different communication methods can be chosen according to this information. Skillicorn notes that optimizations of a BSP library are visible through improvement of a BSP computer's g and l parameters. This means improving the implementation of the library itself can, for example, yield a better g or L parameter for the BSP computer. Some optimizations are listed in [57]. For example, messages packing improves the bandwidth because more data can be sent in the same amount of time, destination scheduling and pacing decrease congestion because the sendings are coordinated and collisions are reduced, and barrier implementation optimizations

improve the l parameter directly because a faster barrier means a better latency.

One major controversy within the BSP community is whether to allow subset synchronization, this question and arguments of both sides are explored by [35]. Subset synchronization in BSP terms means removing the requirement of global barriers involving all existing processes and allowing barriers involving only a subset of all processes. Each subset can then be viewed as a distinct BSP machine with a different number of processes that communicate through barriers only including the processes of this subset. Two reasons are identified for wanting to use subset synchronization; firstly, for expressing higher-level expressions of computations, secondly, for allowing processes, of which the amount of work depends on the data itself, to proceed further without a barrier synchronization if they finished a phase earlier than other processes. Two objections are also given; firstly, subset synchronization increases the difficulty of cost analysis of BSP programs; secondly, there is no agreement on the semantics of BSP programs with subset synchronization. Divide and conquer algorithms are given as examples supporting the first reason for using subset synchronization; however, the author notes that balanced divide and conquer algorithms can be re-written using global barriers instead of subset barriers. The author also points out to balancing algorithms against the first reason. One of the major motivations of using BSP is its cost model. BSP assumes reliable measurement of the g and L values can be captured. Changing the number of processes p within a BSP machine is likely to also change its g and L parameters, which are associated with the new BSP computer corresponding to the subset. Some $O(p)$ BSP computer parameters sets could be taken for every possible number of p, but the author points out there is no proof that the communications of two interconnected BSP computers do not interfere with each other. Hence, the BSP parameters of two subsets would be unreliable, and with them the BSP cost model. This impact on the cost model is also mentioned in the BSPlib paper [39] as a reason for not providing subset synchronization features in the BSPlib API. BSP programming is often thought as writing a single program in a single BSP machine that communicates with a single global barrier. Rewriting programs is mentioned in [35] so that divide and conquer branches of a single algorithm are balanced and communicate with global barriers, but do not explore having functionally different tasks as motivation for allowing subset synchronization. A simple example is when a BSP algorithm solving

a single problem coexists with other algorithms in the same program. The difficulty of writing an algorithm with all the others in mind increases with the number of algorithms in the program. A single global barrier becomes more and more likely to slow down independent programs as they become stuck in barriers waiting for others processes that perform work that have nothing to do with theirs. In this case, subset synchronization is mentioned as a solution. Nevertheless, the BSP cost model is bound to become more difficult: it becomes more difficult to deduce the cost model from an application, and it becomes less reliable as network interferences between different subsets that do not coordinate their communications might exist.

Tiskin focused specifically on divide and conquer algorithms implemented in BSP without subset synchronization [65]. He proposes a concept of superthreads, that are SPMD-style threads of execution comprised of BSP supersteps. Several superthreads are orchestrated by a superthread manager. The idea is that the manager schedules each superthread for execution until it reaches a barrier synchronization, but the communications or the global synchronization is not yet executed. The next superthread is then scheduled in the same manner. Tiskin defines a superthread as an object with a `run()` method, where the superthread manager calls these methods and makes progress until they reach a barrier. The manager may even de-schedule it if it decides that the superthread takes too much time. Once every superthread reached a barrier, the superthread manager then triggers all superthread's requested communication and the global synchronization. The superthread style of programming enables viewing each superthread's code as independent of the others, while maintaining the flat BSP model without subset synchronization. The idea of a superthread manager being able to de-schedule superthreads without their consent however makes implementation difficult as Tiskin writes in [65]; the difficulty is assigned on the SPMD style of programming.

For a balanced parallel algorithm performing collective operations, a process getting delayed actually delays the entire execution as other processes will be waiting for the slowest process. Sparsely distributed OS routine execution can generate this kind of delay. This can be a problem for large scale executions. Jones proposed scheduling together those OS routines to reduce this global delay [42]. Each process does not risk delaying other processes because all of them will be

executing their OS routines at the same time.

Kim et al. [47] studied an alternative where the global barrier synchronization is removed and only the necessary exchanges are made. This change is introduced as a relaxed barrier synchronization. While experimental results are presented as outperforming a standard BSP implementation and maintaining its consistency, it can be argued that adopting such ideas may remove possibilities for other optimizations. For example, the relaxed barrier optimization requires all inter-process exchanges to be made with a PUT operation, which is a very small subset of the core primitives presented by Hill et al. [39], which also includes GET, SEND and RECV primitives. While one of such optimization may outperform any combination of others, the fact that researcher's intuitions are looking for different ways to optimize the BSP model while changing its core or its interfaces may imply that it is still not generally accepted as a standard for BSP programming.

2.3.5 BSP languages and applications

The BSP model itself does not specify how parallel applications are to be programmed, it only describes how a BSP computation proceeds on a BSP computer. The choice is left open as to how BSP computations are to be programmed, with the possibility of having both high level languages and low level ones. For example, Valiant suggests [66] automating memory and communication management through hashing for high level languages. Lower level languages are better able to exploit the bandwidth parameter g as they can minimize the number of supersteps whereas higher level languages are not always able to produce a BSP execution that minimizes the number of supersteps.

This means lower level, direct BSP programming, is better suited for execution on high bandwidth BSP computer, because a lower-level program might be better able to produce a code requiring fewer barrier synchronization, hence the performance of this program would be less dependent on the latency parameter L.

It is even worth considering the BSP computer parameters at runtime to decide which algorithm would run faster on which BSP computer. For example, an algorithm that communicates less data but in more supersteps is better suited for a low latency BSP computer, whereas an algorithm that

communicates more but in less supersteps is better suited for a high bandwidth BSP computer.

Multiple languages and environments have been built for writing BSP programs. First, Hill et al. proposed a library interface for the C language called BSPlib [39], which they implemented in the Oxford BSP toolset. BSPlib will be explained in more details in Section 2.3.6.

When BSPlib was presented as a C library interface implementing the BSP model [39], practical example implementations were also presented along with benchmarks. These applications included Fast Fourier Transform, Randomized sample sort and the Barnes-Hut N-Body Algorithm.

Nibhanupudi et al. [54] used BSP for plasma simulation on networks of workstation. This work also compared performances with MPI, the results are presented as comparable for both libraries with the algorithm they used.

An attempt was made to use the BSP model for high resolution visualization [14], the initial goal was to implement a visualization program as a BSP one for plotting in parallel. But point-to-point communication is not required for this usecase, so supersteps are implicit and serve as synchronizations for orchestrating plotting functions.

Works around BSPlib

Several implementations and variants of BSPlib exist. Yzelman implemented BSPlib for shared memory computers [68], his library includes a `bsp_direct_get()` instruction which allows direct access to shared memory. This saves a global synchronization for reading data from remote memory, but it effectively bypasses the superstep concept, and the programmer has to take care of avoiding concurrent reads/writes to an accessed memory area, this loses the deterministic property that can be attained in particular implementations, in exchange for superstep-saving accesses to shared memory. Yzelman also proposes nesting calls to the BSPlib `bsp_begin()` primitive, allowing nested BSP computations with their own independent context, effectively supporting hierarchical execution of different groups of processes, each owning a subset of processes. When the processes are split into subsets, synchronization is not global. This means subset synchronization is encouraged as a hierarchy of processes. This is a bit more constrained than generic subset synchronization. Because each new group has its own independent context and pauses the execution of the parent process,

overlapping subsets are forbidden.

The Paderborn University BSP library also presented an implementation [15], which has the particularity to allow subset synchronization. Their main motivation in allowing subset synchronization is for different completely sub-algorithms with different communication patterns to be run in parallel; they cite a parallel ocean simulation program as example. They also argue that synchronizing fewer processes is faster than synchronizing all of them. However they do not try to explain how this impacts the BSP performance prediction model. As an optimization to implementing the barrier synchronization, they provide an oblivious synchronization primitive that saves a metadata communication round when it is known how many messages each process is going to receive. The authors mention interleaving as another option for implementing this kind of algorithms, but regard it as inefficient because of different synchronization requirements, they also consider it as more difficult to implement. The interleaving approach was explored by Tiskin [65] for writing divide and conquer algorithms, but not for writing functionally different algorithms. The PUB library also implemented collective primitives such as broadcast, reduce and scan, on the top of the architecture directly instead of building them from core primitives.

There has been work on providing formal semantics of BSPlib. The DRMA (Direct Remote Memory Access) part of BSPlib, excluding the BSMP (Bulk Synchronous Message Passing) part, was formalized by [62]. The authors propose a BSP-IMP formal language which has the put and get DRMA operations, without including the register allocation primitives for simplicity. Local reduction rules evaluate local statements of individual processes while global rules orchestrate evaluation of local rules on parallel processes. Global rules have visibility over all process states, one of these rules is able to detect termination of a program when all processes terminated executing their local operations; but more interestingly, they are able to detect synchronization errors when a process is in finished state while another is waiting at a barrier (meaning there is a deadlock). This error detection can be implemented in BSPlib relatively easily compared to general message passing languages. Overall, we can say that there are no deadlocks in BSPlib because a deadlock is easy to detect at runtime and then an error can be raised. The global rule for processing the global synchronization, triggered after each process reached a sync operation, employs a black-box oper-

ation for exchanging data between processes. While the authors do not define this sync operation, they describe three different ways it could be done;

1. **BSPlib compliant**: communication requests are processed in an unordered manner, as specified by the BSPlib specification [39]. For example, the result of two write operations by different processes on the same remote memory area is unspecified.

2. **Determinized**: communication requests are processed in an ordered, deterministic manner, with different priorities given to each process when communication requests are processed.

3. **Debug mode**: an error is raised when two communication requests access the same remote memory area in the same superstep.

This discussion is interesting, as by showing these different ways to define the synchronization rule in a BSPlib semantics, it is shown that the same variations exist in different BSPlib implementations, depending on the desired focus: performance, determinism, or error detection.

A simplified subset of BSPlib was formalized by [28] with a big-step semantics. The paper targets formalizing the PUB library but the original features of this BSPlib variant such as subset synchronization are not represented in the BSP formal language of this paper. This means that their formalization is closer to the original BSPlib specification than the PUB library, and therefore this formalization should be considered more general than the PUB library. DRMA communication aspects are modeled through sync, push, pop, put and get operations, while message passing are modeled through a send operation and a `findmsg` operation that reads from a special set of received variables. For simplicity, aspects of C such as pointers are not modeled and are instead replaced by variable names. As in [62], there are local rules for local operations and global rules for orchestrating operations. This formal semantics was implemented in the Coq proof assistant the author's semantics was used to solve the N-Body problem and prove its correctness; however, hundreds of applications of Coq's tactics are needed, which makes the proof of this program very tedious.

Another implementation of BSPlib by Suijlen called BSPonMPI was written in 2006, then rewritten in 2019 [59]. The latter implementation tackles the subset synchronization controversy

with a concept of delayed one-sided communication requests, illustrated by implementing bulk synchronous collective communication primitives. For example, it is known that a naive BSP broadcast has a BSP cost of $p.n.g+L$, while a two-phase broadcast has a cost of $2.n.g+2L$. While the naive broadcast is more efficient for smaller data because time is mostly spent on latency, larger data benefit from a two-phase broadcast. A problem arises when one tries to build a broadcast function; because a two-phase broadcast requires two supersteps (hence two BSPlib's `bsp_sync` calls), the calls to sync have to be inside the wrapping function to trigger a barrier. The problem is when two independent consecutive calls are made to this broadcast function: a total of four supersteps are required because two calls trigger two synchronizations each. The Bulk Synchronous Collectives component of BSPonMPI proposes writing this broadcast function with delayed communication primitives so that communications are requested to happen after a specified superstep number, and synchronizations are only requested to happen before processing the following communications. A new broadcast primitive is provided along with other common collectives that make use of these delayed communication primitives to avoid triggering a real synchronization within a function call requiring multiple supersteps, but instead returns a superstep number pointing to the future where the collective will have performed all its action, it is then possible to write a sequence of collective calls where only the minimal number of supersteps is required, with the user triggering the barrier synchronization after knowing when all calls will have performed their intended actions. This delayed communication concept more generally enables parallel composition of different BSP functions in a different way than subset synchronization.

Inspired from BSPlib variants of the time, Suijlen and Yzelman made another direct BSP implementation called Lightweight Parallel Foundation (LPF) [60]. The programming style is similar to BSPlib's DRMA style, with `lpf_put` and `lpf_get` primitives as main communication primitives. They focus on BSP model compliance and performance prediction by providing asymptotic complexity guarantees to their primitives. Memory usage from the library is exposed to the user with primitives controlling internal memory usage. Another focus of this library is interoperability. For this purpose, it is possible to create an LPF context from another parallel environment by calling an `lpf_hook` primitive, thus allowing the use of the LPF library within different parallel frame-

works to solve sub-problems. An example of calling LPF to solve the PageRank algorithm from Spark is given. Interoperability has always been an issue with BSPlib, because BSPlib programs are required to start the execution environment at the beginning of a program, and also to end it, by terminating the parallel execution environment, making co-existence with other frameworks difficult. The LPF approach address this problem by making it possible to call a BSP program inside existing frameworks in order to solve sub-problems.

Functional BSP programming

Loulergue et al. introduced $BS\lambda$ [50], a calculus for functional programming languages extending the $\lambda - calculus$ to write BSP algorithms. They also proved the flat version of $BS\lambda$ to be confluent(the result is the same no matter in which order applicable rules are applied), they introduced a parallel composition operation, but proved that it breaks the confluence property of flat $BS\lambda$.

A functional language called BSML [6] was implemented after $BS\lambda$. BSML was implemented as an interface between OCaml and BSPlib. The syntax of BSML was later revised [16] because experience with users showed that a communication primitive is difficult to use.

Based on the $BS\lambda$ calculus, Gava formalized BSMLib (an implementation of BSML in Objective Caml) in the Coq proof assistant [29]. Using this formalization in Coq, parallel operations were certified. These operations include the two-phase broadcast, scatter and gather operations.

Higher-level BSP languages and Frameworks

Kessler implemented the BSP model as a dedicated language called NestStep [44]. NestStep comes with the concept of virtual shared memory, where one can declare shared arrays that can be either replicated, distributed, or volatile. It does not mean that processes will use shared memory to store the data, but it gives a shared memory view to the programmer, who can arbitrarily access any part of the array. This shared memory view respects the BSP model as accesses to data will only be performed after barrier synchronizations. For distributed or volatile data, it is respectively distributed among all the processes' distributed memory (blockwise or circularly), or assigned to an arbitrary process, but each process internally knows where the data is, even if the programmer

doesn't. Kessler also argues for subset synchronization and implemented it in NestStep. He views global barrier synchronization as an inflexible mechanism for structuring parallel programs which is also less effective than synchronizing a subset of processes when p is large. He points that subset synchronization allows overlapping communication and computation when different process subsets are in different BSP phases. Data accesses are combined at the barrier to reduce the number of produced messages. Kessler later proposed automatically prefetching data from read operations at the beginning of the previous superstep [45].

The Orleans Skeleton Library (OSL) [40] proposes building a data-parallel skeleton library following the BSP model. The computation skeletons of OSL manipulate distributed arrays objects to identify and group array parts stored by each process so that they represent a distributed array for the programmer. Local computations such as `map` and `zip` can be used to apply user functions to elements of distributed arrays. The communication skeletons of OSL (`shift_left`, `short_right` and `permute_partition`) allow transferring distributed array parts between different processes, with communications that follow the BSP model.

As a model for parallel computation, obvious practical applications for BSP are general scientific computing problems such as matrix multiplications, FFT, and dynamics simulations. However, the BSP model has also been used by Malewicz et al. for graph processing at Google [51]. Their paper presents Pregel, a specialized framework for large scale graph processing based on the BSP model. Besides showing how BSP was more appropriate in this context than industry-standard MapReduce, it exploits the BSP barrier property to introduce fault tolerance by checkpointing on supersteps. Confined recovery is also suggested for improving the cost and latency of the recoveries. If messages are logged, it is possible to avoid some computation by using logged messages to recover, instead of asking for all processes to recompute everything even though we know what messages will be produced by healthy processes. A way to virtually exploit the locality of VMs on the same hosts is also mentioned. If functionally coherent, processes on such VMs have the possibility to combine messages to be sent to the same recipient, similarly to the previously described message packing technique but with a functional merging of messages. Performances is then not only likely to depend on the graph partition strategy that is used, but also on the partition placement on VMs

and hosts. These can be specified in Pregel, even though the benchmarks that were performed are using a simple hash function to do so.

Starting with Pregel, BSP has generated considerable interest in the domain of graph computing. This is due to the inefficiency of standard MapReduce frameworks for implementing iterative algorithms, like most graph algorithms. Algorithms are expressed from a vertex's point of view, hence this kind of programming called "Think Like A Vertex", or TLAV. Other graph computing frameworks based on BSP have been developed since Pregel, Apache Hama [56] and Apache Giraph [2] are among those. Hama is a direct BSP implementation and it does not only cover graph computing. Giraph is vertex-centric and was extended by Facebook for running graph algorithms on a trillion edges graph [20]. When evaluating graph computing platforms, their choice was strongly influenced by the BSP model. They cite its deterministic property (that eases debugging), ease to understand, and straightforward scaling. Weaknesses of TLAV have been underscored by Tian et al. [64], who showed how the vertex-centric abstraction prevents making use of the locality between different vertexes on the same partition. They proposed instead to have a graph-centric model to process by partition instead of vertex to reduce the number of supersteps, therefore reducing the communication costs. This paper also points that sequential programs are often graph-centric; sequential programs are thus easier to adapt into a graph-centric framework, even if the framework works on partitions instead of the whole graph.

2.3.6 A focus on BSPlib

In this section, we go into further detail about BSPlib, a standard programming library for programming BSP algorithms. First, we will take a look at some core primitives, then we will show a code example making use of this library.

BSPlib API

The original BSPlib specification from Hill [39] is a direct low-level BSP implementation in C where data exchanges are programmed explicitly. Just as with MPI, every process executes its own version of an SPMD program and is able to communicate with other processes. BSPlib proposes a set of

20 core primitives, organized into three different categories.

The first category is called "SPMD category" and contains primitives that control parallel execution. The `bsp_init` has to be called at the beginning of a program, with a `void(*spmd)()` function as parameter that will be the entry point of all parallel processes. Then a `bsp_begin` function has to be called with a number of required processes as parameter. A symmetrical `bsp_end` function must be called after the parallel execution. These control functions must be called only once in a program's lifetime. This imposed program structure is one of the reasons why the original BSPlib specification is impractical to use. Indeed, an existing parallel program cannot use the BSPlib library to solve a sub-task as the aforementioned primitives delimit the beginning and the end of a program. program mus calling a BSPlob from from within a different is difficult A process identifier (pid) can be queried with the `bsp_pid` function, as well as the total number of process with the `bsp_nproc` function. One of the most important primitives of BSPlib is the `bsp_sync` function, which delimits a BSP barrier. All requested communications are only effective after a call to this primitive by all processes.

The second and third categories of BSPlib are describing communication primitives for two kinds of programming: DRMA (Direct Remote Memory Access), and BSMP (Bulk Synchronous Message Passing). DRMA programming consists of reading/writing data directly from/into a remote process's memory through `bsp_get`/`bsp_put` primitives. Remote memory that can be accessed has to be registered with a `bsp_push_reg` primitive; it can also be de-registered with a `bsp_pop_reg` primitive. Message passing consists of explicit message exchanges between processes: a sent message has to be explicitly received by a process so that its content may be used by the programmer. The BSPM category allows messages to be sent using the `bsp_send` primitive, and received by the destination with a `bsp_move` primitive. A tag is also attached to each message and can be introspected before dequeuing a message, but contrarily to MPI, messages cannot be selected according to their tag, the message at the start of the queue has to be dequeued no matter its tag. The `bsp_get_tag` function allows reading the tag of the next message in the queue without dequeuing this message.

2.3.7 BSPlib Example

Here, we show an example program from BSPedupack [12], a software package written by Bisseling that uses the BSPlib API. This example shows a simple BSPlib program that computes the inner product of two vectors; it is split into three figures for size concern (Figure 2.2, Figure 2.3 and Figure 2.4). These split figures form, in the same order, a single file taken from the BSPedupack. Figure 2.2 shows the core of this example: the `bspip` function that computes the inner product of its input vectors. Figure 2.3 shows a function called to initialize an input, and to call the `bsp_begin` and `bsp_end` functions. Figure 2.4 shows the `main` function of this program, which illustrates the use of the `bsp_init` function.

The `main` function in Figure 2.4 is entered in parallel by every process in the SPMD parallel environment. This function first calls `bsp_init` (line 3) with the function `bspinprod` as parameter to instruct all processes, except the process of pid 0, to enter this function; from this point, only the process of pid 0 proceeds in the `main` function. This process asks the user to input a number of processes to use for executing the rest of the program; this input is put into the variable `P` (at the global scope), declared above in the original code (here it is in Figure 2.2). After a simple check on this number to make sure enough processes are available, the `bspinprod` function is then called (line 14) so that the process of pid 0 joins the processes that entered this function before through `bsp_init`.

The `bspinprod` function in Figure 2.3, starts by executing `bsp_begin` at line 8, requesting the use of the number of processes that was requested (that is now in the `P` global variable) in the `main` function. Most of the rest of this function is about initializing the input for the example, we will show the use of the communication primitives in a more interesting and concise setting in Figure 2.2. Here we can note that the `bspip` is called in parallel, and the expected result will be set into variable `alpha` of every process. After each process prints this result at lines (37-38), the `bsp_end` function is called to mark the end of the parallel execution of the BSPlib program. Only the process of pid 0 returns to the `main` function before calling `exit`.

We now take a look at the core of this example, the `bspip` function of Figure 2.2. This function

```
#include "bspedupack.h"

/*  This program computes the sum of the first n squares, for n>=0,
sum = 1*1 + 2*2 + ... + n*n
by computing the inner product of x=(1,2,...,n)^T and itself.
The output should equal n*(n+1)*(2n+1)/6.
The distribution of x is cyclic.
*/

int P; /* number of processors requested */

int nloc(int p, int s, int n){
    /* Compute number of local components of processor s for vector
    of length n distributed cyclically over p processors. */

    return  (n+p-s-1)/p ;

} /* end nloc */

double bspip(int p, int s, int n, double *x, double *y){
    /* Compute inner product of vectors x and y of length n>=0 */

    int nloc(int p, int s, int n);
    double inprod, *Inprod, alpha;
    int i, t;

    Inprod= vecallocd(p); bsp_push_reg(Inprod,p*SZDBL);
    bsp_sync();

    inprod= 0.0;
    for (i=0; i<nloc(p,s,n); i++){
        inprod += x[i]*y[i];
    }
    for (t=0; t<p; t++){
    // Write the double value inprod (of size SZDBL) at position s
    // of the registered memory area Inprod of process t
        bsp_put(t,&inprod,Inprod,s*SZDBL,SZDBL);
    }
    bsp_sync();

    alpha= 0.0;
    for (t=0; t<p; t++){
        alpha += Inprod[t];
    }
    bsp_pop_reg(Inprod); vecfreed(Inprod);

    return alpha;

} /* end bspip */
```

Figure 2.2 – BSPlib inner product from BSPedupack

```
void bspinprod(){

    double bspip(int p, int s, int n, double *x, double *y);
    int nloc(int p, int s, int n);
    double *x, alpha, time0, time1;
    int p, s, n, nl, i, iglob;

    bsp_begin(P);
    p= bsp_nprocs(); /* p = number of processors obtained */
    s= bsp_pid();    /* s = processor number */
    if (s==0){
        printf("Please enter n:\n"); fflush(stdout);
        scanf("%d",&n);
        if(n<0)
        bsp_abort("Error in input: n is negative");
    }
    bsp_push_reg(&n,SZINT);
    bsp_sync();

    bsp_get(0,&n,0,&n,SZINT);
    bsp_sync();
    bsp_pop_reg(&n);

    nl= nloc(p,s,n);
    x= vecallocd(nl);
    for (i=0; i<nl; i++){
        iglob= i*p+s;
        x[i]= iglob+1;
    }
    bsp_sync();
    time0=bsp_time();

    alpha= bspip(p,s,n,x,x);
    bsp_sync();
    time1=bsp_time();

    printf("Processor %d: sum of squares up to %d*%d is %.lf\n",
    s,n,n,alpha); fflush(stdout);
    if (s==0){
        printf("This took only %.6lf seconds.\n", time1-time0);
        fflush(stdout);
    }

    vecfreed(x);
    bsp_end();

} /* end bspinprod */
```

Figure 2.3 – initializing function of BSPedupack inner product

```
1  int main(int argc, char **argv){
2
3      bsp_init(bspinprod, argc, argv);
4
5      /* sequential part */
6      printf("How many processors do you want to use?\n"); fflush(stdout);
7      scanf("%d",&P);
8      if (P > bsp_nprocs()){
9          printf("Sorry, not enough processors available.\n"); fflush(stdout);
10         exit(1);
11     }
12
13     /* SPMD part */
14     bspinprod();
15
16     /* sequential part */
17     exit(0);
18
19 } /* end main */
```

Figure 2.4 – main function of BSPedupack inner product

is entered in parallel by the number of processes (now in `p`) specified by the user; it computes in parallel the inner product of vectors `x` and `y` of sizes `n`, using `p` processes. The `bspinprod` function (calling `bspip`) of Figure 2.3 made every process allocate (line 25) and initialize (line 28) the vector parts so that they are already distributed, each process owns a part. We can see that first, a vector `Inprod` is allocated (line 27) by every process (every process has its own copy); this vector is later used for storing intermediate results, with one slot for each process. This vector is then registered using `bsp_push_reg` for enabling DRMA communications. Then processes start computing the local inner products of the vector parts they hold; after each process s has this intermediate result, it communicates it to every process `t` (ranging from 0 to `p`) using the `bsp_put` primitive (line 37) into the slot s of the intermediate result vector that was previously registered. As we have explained before, the BSPlib communication primitives only initiate communications, they are only effective after a call to `bsp_sync`, which is made right after the calls to `bsp_put`. After every process has the intermediate results of every process, only a sum of these is necessary to get the inner product result of the whole distributed vectors; this result is written into the `alpha` variable. As every process does this, the result is available on the `alpha` variable of all these processes (each process knows the final result of the inner product). Before returning the result, the registered memory area is

removed using the `bsp_pop_reg` primitive (line 43). After the next superstep, this memory area will no longer be a valid target for DRMA operations. The result is returned at line 45 by every process, as the `bspip` function was called in parallel by every process from the `bspinprod` function at line 33 (`alpha= bspip(p,s,n,x,x);`) of Figure 2.3. The result is then printed by every process as explained above.

2.4 Futures, Promises, Actors and Active Objects

In this section, we give an introduction to the concepts of *Futures*, *Promises*, *Actors*, and *Active Objects*. These concepts are dedicated to task-parallelism, and we will see their most valuable properties.

We picked active objects and futures for our hybrid model between task and data parallelism because of their interesting properties concerning safety and absence of data-races. The concepts of futures, promises, actors and active objects are not only strongly tied to task-parallelism, but also strongly tied together, which is why we introduce them together in this section.

We start by defining futures and promises in Section 2.4.1, before defining actors and active objects in Section 2.4.2 because the actors and active objects use the concepts futures and promises. We describe a few languages and implementations of actor and active object languages in Section 2.4.3, before giving a few examples of works that bring a data-parallelism flavor inside actors and active objects. We then describe a few applications that use actor and active object languages in practice.

2.4.1 Futures and promises

The definition of a future is somewhat varying for developers. We will start this section with a brief history of futures and promises, followed with the definitions we will use for the rest of this thesis. We will then give a brief survey on futures.

Futures were first introduced in 1985 by Halstead and Robert [36] and they are often confused with promises, later described by Liskov and Shrira in 1988 [49]. The futures of [36], were introduced

into a parallel functional language, based on the Scheme language, called Multilisp. A future is basically a placeholder for the result of a task that that runs in parallel. In Multilisp, the construct `future X` immediately returns a future and executes expression `X` in parallel as a separate task. The future is initially undetermined, and becomes determined when the `X` expression finished being computed.

When comparing both concepts, Liskov and Shrira distinguish them by saying promises have a strong typing and allow better exception handling. This comparison is quite old and implementations vary, with strong typing and exception handling for futures of nowadays too. The two concepts are in practice similar, but we will see below a distinction that appeared later on between the two concepts: read or write access modes.

Liskov and Shira introduced Promises [49] and focus on comparing their use to Remote Procedure Calls (RPC). They note that RPCs callers have to wait for the result to be computed and returned before continuing, and that they could instead proceed doing other things before trying to synchronize the result when they need it. Within their language, they also note that arguments are passed by value and that promises are not valid arguments. This effectively means their implementation of promises have to synchronize the results before passing them to other processes, even if this result is not required in the calling process.

Futures and promises both have an associated state; the first future version [36] defines it as determined and undetermined, the first promise version [49] defines a promise as a 2 states future : blocked or ready. Another version [11] features promises as 3 states futures : initially unresolved, fulfilled or broken.

In some implementations, a Promise object is given as parameter to the newly created task. While a future is fulfilled when a task finishes, a promise can be used explicitly to be fulfilled. For example, the Scala language allows a called process to use a Promise object to give a value to a future, in contrast with the future being completed by the return statement of the called function. Because the future object is used to read a value, and the promise object is used to write it, a future can be seen as a read-only view to its bound promise object. Giving a value to a future or a promise is called fulfilling the future (resp. promise).

This is the definition of futures and promises we assume for the rest of this document. A future is thus what is returned from an active object call, and there are two ways to fulfill this future: either with a return value from the called function, or through promise object given to the called function that allows fulfilling the promise by setting its value. In the works of this thesis, we will only use futures that are fulfilled by returning a value.

An efficient use of a future variable is to perform as many computations as possible before trying to access the future variable. This way the caller does not have to wait for the result of the call associated to the future, as this computation will be executed in background in a separate thread.

Some languages allow passing futures as argument of remote calls without trying to resolve them. These futures are called *first class futures*.

The moment when a future is retrieved impacts performance; while most local implementations simply check the availability of some data in the memory, the access to future values is less obvious in a distributed setting, especially if future references can be transmitted between processes (first class futures) and thus many processes may need to access the same future value. An overview of some of the possible future update strategies is given by [46], identifying three main strategies: "eager forward", "eager message", and "lazy message". In eager forward every process that sends a future object as a call argument is responsible for forwarding its value to processes they sent the future to, when this value is available. All futures eventually become updated as values are recursively forwarded along the call graph after this value is computed; in eager message, the process computing the future is responsible for sending the value back to every process that owns a future reference, which means it has to be notified each time this future is passed as argument; in lazy message, the process computing a value will store it after it is produced, and answer this value when it is requested by another process trying to access it through a corresponding future object.

While synchronizing the data associated to a future in one aspect of the future concept, accessing this future in the implementation language is a separate aspect. Depending on the implementation, there can be different ways to access a future. A *get* operation on a future blocks until the future is resolved, or simply continues with the value if it is already available. The execution thread can also be released with an *await* operation, until the future value is available, but it has some overhead

for saving and retrieving the execution context. Another way to access a future is to not explicitly access it, but to register a continuation to be executed when the future is resolved. Synchronization with a get operation is more efficient and lightweight, whereas synchronization with await is more difficult to implement (because it requires cooperative thread scheduling). The great advantage of cooperative scheduling is that it removes some deadlocks, but it requires additional care with the state of variables between context switches. Synchronization with registration of continuation loses the variable context because the called function has a different scope but allows an execution thread to be used only when it is possible to proceed with a computation.

When a future represents large data that are then processed by the way of an iteration, it is possible to overlap the transfer of this future with the processing of parts being received, as shown in [8]. In their implementation within the C++// language, a *later* object is introduced, which behaves the same as a future, but is not sent during the first inspection of the object when it is passed as a call parameter, but later. For example, a list of *later* objects can be given as parameter, and the later objects are fulfilled independently. Consequently, a request is able to start without all the data pieces that are defined as parameters, but can proceed by requiring smaller pieces separately as it goes on.

Futures can also be used to represent streams of result data. Within the context of the Abstract Behavioral Specification (ABS), Azadbakht and De Boer introduced streaming futures [5], which are futures that can be fulfilled multiple times in a queue-like manner. Inside a method, a `yield` statement can be used, instead of a `return` statement, to fulfill a future without exiting the function. A future can then be fulfilled multiple times, and especially, accessing the future multiple times gives different values, in the order of which they were fulfilled. This gives the rise of non determinism (race conditions) as a future shared between concurrent actors can be accessed in an unspecified order, with different results depending on this order.

When using promises to fulfill a future, there is added flexibility because the promise can be fulfilled at any given time by any process, it is also possible to continue executing a method after a promise has been fulfilled. But one problematic case is that a programmer can forget to fulfill a promise, or fulfill it several times. This is not a possibility when the future is managed by a return

statement because most compilers will notice when a return statement is missing and a return terminates the function. But one issue with futures managed by return value is that the return value type is inflexible when futures are explicitly typed. Let us take this example: a method `M1` has an `int` as return type, but it requires the execution of another method `M2` to compute this `int` (in an asychronous way); thus, it would be nice that the `Future<int>` of the `M2` call be immediately returned by `M1` instead; due to type inflexibility, it has to make the call synchronize its value before returning the `int` value. The concept of forward [25] aims to solve this issue with a `forward` statement that may replace the `return` statement by specifying that the future is not actually fulfilled and that it will be by the future given as parameter. Enforcing the presence of either `forward` or `return` statements is enough to guarantee that a future will be fulfilled (except when there is an infinite sequence of calls), because the forward concept explicitly delegates the burden to another return statement.

2.4.2 Actors and active objects

The concept of active objects was formally introduced in 1996 by Lavender et al. [48] as a design pattern. The basic principle is to decouple method invocation from method execution. When calling a method, a main thread can ask a second thread to execute this function while continuing its execution. The main thread only needs to perform the invocation. To further embed the calls into easy-to-use language aspects, return values are managed through futures. Upon call, the caller obtains an unresolved future. Upon returning a value from the called thread, the future is fulfilled with this value and can be accessed.

Active objects are basically objects that run in their own thread of control. When a method of an object is called, it will be up to the target active object's thread to run this method. The caller process may then proceed execution without waiting the result because it is managed through a future.

Future variables are wrapping regular data, but this data may not be available at any time. In the context of active objects, the data is only available after the execution of the method (recall each future corresponds to the execution of a method) and must be sent back to the process accessing

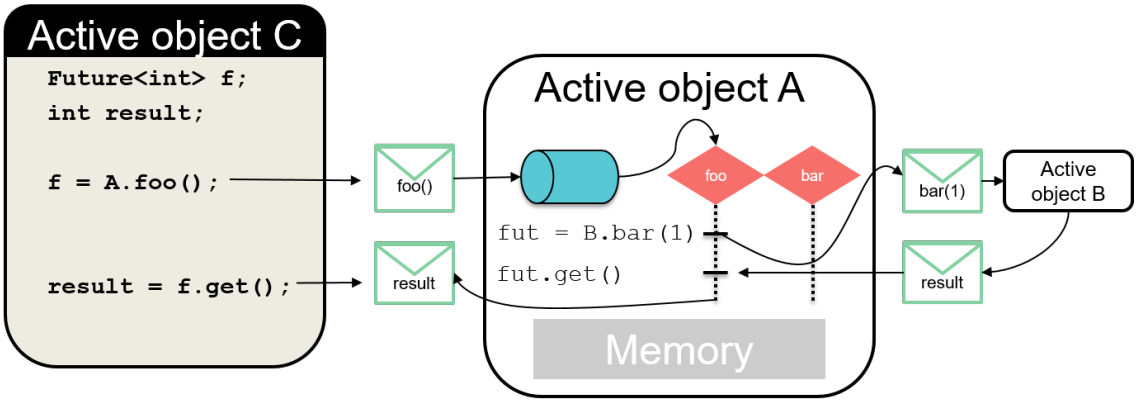


Figure 2.5 – Active objects

it, or later retrieved. Note that if the implementation supports first-class futures, there may be multiple processes accessing the data, as a future can be passed as a method parameter between active objects; in particular a process accessing the data is not necessarily the process that initially called the corresponding active object method when the future object was first created.

Figure 2.5 shows an example execution using active objects where futures are synchronized with a `get` operation as described in Section 2.4.1. When the function `foo` of active object `A` is called by active object `C`, `A` receives a message representing this call and the corresponding parameters into its execution queue. Eventually, this call is executed by `A`'s execution thread as the object's `foo` function. Here, this function calls function `bar` of another active object `B`. A future `fut` is returned from the call, and `A`'s function `foo` is free to do other things before synchronizing the future. When the future is required, the future's `get` function is called, which blocks until the value is produced and sent by `B`. When `foo` of `A` has produced its result, it gives it as return value, and the active object will take this return value and fulfill the future `f` corresponding to the initial call to `foo`. This value can then be retrieved by `C`, which was waiting to synchronize the corresponding future `f` with a `get` call.

This basic principle of active objects is the same as with the actor [4] model, both are parallel entities managed with their own thread. Actors propose building applications by modeling independent entities interacting with each other through messages, these messages may for example

be basic strings or with a type dedicated to the desired interaction. Active objects are actors but propose stronger typing by abstracting the notion of messages from the language. In this sense, an active object is basically an actor managed as an object. Entities call other entities through a typed set of methods associated to each entity. At the language level, it looks like a simple call to an object's function. These objects are called active objects because they each have a dedicated thread and thus are able to live independently. They can serve requests to execute a method from other active objects and they can issue requests of their own. Usual objects are then in contrast called passive objects as they can not behave independently. An active object thus provide a natural object oriented way to schedule tasks on parallel processors.

Because a standard active object is running in its own thread and that this thread is only assigned to a single active object, data-races are naturally prevented when accessing object data because of the guarantee that no other thread has access to the same memory. The only way to communicate between active objects is through method calls, which are processed one at a time (sequentially) by an active object. So there is no need for a programmer to protect data accesses through mutual exclusion mechanism and risk introducing resource deadlocks. As we will see in Section 2.4.4, communication deadlocks can still be introduced depending on how Future accesses are managed. The fact that active objects are programmed as sequential programs gives them a similar ease of programming.

Active objects in general are mentioned in a multi-threading environment, but they may also be used in a distributed system. A proxy can send request messages that are handled on separate machines, at the cost of marshalling and unmarshalling request messages, and network communications.

2.4.3 Languages and implementations

Actors and active objects were implemented in the form of different languages. Here we will give a brief survey of actor and active object languages. A more complete survey of active object languages is given by [13].

Caromel et al. created the ProActive java library, first named Java// [19], this library was then

formalized as the ASP formal semantic [18]. Caromel et al. showed in [18] that the only source of non-determinism in ASP is when two clients are calling the same active object at the same time. Also, they prove that if at any time the request flow graph forms a set of trees then the reduction is deterministic.

A normalization of active objects was attempted by Cunha and Sobral as part of an annotation based framework for Java [58] where a class only requires an ActiveObject annotation to produce active objects when implemented, with an OpenMP-style philosophy in mind. Several annotations for different concurrent behaviors are proposed, with different levels of parallel programming. There are annotations for One way (remote method calls), futures, active objects, barriers, synchronized accesses (protected with locks), read/write locks (with different concurrent protections), scheduling of remote method call execution. While these annotations do not offer the same kind of parallelism than OpenMP, they provide a high level tool to program parallel object oriented applications.

Creol [41] is another active object language, but it has the particularity to allow explicitly releasing the current process from the processor, this enabling cooperative scheduling. The `await e` statement releases the current process if expression `e` evaluates to false, and the fulfillment of a given future `f` can be queried by expression `f?`. Descheduling the current process if the future is not fulfilled can then be achieved by `await l?`, or by `await l?(x)` as shortcut for assigning the result to variable `x` when it is ready. Cooperative scheduling for active objects avoids some deadlocks because a given active object can process another request if one request execution requires a future result not yet available. But cooperative scheduling comes at the cost of saving and loading the context of intermediate variables. Also, the interleaving due to cooperative scheduling can lead to data modifications by several requests served in an interleaved manner: thread scheduling leads to a race condition that can make the object's state inconsistent if not programmed correctly.

Bernstein and Bykov presented the Microsoft Orleans active object middleware [10, 11] who introduces the concept of virtual actors to scale services in a cloud computing environment. Here single threaded active objects are dynamically loaded as they are called, and unloaded as they try to access future variables. Active objects are unloaded into a global store and can then be loaded on any server, which is a compromise on performance to attain scalability and a compromise

between stateless and stateful services. This concept can be seen as an analogy to virtual memory. Depending on the usage, there can be peaks of required active objects and threads, they mention this model is weak for bulk collective operations, because many objects need to be awaken at the same time, giving a peek of required physical threads that may exceed the number of available ones.

Akka [1] is an actor framework built on the top of the Scala and Java programming languages. Messages are created and handled explicitly and an object declares the message types it is able to handle, associating it to an internal method with a given message instance as parameter.

The Abstract Behavioral Specification language (ABS) [34], is an actor modelling language for formal study of distributed systems. Parallel processes are represented by tasks and they interact through asynchronous message exchange where a method that has to be executed on the target is specified. The return values of these methods are managed through futures. ABS uses a concept of Concurent Object Groups (COGs), where a COG represents a processor, which is responsible for running a group of one or several active objects. Only one active object per COG is active on the associated processor. This means a COG effectively shares the usage of a single processor between different active objects. The active objects inside a single COG are not preempted, and must explicitly release control over the processor in order to allow another active object in the same COG to take control; this may be done either through an explicit release point with the `release` statement, or through an `await` operation on a future, in a way similar to the active objects of Creol [41].

2.4.4 Parallel processing

In this section, we will discuss the benefits and weaknesses of having a single thread per actor; we will also discuss some works that attempt to bring parallelism (including data-parallelism) inside actors without sacrificing the benefits of having only a single thread.

One issue with the basic active object model is that an active object has only one thread of control. It is the core concept that makes active objects easy to program and debug. Programming an active object is almost as easy as programming a sequential program, because requests are really programmed sequentially and thus there is no parallelism inside an active object.

This makes proving properties easier but it is a major issue when building scalable multi-threaded applications because by default data manipulated by two threads needs to be duplicated. Also, if accesses to future variables are blocking until the result is available (blocking get access), and the executing thread is not released (no cooperative scheduling), deadlocks in active objects are possible due to the blocking semantics of futures and the inability to reschedule the current thread. For example, when an active object `A` calls active object `B` and gets future `f` from the call, if `A` requires `f` and executes its blocking `get` method, the execution thread of `A` is blocked. If throughout the request execution on `B`, the result depends on the result of a second call on `A` (recursion or not), then `A` is unable to process this request because it is busy waiting for its future `f`, hence a deadlock occurs because the fulfillment of `f` depends on the execution of the second request on `A`, which depends on the fulfillment `f`.

Allowing parallel request handling on active objects can be a solution to this problem because then the second request could be executed in parallel to the first one currently blocked. But this solution requires considering when mutual exclusion is required, and providing a mean to enforce it, because the possibility of having data-races is reintroduced by having several threads possibly accessing the same variables at the same time. In the active object model, a natural way to enforce it is through the request scheduler. As the scheduler dequeues what request it judges as runnable, simply offering more complex conditions in the scheduler can be a solution. This has been studied in the Parallel Actor Monitor framework [55] by implementing the scheduling loop directly.

This solution was judged too low level to be concise by Henrio et al. [37], which provided the multi-active object concept with higher level condition evaluations such as compatibility groups of methods or using a predicate function as entry point to decisions. Henrio and Rochas [38] added a priority mechanism to the multi-active object model where requests can overtake other requests based on their priorities. These priorities are specified by annotations in the language representing a priority graph.

By changing the scheduling of requests, however, the convenience and predictability of the sequential handling of requests (thanks to FIFO queues) is lost. Assumptions on the order of request execution, and absence of data-races are thus more difficult to make. While the approach

is more efficient and more expressive, it requires more expertise from the programmer.

To conclude, the single threaded nature of active objects can be a limitation. Several solutions exist, based either on cooperative multithreading, on the asynchronous reaction to future resolution (preventing blocking a thread), or local multithreading.

2.4.5 Applications

Gibbs used active objects to model multimedia systems [32]. Each active object represents multimedia components such as screens, CD players or speakers, consuming and/or producing multimedia values at specific rate.

Microsoft used Orleans, their Active object middleware for building service for the Halo 4 game [11]. Considering today's leaning towards Service Oriented Architecture for distributed application, the active object concept may look familiar as it can be used to implement services where an active object framework can be part of a middleware.

A successful actor framework is Akka [1], which has been used by several industry big players to build applications along an event-driven architecture.

The Scala language integrates the actor model as its default concurrency mechanism, using just a combination of libraries and convention. Because other concurrency mechanisms exist in Scala, the actor model can be bypassed easily. Indeed, Tasharofi et al. concluded a survey [61] on 16 major Scala projects hosted on github, to analyze why and how developers mixed the actor model with other types of concurrency models for building their applications. They found that 80% of the major projects they selected are mixing actors with manual threads. They found that only 3 out of 16 projects are using actors for distribution, contradicting their initial thoughts that they were using actors for distribution and threads for local parallelism. Instead, their results show that actors tend to be used for local parallelism. They also analyzed how often actors use other kinds of communication mechanisms than asynchronous messaging. They found that only 2 programs use futures to receive messages asynchronously and only 2 other programs use blocking future accesses. At least 6 use other communication means inside actors, 2 communicate through I/O and 4 through shared objects. As they queried the developers, they found 3 main reasons for not using purely

actor communications: the lack of developer experience, some protocols were easier to implement using shared memory, and the lack of efficient I/O provided by the actor libraries.

2.5 Parallel data communication

Data parallelism involves working on distributed data and often producing a distributed result. While a distributed entity may use the distributed result of another distributed entity as input, communication between these parallel entities (parallel tasks) presents a choice of how to communicate this data efficiently. This scenario is referred as the MxN problem, where M parallel processes exchange data with N other parallel processes. The naive approach is to have the first entity gather all data into a single process, and let this process handle communication to the second parallel entity, which requires this data. This other entity is then left with all the data into a single process and decides how to redistribute this data among its peers if needed. This sequential transfer is inefficient as it requires gathering data, transferring it from one process to another, then scattering if redistribution is needed.

Using the low-level programming API such as MPI, data can be directly exchanged by parallel processes using point-to-point communication, or even using MPI collectives on intercommunicators, a concept bridging two independent process subsets. This problem can also be seen as an all-to-all communication pattern in order to optimize communications. However, these communication primitives are usually difficult to use and lack flexibility, as the precise distribution and redistribution has to be known by both parallel entities, which is not convenient for programmers.

On the side of Remote Memory access (RMA) programming paradigms, such as MPI RMA or BSPlib, remote memory is usually declared with a fixed size prior to communications. All remote memory segments put together implicitly form a distributed array. A collection of communications from one distributed area to another effectively forms a distributed communication, however this approach is also difficult to use and lacks flexibility because, just as with message passing, the distribution on other processes has to be known by the group that initiates the communication.

The GCM distributed software components model [7] and its implementation in the ProActive

active object middleware specifies collective interfaces between parallel and distributed hierarchical components. A broadcast interface replicates a service method invocation on parallel components of the same type, and can automatically split a call parameter of type collection into parts distributed evenly on the target components. Reception of the call's result blocks until all parts have been retrieved as they are aggregated as a collection. A gathercast interface synchronizes incoming method calls from a set of parallel components to form a single one, where pieces provided for a given call parameter are aggregated as a collection, before the call is forwarded to the receiving component; on return, if the result (received on a central point) is a collection, it can be split in parts, returned to respective callers. An MxN collective interface combines a gathercast from M parallel components, and a multicast to N parallel components. The combination of gathercast-multicast effectively gathers the distributed data from M processes into a single process, which is the hierarchical component embedding the M inner components, which then broadcasts pieces to the N other processes, themselves embedded in a hierarchical component. The optimized implementation of an MxN in GCM/ProActive automatically replaces this combined view as follows: each of the M parallel components gets a new multicast interface, bound to the subset of the N parallel components to which it needs to distribute pieces of the call parameter; each of the N components exposes a gathercast interface from which it can synchronize invocations and receive the needed pieces in order to execute the invoked method once. The optimization directly binds the M and N processes to parallelize the communication, this, bypassing the two hierarchical components. To enable it, the user has to implement an MxN component, including a hierarchical and parallel component of size M, and another hierarchical component of size N, and associated controller(s) which specify the distribution. Here, the data distribution needs to be known at composition time to create the adequate component interfaces with their corresponding controllers.

Research work by Keahey et. al. [43] suggests that a central component is needed in order to achieve maximum flexibility for solving the MxN problem. They introduce a notion of collective ports, which allows collective components to act as one parallel entity. They introduce a class of translation component, which translates between data distribution formats used by two different components. This component provides its own standardized way of representing data distribution,

and is considered to be distributed over the sum of the producing and consumer components. They note that having this central component which coordinates the transfer enables transferring data collectively, which is more efficient than having transfers performed individually, but that this costs more synchronization, hence latency. The approach of this work however has the performance penalty of having to copy data to and from the parallel component.

Dameveski et. al. propose that Parallel Remote Method Invocation (PRMI), which involves a collective call to and/or from parallel components, be integrated so that automatic data redistribution mechanisms are triggered, this is achieved at the level of an interface definition language compiler [23]. They base their distribution representation on the model used by PAWS [9], a distribution is represented with three elements: the index of the first element, the index of the last element, and the stride, which is the number of array spaces between two elements. The distribution schedule is then expressed as a collection of intersecting distributions, which represents the data that has to be transferred between processes. While MxN redistribution is performed through PRMI through argument passing, a call means data has to be available, and a callee receives the data upon this call. This approach requires synchronizing data between each call, without allowing to delay synchronizing and transferring the data if it is not needed by the target component or if this component could do something else before this data is available.

The notion of ParT [26] represents an array of futures, representing results from different invocations, effectively representing data among parallel processes, even if the implementation of ParT is not distributed. Synchronizing elements of a ParT effectively gathers data from parallel processes. ParT can be used to implement speculative parallelism or barriers gathering a set of results. The set of futures represents different method calls and not a single method call producing a distributed array.

To summarize, several approaches exist for data distribution and communication. They all implement different compromises but we deem none of them are sufficiently well integrated in the language and they seem to build too much on existing well-founded mechanism instead of trying to integrate distributed data and parallel function calls into the execution model and its language.

2.6 Conclusion

In this chapter, we have seen that the difficult aspects of parallel programming can be abstracted into higher level programming models and languages. This often leads to works that attempt to bring parallel programming closer to sequential programming, and we have seen that active objects and BSP are specialized models, respectively for task and data parallelism, that aim to do just that. The only synchronization mechanism provided by the BSP model is the barrier synchronization; even if the programming interface exposes point-to-point communication primitives as in BSPlib, they are only synchronized during the barrier synchronization, which makes programs relatively safe, and easy to program and debug. Actors and active objects also make programming closer to sequential programming by limiting the concurrency mechanisms exposed to the programmer: the only ways for processes to communicate are through remote method calls and future synchronizations.

We have seen that one major issue with the BSPlib programming API is that it prevents the efficient cooperation of several functionally different tasks in parallel due to its rigid barrier structure. This issue was addressed in several forms, the more general of them is subset synchronization, which partitions processes into subgroups which communicate using communication involving only themselves. This allows groups to execute different codes with different synchronization patterns, at the cost of compromising the accuracy of the BSP cost model.

On the other hand, we have seen that there has been works to parallelize the execution of an active object, but we deem these approaches too heavy and restrictive. This is because active objects require going through the queuing mechanism for every communication between processes. When implementing data-parallelism with actors, each sub-task is managed through a dedicated active object, these communications between the sub-tasks are restricted to active object parameter passing. Instead, they could communicate directly with each other during the execution of their request. This would avoid suspending their execution or introducing a deadlock because an active object sends data to another that is already executing a similar function.

In the next chapter, we will show another approach solving the enumerated shortcomings of both models, by actually unifying them.

Chapter 3

BSP Active Objects

3.1 Introduction

In this chapter, we propose a programming methodology that mixes BSP, a well-structured data-parallel programming model, with actor-based high-level interactions between asynchronous entities. This way, we are able to express in a single programming model, several tightly coupled data-parallel algorithms interacting asynchronously. More precisely we design an active-object language where each active object is able to run parallel BSP code using multiple processes owned by the same active object. The communication between active objects is remote method invocation, while it is delayed memory read/write inside the BSP code. These two programming models were chosen because of their properties: BSP features predictable performance, absence of deadlocks under simple hypotheses and relative ease of programming compared to more general purpose parallel programming models. Active objects have only a few sources of non-determinism and provide high-level asynchronous interactions. Both models ensure *absence of data races*, thus our global model features this valuable property. The benefits we expect from our mixed model are the enrichment of efficient data-parallel BSP with service-like interactions featured by active objects. Elasticity is also a feature of our model. Indeed, creating a new active object creates a new thread; this is not a feature that is part of the classical BSP model. Automatic scaling as the one available in most

cloud services could be implemented by a user program, but it is not a goal of the model itself.

We start this chapter by describing our model in Section 3.2, then we illustrate this model through a code example using our C++ implementation in Section 3.3. We then follow in Section 3.4 with some implementation aspects that proved to be key in the design of a distributed implementation, before concluding in Section 3.5.

3.2 Execution model

3.2.1 Design choices

We have seen in Section 2.4.2 that active objects enable a natural way to schedule tasks on parallel processors. In order to keep this benefit, it is natural to keep active objects as the top-level model of a hybrid BSP active object model. We go with the following guideline: active objects should be used to coordinate high-level tasks, and a BSP execution of these tasks should speed them up with multiple processes. After this choice is made, it now becomes clear that active objects should now have more than one process each to properly embed parallel tasks. The next points to address are now how to use these processes, and at what point in time should we consider using them for parallel execution.

To help make the description of our model clear, we first give definitions for some of the vocabulary we will use. A *task* is a job handled by an active object. We refer to a single task even if this task is parallel and involves multiple processes on a single active object. We sometimes refer to a *parallel* task to emphasize that a task involves multiple processes. For example, if we use the words *parallel tasks*, we refer to multiple active objects, each of them solving their problem in parallel with multiple processes each. We refer to these processes as active object processes, with possibly several of these processes assigned to a single active object.

We already know that from the outside, a task should look like an active object function, and its result should therefore be represented as a future. For example, and in order to achieve the easiest and most familiar active object call syntax, a call should still look like `fut = a.foo(p1,p2)`. After defining this, the first choice has to be made: is the execution becoming parallel right at the start

of the `foo` function ? If so, then some questions have to be answered: how to distribute the call parameters and how to manage the return values; if not, then the parameters can be transferred directly to a single process, or this process can coordinate the distribution of parameters if needed. While addressing this choice, it must be kept in mind that the solution should be kept simple, both to ease an implementation and to avoid exposing too much complexity to the user.

Managing the passing of parameters to a parallel remote method requires a choice on how to send the parameters. These parameters may be sent to just one process, to all the processes, or, in the case of array parameters, they may be distributed among all the processes. This choice, especially for large arrays, has an impact on communication performance. A similar choice has to be made for return values: how to manage the return values of a parallel execution. The return value of an active object function is usually managed through a future, which represents a single object. This single object may represent an array, but it can not store a distributed array.

For usual active objects, a distributed result may be produced through multiple active objects, and represented by a collection of future objects. More optimized solutions based on this principle were developed such as ParT [26]. We could adopt a solution based on having an array of futures to provide a distributed result, but this would require multiple active objects and we are more interested in working with a single active object to produce a result in parallel.

From the reasoning we detailed above, and the complexity of solutions that must be addressed, we decided that an active object call should at first trigger only a sequential execution. This sequential execution is handled by one process among those owned by the called active object. We also decided that this process should always be the same for a given active object in order to keep the state of object variables after different calls on the same active object. Choosing otherwise would leave open the possibility of an active object to execute several sequential requests in parallel as proposed in multi-active objects [37]. However, in our work, we are not interested in the parallel execution of different requests simultaneously in the same object, but rather in the execution in parallel of a single request at a time. Picking the same process starting sequential executions on an active object effectively closes the possibility of parallel execution of different requests, but conveniently allows the programmer to keep the same context between different calls, because the

same process has the same memory, and thus the same state of variables. To better describe our model, we refer to this process in an active object as its *head* process.

Starting sequentially the execution of a method allows its parameters to be sent from just one process to one other process, as simply as a normal active object call. The caller does not have to make a decision on how to transfer the parameters to the parallel processes as this will be handled by the receiving active object's head process. As the return value is returned by this head process, this solution also allows the return value to be managed by a single future, may this value be a simple integer or an array. This has some drawbacks for communication of large arrays, which we address through distributed futures in Chapter 5. For now, this solution allows a base model and implementation to be kept simple by default and allows extensions to handle more complex parameter passing as well as return values later on.

Now that we have decided that the execution of a task should be started sequentially, comes the choices of how to switch from sequential to parallel execution. We chose to create a dedicated primitive called `bsp_run`, with another function as parameter that is to be called in parallel by all the active object processes. Another choice to make is whether the head process should be part of the called parallel function. We deemed this choice to be mostly related to the nature of the parallel execution model. Since we chose BSP as the parallel execution model, we saw no counter-indication to using all the processes at the same time, including the head process. If we would have chosen the parallel execution model differently, for example a CUDA execution for GPUs, where parallel GPU execution using tightly coupled threads is orchestrated from a CPU thread (which would be the head process in our context), it would have made sense to exclude the head process from the processes used for parallel execution. Because of how different purpose and resource configurations the CPU thread has compared to the GPU threads, it would be inappropriate to have the CPU thread perform the same kind of work as the GPU threads.

3.2.2 Model overview

In the previous section we have explained a few choices that were necessary to make in order to design a hybrid BSP active object model, along with the decisions we made and for what purpose.

Here is a summary of the decisions that were taken:

- A call to a BSP active object should be as similar as possible to a normal active object
- Execution of a parallel task starts sequentially; the active object process handling sequential execution is referred as the *head* process
- The head process is always the same throughout different function calls
- Call parameters are passed sequentially from the calling process to the head process
- The head process returns the result value
- The head process is able to start BSP parallel execution along with other processes using a new `bsp_run` primitive
- The head process is part of the parallel execution

We have detailed the choices regarding the originalities of our hybrid model above. Since we join two existing programming models, each of them having different implementations with their own originalities, we must also specify the kind of active objects and BSP variant we want to use for our model. Some variations may not matter specifically for our hybrid model, and different choices may not break its core principles, but could complicate the base model and its implementation.

First, at the top level of our model, we chose to use active objets and futures which are as simple and explicit as possible. This is because we do not want to complicate our model with unnecessary details and we prefer to be explicit regarding the behavior of our active objects and futures so that our examples expose no complexity to the reader. Each active object, once created, is assigned to one or multiple processes available in the execution environment. In particular, and for the reason of simplicity, we do not integrate the notion of COGs as exist in ABS [34]. Integrating COGs would enable original aspects such as having different active objects share different processors; while possible, this would not only make our model more complex, but make it possible to have overlapping processor subsets, which part of the BSP community regards as a worse variant of subset synchronization because of communication interferences between different subsets.

We also specify that the requests are enqueued in FIFO order into the request queue of each active object, so that the execution order of several requests may be predicted to some extent. More complex queuing mechanisms such as request priorities or a different request ordering would not interfere with the principles of our model.

As for the syntax and typing of futures, we chose futures with explicit types because we prefer our examples to be easily readable, and because explicit futures were easy to implement in our C++ implementation. Also, we do not explore recursive active object calls, which would benefit from implicit or dataflow explicit futures. While we made this choice of future types for the aforementioned reasons, a different choice, in some particular implementation that would require or benefit from implicit futures, would not go against the concepts of our model.

In our language, the synchronization of a future is triggered by a `get` operation, which blocks the current thread until the value associated to the future is available. We chose `get` operations as the only way to synchronize futures to simplify the implementation and have precise control over the runtime. Allowing cooperative scheduling through `await` would raise questions such as what to do when a thread in a parallel function synchronizes a future and get de-scheduled, while the other threads continue with the execution. Tackling this question would greatly complexify our model and implementation, which is why we chose to stick with simple blocking `get` operations.

We chose a "lazy message" synchronization strategy as described in Section 2.4.1 in order to ease the understanding of events happening at runtime: a `get` operation triggers a request and the possible transfer of the value associated to the future. At this point, there is no other argument against another synchronization strategy, but we will see that the contributions of Chapter 5 will complexify the introduction of eager strategies.

To summarize briefly, we can say that our base active object model resembles the one of ASP [18], but with explicit futures and lazy future synchronization as the only synchronization strategy.

We now detail how a request is executed following the example of Figure 3.1. In this example, we have an active object `A` which, from the outside, behaves similarly to the actor presented in Figure 2.5. Our BSP active object can be called, just as a normal active object, through a function call in order to produce and send a request message. Each active object has its own request queue,

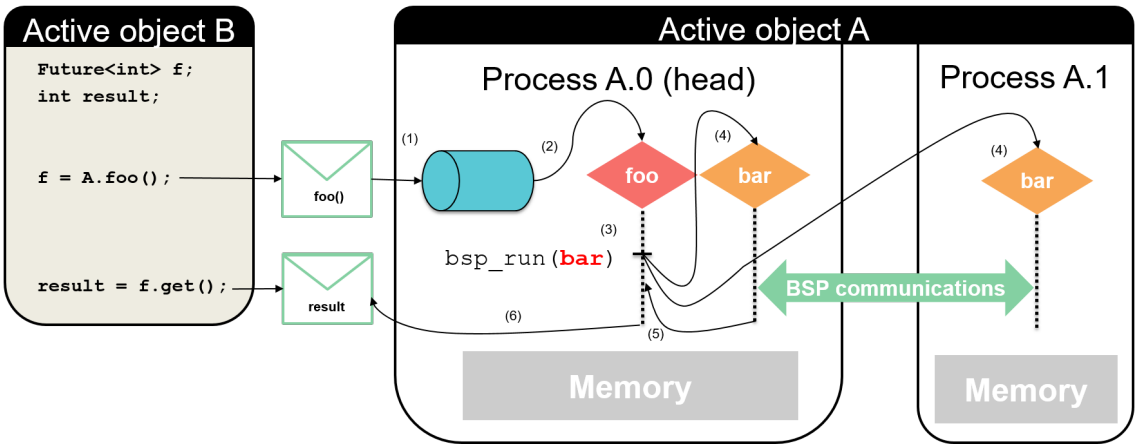


Figure 3.1 – BSP active object model

and each of its processes has its own memory; the programmer is exposed to no shared memory, not only between different active objects, but there is also no shared memory between the parallel processes of an active object.

The first noticeable difference we can see between an usual active object of Figure 2.5 and our BSP active object of Figure 3.1 is that the BSP active object has two processes, process `A.0` and process `A.1`. The head process is, by convention, the first process (`A.0`). Active object `A` is able to execute function `foo` sequentially, and function `bar` in parallel. In our scenario, the `foo` function is first called on `A`. Just as with a normal active object, this translates into a message being sent from the caller to `A`'s request queue, here stored on its head process, as we can see in (1). Eventually, the active object is going to be execute the request, running the `foo` function, here in (2). Inside the `foo` function (3), the `bsp_run` function is called with function `bar` as parameter. This makes the head process `A.0` require process `A.1` to participate in the parallel execution of function `bar`, as shown in (4). If there were other processes than `A.1`, they would also be required to participate. After this, both processes (`A.0` and `A.1`) enter function `bar` in parallel. From this point, the execution is parallel and following the BSP model. This means communications between the two processes are orchestrated in supersteps. After both processes finished executing `bar`, `A.0` goes back to executing `foo` where it was (5), after the call to `bsp_run`. From this point, the execution came back to

sequential execution and `A.1` is not involved. The head process is free to call `bsp_run` again, but here it simply proceeds and returns a `result` value in (6). This return value fills the future that was created by calling this active object's function `foo`.

While explaining Figure 3.1, we mentioned that processes communicate following the BSP model during the execution of the `bar` parallel function. We chose a direct-mode of BSP communication so that the programmer has precise control over communications, which are explicitly triggered. The BSPlib standard, described in Section 2.3.6, is an existing specification for direct-mode BSP programming, with we conveniently integrated in our BSP active object model along with some of its existing implementations.

We have shown how our BSP active objects work internally, and several of these active objects can coexist: they communicate through method calls and futures, just as how it can be expected from a classic active object model. In Chapter 5, we will introduce another mean of communication that is more specialized to multi-process actors.

3.3 BSP active objects by example

In this section, we will give an example of what the syntax of the BSP active object model can look like. Here we use the syntax that is used in our C++ BSPlib implementation.

In this example we also show how we can reuse existing code written in BSPlib. For consistency of examples, we reuse the inner product code from BSPedupack [12] that we showed in Figure 2.2 to present BSPlib. We do not reuse the codes of Figure 2.3 and Figure 2.4 to initialize the example because theses codes generate arguments and forward them to the inner-product code. Handling arguments is now managed by the active object interface.

When the head process triggers a parallel execution using the `bsp_run` primitive, all processes enter the function given as parameter, including the head process. Because the sequential and parallel functions do not share the same function scope, the only scope they share is the object scope. In order for the head process to share data between its sequential and parallel execution contexts, variables at the object scope must be used. This is the case in our example, where the

return value is produced in the parallel context and returned from the sequential context. After the execution of the parallel function, each process has the value set in variable `_alpha` (cf line 26 of Figure 3.2). In our example, this variable is at the object scope so that the head process can return it from its sequential execution context. In this example, and by convention, all variables starting with '_', are variables at the object scope, thus they are reachable from any function in the object; this is useful because the function used for calling the active object (`ip`) is different than the function used for parallel execution(`bspinprod`). The head process, which enters both, may use these variables to keep some context between parallel and sequential executions.

The IPActor class interfaces the inner product implementation included in the BSPedupack software package [12]. We only show parts of the code we deem interesting to present our model.

In this example, we show how an active object can encapsulate process data through object variables and how its function interface can act as a parameterized sequential entry point to a parallel computation. We also show the result of a call being used to call another active object to do another computation, which is not shown.

In the `main` function, each active object is created and given two processes, here their pids are given directly as parameters of the `createActor` primitive (lines 34-35). Then the `ip` function of the first active object is called with vector `v` as the two parameters. This asynchronous call returns with a future `f1`. As explained above, the `ip` function of this active object is run by its head process sequentially. Using BSP primitives, the input vectors are split among the processes of the active object. Here, for simplicity, we assume `bsp_nprocs()` divides `v1.size()`. Then the `bsp_run` primitive is used to run `bspinprod` in parallel, which calls the `bspip` function of BSPedupack. Immediately after the call on the first active object, the main method requests the result with a `get` primitive on `f1`, blocking until the result is ready (line 38). We also show this result being sent to another active object (`actorB`) as request parameter of a `multiply_all` function.

A view of the active object running the BSP inner product is given in Figure 3.3. First, the head process receives vectors `v1` and `v2` in its memory. It first initiates the sending (lines 14-15) of vector parts to other processes so that they are evenly distributed, even though they do not participate yet. Then the head process calls the `bsp_run` primitive, which enables the other processes of

```
class IPActor : public activebsp::ActorBase {
  private:
    double * _x_part;
    double * _y_part;
    int _n_part; // Number of elements in each vector part (_x_part and _y_part)
    double _alpha;

  public:
    double ip(vector<double> v1, vector<double> v2) {
        // ... (Initialize _n_part and allocate then register _x_part and _y_part)
        for (int i = 0; i < bsp_nprocs(); ++i) { // Split data
            int i_beg = _n_part * i;
            // Send array block starting at offset to process i
            bsp_put(i,&v1[i_beg],_x_part, 0, _n_part*sizeof(double));
            bsp_put(i,&v2[i_beg],_y_part, 0, _n_part*sizeof(double));
        }

        bsp_run(&IPActor::bspinprod);

        return _alpha;
    }

    void bspinprod() {
        bsp_sync();
        // call BSPedupack inner product, returns result on all p
        _alpha = bspip(bsp_nprocs(), bsp_pid(), _n_part, _x_part, _y_part);
    }
};

int main() {
    // ...
    vector<double> v;
    // ...
    Proxy<IPActor>   actorA = createActor<IPActor>  ({1,2});
    Proxy<MultActor> actorB = createActor<MultActor>({3,4});

    Future<double> f1 = actorA.ip(v,v);
    double ip = f1.get();
    Future<vector<double>> f2 = actorB.multiply_all(v, ip);
    v = f2.get();
    // ...
}
```

Figure 3.2 – ActiveBSP example

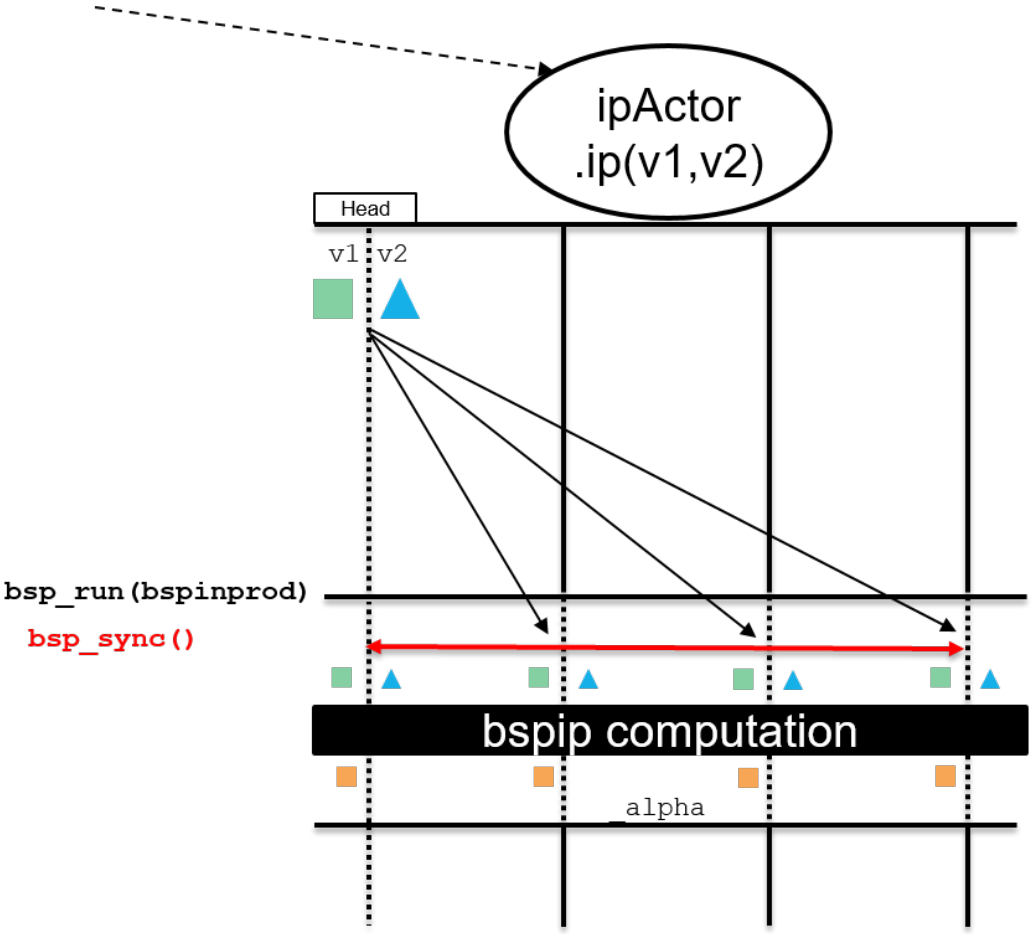


Figure 3.3 – Inner product execution

this active object to participate by entering the `bspinprod` function together in parallel. Then the BSPlib's `bsp_sync` function is called (line 24) so that the current superstep is ended and the initiated communications by the `bsp_put` calls take effect. From this point, every process gets its relevant pieces of vectors `v1` and `v2` in its memory. After this, the inner product computation is performed inside the `bspip` function, and every process gets the result `_alpha` in its memory, as implemented in the inner product computation of BSPedupack, even though we are only interested in the `_alpha` variable contained in the head process memory. The parallel function is then finished and the head process proceeds with sequential execution, it just has to return the content of the `_alpha` variable, which contains the previously computed result of the inner product computation.

3.4 Management thread for distributed implementation

In this section, we describe an implemention aspect we had to use for our distributed implementation of active objects: the management thread.

3.4.1 Motivation and terminology

Active objects are asynchronous in nature. They represent independent entities working in their own thread that can be called by other threads to request the execution of some function. When an active object is already executing a function, it must also be available when another active object calls it; in particular, it must be able to receive requests so that a caller thread does not remain blocked until the called active object finishes to execute a previous request. In shared memory architectures, this problem does not occur, because a caller has access to the memory of the callee, and can just queue its request on a shared memory queue. A distributed memory implementation does not have this easiness, even in Remote Memory Access programming. A solution to this problem is to have a separate thread handle communications with other active objects. To make our description clear, we give some definitions regarding these threads.

The *worker thread* is responsible for executing the active object user code, its role is handling the active object requests that it reads from its queue.

The *management thread* is responsible for enforcing the reponsiveness of an active object, for example, receiving requests from the outside and putting them into the active object request queue while the worker thread is processing another request.

An *active object process* is a process that encompasses the management and worker threads. It does not have its own execution context, and we use this term when we are not interested in whether we reason on the management or the worker thread. This is sufficient when we put our discussion at the level of active objects and their processes in charge of parallelizing a task.

Note that the head process of an active object is just one active object process that satisfies to this definition, and not another term for management or worker thread.

The motivation we gave so far for having a separate management thread only concerns the head process because it handles active object requests. We will see in Chapter 5 that there will be a reason for having a management thread for not only the head process, but so far, note that this is not required. Also, it must be kept in mind that the need for a management thread emerged from our interest in a distributed implementation.

3.4.2 Illustration: Processes and threads

Figure 3.4 shows the inner-workings of the head process of a BSP active object (`objectA`) handling two requests and returning the values of the corresponding futures. While this active object may have several processes, here we focus on its head process and its corresponding management and worker threads. In the illustrated scenario, an object `objectB` calls `foo` on the same object (`objectA`) two consecutive times, then retrieves the two corresponding futures, successively, with a blocking `get` operation. Both active object calls send a corresponding request to the management thread, that just handles the request by queuing it into a shared memory queue. The first request is then dequeued by the worker thread, which starts handling this request.

We implemented first-class futures which have a lazy synchronization strategy (As described in Section 3.2.2). The data associated to the future is stored on the object that computed it, which transfers this data to any other active object which requests it with a `get` operation on the corresponding future object.

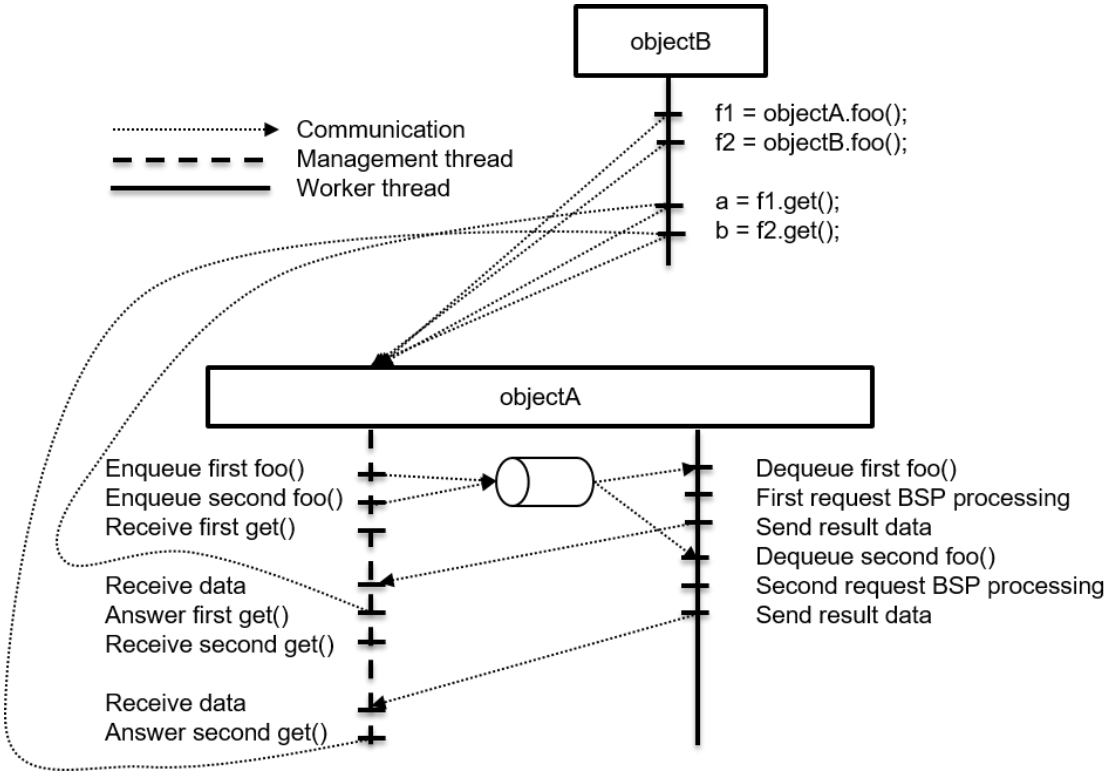


Figure 3.4 – Head process request handling with management thread

Handling a request can take any amount of time, depending on the active object function itself, but here we will assume, for illustrative purposes, that it takes enough time so that the execution of the two requests does not finish instantaneously and the management thread has time to handle the first `get` request (the second is not sent because the first is blocking). When handling this `get` request, the management thread does not yet have a result to answer, because the worker thread did not finish its computation. If the result would have already been computed, then the management thread could have answered the result right away, but at this point in time, it simply remembers that the future was requested and it continues being available for other requests (here there is none yet so it does nothing). This leaves `objectB` pending because it is waiting for the result of the `get` operation on the `f1` future, which the management thread did not yet answer. When the worker thread finished executing its function and possible (but not necessary) associated BSP computations, it sends the associated return value to the management thread and then proceeds with handling the next foo() queued request. The management thread then takes care of forwarding the data corresponding to the `f1` future to any process that requests it, here it was requested by `objectB`. When the result is transfered, the `get` operation on `f1` unblocks on `objectB` and the `get` operation on `f2` is then executed, and similar steps follow for handling the future.

We chose that the worker thread would not be responsible for communicating the result data because there can be a request at any point in time for this data, and when the worker thread is busy doing some computation, it is not available to handle request for data communication. This would effectively block the `get` operation on a future already computed if the active object that produced the value is busy processing another request. Instead it forwards the result to the management thread, which has the responsibility to be available to answer the value to whoever requests it.

3.5 Conclusion

In this chapter, we have presented a new programming model for the coordination of BSP processes. It consists of an actor-like interaction pattern between SPMD processes. Each actor is able to run an

SPMD algorithm expressed in BSP. The active-object programming model allowed us to integrate these notions together by using object methods as entry points for asynchronous requests and for data-parallel algorithms. Our model can be seen as a way to program subset-synchronized BSP algorithms, while naturally enforcing disjoint subsets. We have shown an example of this model that features two different BSP tasks coordinated through dedicated active objects. This example also shows the usage of our C++ library implementing this model, that relies on BSPlib primitives for intra-actor data-parallel computations. We have also discussed some key implementation aspects that have to be addressed while implementing this model in a distributed setting.

Chapter 4

Formalization

4.1 Introduction

In this chapter, we present the formal language implementing the BSP active object model we introduced in Chapter 3. We are interested in clear semantics for our model because it formally specifies the behavior of our execution model, and it can serve as a basis for formal reasoning and proof of properties, both concerning the programming model and specific applications.

While our implementation, that we will present in Chapter 6, is a way to show precisely how our model works, this description is downed with implementation details not specific to the model, and with choices that are not part of the model itself and could have been different. The formal description presented in this chapter provides a clear view of the model through the ABSP formal language that is abstract from implementation details.

This chapter is organized as follows: first, we present the syntax of our formal language in Section 4.2, then we describe its semantics in Section 4.3. We then show an illustrative example in Section 4.4 before concluding in Section 4.5.

$$
\begin{array}{rcll}
P & ::= & \overline{\texttt{Act}\{\overline{T\ x}\ \ \overline{M}\}}\ \{\overline{T\ x}\ s\} & \text{program} \\
T & ::= & \texttt{Int} \mid \texttt{Bool} \mid \texttt{Act} \mid \texttt{Fut} < T > & \text{type} \\
M & ::= & T\ \texttt{m}(\overline{T\ x})\ \{\overline{T\ x}\ s\} & \text{method} \\
s & ::= & \texttt{skip} \mid x = z \mid \texttt{if}\ v\ \{\, s\,\}\ \texttt{else}\ \{\, s\,\} \mid s\ \texttt{;}\ s & \\
 & \mid & \texttt{return}\ v \mid \texttt{return} \mid \texttt{BSPrun}(\texttt{m}) \mid \texttt{sync} & \\
 & \mid & \texttt{bsp\_get}(v, x, y) \mid \texttt{bsp\_put}(v, v, x) & \text{statements} \\
z & ::= & e \mid v.\texttt{m}(\overline{v}) \mid \texttt{new}\ \texttt{Act}(N, \overline{v}) \mid \texttt{get}\ v & \text{rhs of assign} \\
e & ::= & v \mid v \oplus v & \text{expressions} \\
v & ::= & x \mid \texttt{null} \mid \textit{integer-values} & \text{atoms}
\end{array}
$$

Figure 4.1 – Static syntax of BSP active objects

4.2 Syntax

ABSP is our core language for expressing the semantics of BSP processes encapsulated inside active objects. Its syntax is shown in Figure 4.1, x ranges over variable names, m over method names, α, β over actor names, f over future names, and i, j, k, N over integers that are used as process identifiers or number of processes; a horizontal bar over terms means it is an ordered list of the terms below the bar. A program P is made of a main method and a set of object classes with name `Act`, each having a set of fields and a set of methods. The main method identifies the starting point of the program. Each method M has a return type, a name, a set of parameters x, and a body. The body is made of a set of local variables and a statement. For simplicity and clarity, we assume that local variables and fields have disjoint names but we could specify rules for allowing their overlapping and prioritizing their selections.

Types T and terms are standard for object languages, except that `new` creates an active object, `get` accesses a future, and $v.m(\overline{v})$ performs an asynchronous method invocation on an active object and creates a future. The operators for dealing with BSP computations are: `BSPrun`(m) that triggers the parallel execution of several instances of the method m; `sync` delimits BSP supersteps; and `bsp_put` writes data on a parallel instance.

`bsp_put` is a request for writing a value into the field of an object associated with a parallel process at the end of the current superstep. A `bsp_get` operation is similar: it will write the field of an object running in parallel into the field of the local object. Delaying the effect of `bsp_get`

and `bsp_put` to the next `sync` prevents data-races as the `sync` operation handles these requests as a collection when they are all requested. There is one noticeable difference in the syntax of `bsp_get` and `bsp_put` in Figure 4.1: the type of their parameters. This difference may raise questions because both functions are similar in purpose, one puts the value of a local variable into a remote variable and the second one gets the value of a remote variable into a local variable. Both operations happen at the execution of a `sync`, but the difference between the two functions is that `bsp_put` takes the value of the local variable at the time of execution of this operation while `bsp_get` only evaluates the value of the copied variable upon synchronisation. This means that after the execution of `bsp_put`, the value of the variable may change, but the value that is going to be written into the remote variable is the value of the variable when the `bsp_put` is executed; any subsequent change to the variable does not change the value that is going to be written when `sync` is executed. This is different for `bsp_get`, which takes the value of the remote variable at the time of the execution of the `sync`, simply because the process executing this operation can not know this value without communicating. This is why the second and third argument types of both functions are different, these are two variable references for `bsp_get`, and one value and one variable reference for `bsp_put` instead. As we just explained and as we will show in Section 4.3, this is because `bsp_put` saves the value of its second argument at the time of execution of the `bsp_put`.

Sequence is denoted ; and is associative with a neutral element `skip`. Each statement can be written as $s; s'$ with s neither `skip` nor a sequence.

4.2.1 Design choices

We chose to specify a FIFO request service policy like in ASP because it exists in several implementations and makes programming easier. In ABSP, all objects are active; a richer model using passive objects or concurrent object groups [13] would be more complex. We choose a simple semantics for futures: futures are explicit and typed by a parametric type with a simple `get` operator. We chose a BSP syntax similar to BSPlib because it is a well-known library for BSP programming. We decided to model only DRMA-style communications although message passing also exists in

$$
\begin{array}{rcll}
cn & ::= & \overline{\alpha(N, A, p, \overline{q}, \mathit{Upd})}\ \overline{f(\bot)}\ \overline{f(w)} & \text{configuration} \\
p & ::= & \varnothing \mid q : \left([\overline{i} \mapsto \overline{\mathit{Task}}]\ ;\ j \mapsto \mathit{Task}\right) & \text{current request service} \\
q & ::= & (f, m, \overline{w}) & \text{request id} \\
\mathit{Task} & ::= & \{\ell | s\} & \text{task} \\
w & ::= & x \mid \alpha \mid f \mid \texttt{null} \mid \textit{integer-values} & \text{runtime values} \\
\ell, a & ::= & [\overline{x} \mapsto \overline{w}] & \text{local store} \\
A & ::= & [\overline{i} \mapsto \overline{a}] & \text{object fields} \\
e & ::= & w \mid v \oplus v & \text{runtime expressions} \\
\mathit{Upd} & ::= & \overline{(i_{src}, v, i_{dst}, x)} & \text{DRMA operations}
\end{array}
$$

Figure 4.2 – Runtime Syntax of BSP active objects (terms identical to the static syntax omitted).

BSPlib; modelling messages between processes hosted on the same active object would raise no additional difficulty, but would complexify our semantics, which aims to be clear and simple.

4.3 Semantics

The semantics of ABSP is expressed as a small-step operational semantics in Figure 4.3 and Figure 4.4.

A small example operational semantics rule is given below for illustrative purpose.

$$\frac{C1 \qquad C2}{\mathit{InitialState} \rightarrow \mathit{ResultState}}$$

The horizontal bar between the two parts of this rule separates the *pre-condition* on the top, and the *post-condition* on the bottom. This rule is read as follows: when the initial state matches $\mathit{InitialState}$ and the preconditions $C1$ and $C2$ are satisfied, the state *may* evolve into $\mathit{ResultState}$. If other rules may apply at the same time, it is not defined which rule applies first; this is typical for formalizations of parallel languages, where processes evolve independently of each other. Note that intermediate variables may be defined in precondition, and be reused in the definition of the $\mathit{ResultState}$.

Our operational semantics in Figure 4.3 and Figure 4.4 relies on the definition of *runtime configurations* in Figure 4.2 which represent states reached during the intermediate steps of the

$$\frac{w \text{ is not a variable}}{[\![w]\!]_\ell \;=\; w} \qquad \frac{x \in \mathrm{dom}(\ell)}{[\![x]\!]_\ell \;=\; \ell(x)} \qquad \frac{[\![v]\!]_\ell = k \qquad [\![v']\!]_\ell = k'}{[\![v \oplus v']\!]_\ell \;=\; k \oplus k'}$$

New

$$\frac{\begin{array}{c}[\![\overline{v}]\!]_{a+\ell} = \overline{w} \\ \overline{y} = \mathit{fields}(\texttt{Act}) \qquad A = [j \mapsto (\overline{y} \mapsto \overline{w}, \mathit{pid} \mapsto j, \mathit{nprocs} \mapsto N) \mid j \in [0..N-1]] \qquad \beta \text{ fresh}\end{array}}{\mathcal{R}_k[a, \ell, x = \texttt{new Act}(N, \overline{v}) \;;\; s] \to \mathcal{R}_k[a, \ell, x = \beta \;;\; s]\; \beta(N, A, \varnothing, \varnothing, \varnothing)}$$

If-True

$$\frac{[\![v]\!]_{a+\ell} = \texttt{true}}{\begin{array}{c}\mathcal{R}_k[a, \ell, \texttt{if } v \,\{\, s_1 \,\}\, \texttt{else} \,\{\, s_2 \,\} \;;\; s] \\ \to \mathcal{R}_k[a, \ell, s_1 \;;\; s]\end{array}}$$

If-False

$$\frac{[\![v]\!]_{a+\ell} \neq \texttt{true}}{\begin{array}{c}\mathcal{R}_k[a, \ell, \texttt{if } v \,\{\, s_1 \,\}\, \texttt{else} \,\{\, s_2 \,\} \;;\; s] \\ \to \mathcal{R}_k[a, \ell, s_2 \;;\; s]\end{array}}$$

Get

$$\frac{[\![v]\!]_{a+\ell} = f}{\mathcal{R}_k[a, \ell, y = \texttt{get } v \;;\; s]\; f(w) \to \mathcal{R}_k[a, \ell, y = w \;;\; s]\; f(w)}$$

Assign

$$\frac{[\![e]\!]_{a+\ell} = w \qquad (a + \ell)[x \mapsto w] = a' + \ell'}{\mathcal{R}_k[a, \ell, x = e \;;\; s] \to \mathcal{R}_k[a', \ell', s]}$$

Invk

$$\frac{\begin{array}{c}[\![v]\!]_{a+\ell} = \beta \qquad [\![\overline{v}]\!]_{a+\ell} = \overline{w} \qquad f \;\text{ fresh}\end{array}}{\begin{array}{c}\mathcal{R}_k[a, \ell, x = v.\mathtt{m}(\overline{v}) \;;\; s]\; \beta(N, A, p, \overline{q}, \mathit{Upd}) \\ \to \mathcal{R}_k[a, \ell, x = f \;;\; s]\; \beta(N, A, p, \overline{q} :: (f, m, \overline{w}), \mathit{Upd})\; f(\bot)\end{array}}$$

Invk-Self

$$\frac{\begin{array}{c}[\![v]\!]_{a+\ell} = \alpha \qquad [\![\overline{v}]\!]_{a+\ell} = \overline{w} \qquad f \;\text{ fresh}\end{array}}{\begin{array}{c}\alpha(N, A \uplus [k \mapsto a], q : \mathcal{C}_k[\ell, x = v.\mathtt{m}(\overline{v}) \;;\; s], \overline{q}, \mathit{Upd})\; \mathit{cn} \\ \to \alpha(N, A \uplus [k \mapsto a], q : \mathcal{C}_k[\ell, x = f \;;\; s], \overline{q} :: (f, m, \overline{w}), \mathit{Upd})\; \mathit{cn}\; f(\bot)\end{array}}$$

Figure 4.3 – Semantics of BSP active objects – Part 1

SERVE

$$\frac{\text{bind}(\alpha, m, \overline{v}) = \{\ell | s\} \qquad i = \mathit{head}(N)}{\alpha(N, A, \varnothing, (f, m, \overline{v}) :: \overline{q'}, \mathit{Upd})\ cn \to \alpha(N, A, (f, m, \overline{v}) : (\varnothing\ ;\ i \mapsto \{\ell | s\}), \overline{q'}, \mathit{Upd})\ cn}$$

BSPRUN

$$\frac{\text{bind}(\alpha, m, \varnothing) = \{\ell' | s'\}}{\begin{array}{c}\alpha(N, A, q : (\varnothing\ ;\ i \mapsto \{\ell \mid \texttt{BSPrun}(m)\ \texttt{;}\ s\}), \overline{q'}, \mathit{Upd})\ cn \\ \to \alpha(N, A, q : ([k \mapsto \{\ell' | s'\} | k \in [0..N-1]]\ ;\ i \mapsto \{\ell \mid s\}), \overline{q'}, \mathit{Upd})\ cn\end{array}}$$

RETURN-VALUE

$$\frac{[\![v]\!]_{A(i)+\ell} = w}{\alpha(N, A, (f, m, \overline{w}) : (\varnothing; i \mapsto \{\ell \mid \texttt{return}\ v\ \texttt{;}\ s\}), \overline{q}, \mathit{Upd})\ f(\bot)\ cn \to \alpha(N, A, \varnothing, \overline{q}, \mathit{Upd})\ f(w)\ cn}$$

RETURN-SUB-TASK

$$\begin{array}{c}\alpha(N, A, q : \left(\left[\overline{i} \mapsto \overline{\mathit{Task}}\right] \uplus [k \mapsto \{\ell \mid \texttt{return ; } s\}]\ ;\ j \mapsto \mathit{Task}'\right), \overline{q}, \mathit{Upd})\ cn \\ \to \alpha(N, A, q : \left(\left[\overline{i} \mapsto \overline{\mathit{Task}}\right]\ ;\ j \mapsto \mathit{Task}'\right), \overline{q}, \mathit{Upd})\ cn\end{array}$$

SYNC

$$\frac{A' = \left[j \mapsto A(j)\left[(y \mapsto [\![v]\!]_{A(i)}) | (i, v, j, y) \in \mathit{Upd}\right]\right]}{\begin{array}{c}\alpha(N, A, q : \left(\left[\overline{k} \mapsto \overline{\{\ell_k | \texttt{sync ; } s_k\}}\right]\ ;\ i \mapsto \mathit{Task}\right), \overline{q'}, \mathit{Upd})\ cn \\ \to \alpha(N, A', q : \left(\left[\overline{k} \mapsto \overline{\{\ell_k | s_k\}}\right]\ ;\ i \mapsto \mathit{Task}\right), \overline{q'}, \varnothing)\ cn\end{array}}$$

BSP-GET

$$\frac{[\![v]\!]_{a+\ell} = i}{\begin{array}{c}\mathcal{D}_k[a, \ell, \texttt{bsp\_get}(v, x_{src}, x_{dst})\ \texttt{;}\ s, \mathit{Upd}] \\ \to \mathcal{D}_k[a, \ell, s, \mathit{Upd} \cup (i, x_{src}, k, x_{dst})]\end{array}}$$

BSP-PUT

$$\frac{[\![v]\!]_{a+\ell} = i \qquad [\![v_{src}]\!]_{a+\ell} = v'}{\begin{array}{c}\mathcal{D}_k[a, \ell, \texttt{bsp\_put}(v, v_{src}, x_{dst})\ \texttt{;}\ s, \mathit{Upd}] \\ \to \mathcal{D}_k[a, \ell, s, \mathit{Upd} \cup (k, v', i, x_{dst})]\end{array}}$$

Figure 4.4 – Semantics of BSP active objects – Part 2

execution. The syntax of configurations and runtime terms is defined in Figure 4.2. Statements and expressions are the same as in the static syntax except that they can contain runtime values.

A runtime configuration is an unordered set of active objects and futures where futures can either be unresolved or have an associated future value. An active object has a name α and a number N of processes involved in α; these processes are numbered $[0..N-1]$. One of these processes is the head process of the active object; it is always the same for every active object. The function $\mathit{head}(N)$ returns the process identifier of the head process of an active object among its N processes. Note that in the examples of our figures, the head process is always the process 0: we will then assume $\mathit{head}(N) = 0$. A associates each pid i to a set of field-value pairs a. It has the form $(0 \mapsto [x \mapsto \texttt{true}, y \mapsto 1], 1 \mapsto [x \mapsto \texttt{true}, y \mapsto 3])$ for example meaning that the object at pid 0 has two fields x and y with value `true` and 1, and the object at pid 1 has the same fields with different values. Note that the object has the same fields in every pid. The function $A(i)$ allows us to select the element a at position i. $\overline{q}$ is the request queue of the active object. The active object might be running at most one request at a time. If it is not running a request, then $p = \varnothing$; otherwise $p = q : \left([\overline{i} \mapsto \overline{\mathit{Task}}]\ ;\ j \mapsto \mathit{Task}\right)$, where q is the identity of the request being served, and $\left([\overline{i} \mapsto \overline{\mathit{Task}}]\ ;\ j \mapsto \mathit{Task}\right)$ is a two level mapping of processes to tasks that have to be performed to serve the request. The first level represents parallel execution, it maps process identifiers to tasks, the second represents sequential execution and contains a single process identifier and task. We use this generic statement hierarchy to remember the pid associated with the sequential execution and to define rules for statements in both levels using a reduction context. Tasks in each process consist of a local environment ℓ and a current statement s. For example, $p = q : ([k \mapsto \{\ell_k | s_k\} | k \in [0..N-1]]\ ;\ i \mapsto \{\ell \mid s\})$ means that the current request q first requires the parallel execution on all processes of $[0..N-1]$ of their statements s_k in environments ℓ_k; *then* the process i will recover the execution and run the statement s in environment ℓ. Concerning future elements, these have two possible forms: $f(\bot)$ for a future being computed, or $f(w)$ for a resolved future with the computed result w.

Figure 4.5 illustrates a runtime configuration, with a focus on one active object α which has two processes. As stated before, the head process is always the process `0`; This active object has

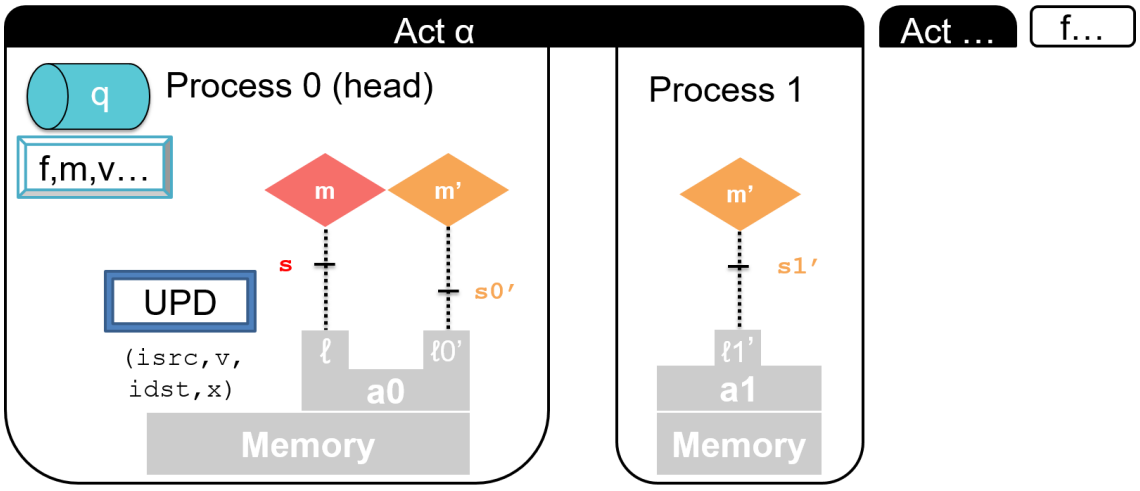


Figure 4.5 – ABSP runtime configuration

a sequential method m and a parallel method m', here we can see each sequential and parallel execution contexts associated to each of these functions. Each process has its own memory; to differentiate the state of variables in each process's memory, we write them with a process number in our illustrative figures. Thus variables in $\ell 0'$ and $a0$, owned by process 0, may have a different state than the variables $\ell 1'$ and $a1$, which are owned by process 1. We can also see that variables in ℓ are only accessible from the sequential execution (they are not explicitly written as $\ell 0$ for this reason) and variables in $\ell 0'$ and $\ell 1'$ are only accessible from the parallel execution. Variables in $a0$ and $a1$, which correspond to object variables, are accessible from both execution contexts. The request queue q holds triplets $(f, m, v...)$, where each triplet corresponds to a request associated to a future f, corresponding to the execution of method m with arguments v.... One such triplets can be in a dedicated field representing the request being processed at a given time. We also show the *Upd* field that holds BSP requests, with each request as a quadruplet (i_{src}, v, i_{dst}, x), requesting to copy the value of variable v as evaluated by process i_{src} into variable x of process i_{dst}. Other similar active objects may exist side by side, and futures are not associated to any particular active object because we consider that the decision about where to store the future value to be implementation-related.

We adopt a notation inspired from reduction contexts to express concisely a point of reduction in an ABSP configuration. A reduction context in general is an expression with a hole, the definition of the valid reduction contexts is convenient for defining the point of execution in a configuration. A global reduction context $\mathcal{R}_k[a, \ell, s]$ is a configuration with four holes: a process number k, a set a of object fields, a local store ℓ, and a statement s. It represents a valid configuration where the statement s is at a reducible place, and the other elements can be used to evaluate the statement. This reduction context uses another reduction context focusing on a single request service and picking the reducible statement inside the current tasks. This second reduction context $\mathcal{C}_k[\ell, s]$ will allow us to conveniently define rules evaluating the current statement in any of the two execution levels, it provides a single entry for two possible options: the sequential level and the parallel one. Note that this reduction context also defines that the parallel level is picked first instead of the sequential one if it is not empty. The two reduction contexts are defined as follows (the variables that are parameters of the reduction context should not appear anywhere else in the terms: they are fresh variables):

$$\mathcal{R}_k[a, \ell, s] ::= \alpha(N, A \uplus [k \mapsto a], q : \mathcal{C}_k[\ell, s], \overline{q}, \mathit{Upd})\ \mathit{cn}$$
$$\mathcal{C}_k[\ell, s] ::= (\varnothing\ ;\ k \mapsto \{\ell | s\}) \quad | \quad \left([\overline{i} \mapsto \overline{\mathit{Task}}] \uplus [k \mapsto \{\ell | s\}];\ j \mapsto \mathit{Task}\right)$$

Let us take the assignment rule as example, it applies in two kinds of configurations: $\alpha(N, A \uplus [k \mapsto a], q : \left([\overline{i} \mapsto \overline{\mathit{Task}}] \uplus [k \mapsto \{\ell | x = e\ ;\ s\}]; j \mapsto \mathit{Task}\right), \overline{q}, \mathit{Upd})\ \mathit{cn}$ and $\alpha(N, A \uplus [k \mapsto a], q : (\varnothing\ ;\ k \mapsto \{\ell | x = e\ ;\ s\}), \overline{q}, \mathit{Upd})\ \mathit{cn}$. Using contexts both greatly simplifies the notation and spares us from having to duplicate rules.

To help defining DRMA operations, we will also use $\mathcal{D}_k[a, \ell, s, \mathit{Upd}]$, which is an extension of $\mathcal{R}_k[a, \ell, s]$ exposing the *Upd* field. It is defined as:

$$\mathcal{D}_k[a, \ell, s, \mathit{Upd}] ::= \alpha(N, A \uplus [k \mapsto a], q : \mathcal{C}_k[\ell, s], \overline{q}, \mathit{Upd})\ \mathit{cn}$$

We use the notation $[\overline{i} \mapsto \overline{\mathit{Task}}] \uplus [k \mapsto \{\ell | s\}]$ to access and modify the local store and current statement of a process k. Just as a statement can be decomposed into a sequence $s; s'$ with the

associative property, the task mapping can be decomposed into $[\tilde{i} \mapsto \overline{Task}] \uplus [k \mapsto Task]$, we use the disjoint union $\uplus$ to work on a single process disjoint from the rest.

The first three rules of the semantics define an evaluation operator $[\![e]\!]_\ell$ that evaluates an expression e using a variable environment ℓ. We rely on $\mathrm{dom}(\ell)$ to retrieve the set of variables declared in ℓ. While these rules may involve a single variable environment, we often use the notation $a + \ell$ to involve multiple variable environments, e.g. $[\![e]\!]_{a+\ell}$. As stated above, we assume for simplicity that there are no variables with the same name on a and ℓ; this means we do not have to define what it means to access a variable declared in both object and local scopes. It is important to note that $[\![e]\!]_{a+\ell} = w$ implies that w is not a variable, it can only be an object or future name, `null`, or an integer value.

New creates a new active objects on N processes with parameters v, used to initialize object fields. We use $fields(Act)$ to retrieve names and rely on the declaration ordering to assign values to the right variables. We also add a unique process identifier and N, respectively as *pid* and *nprocs*. The new active object β is then initialized with N processes and the resulting object environment A.

Assign is used to change the value of a variable. The expression e is evaluated using the evaluation operator, producing value w. Since variable x can either be updated in the object environment a or the local variable environment ℓ, we use the notation $(a + \ell)$ to represent the unified environments. We can then use the notation $(a + \ell)[x \mapsto w] = a' + \ell'$ to update either of these environments and retrieve both updated environments as a' and ℓ'. In this rule, a value is assigned to a variable that can be in either of these environments, They replace old ones in the object configuration.

If-True and If-False reduces an `if` statement to $s1$ or $s2$ according to the evaluation of the boolean expression v.

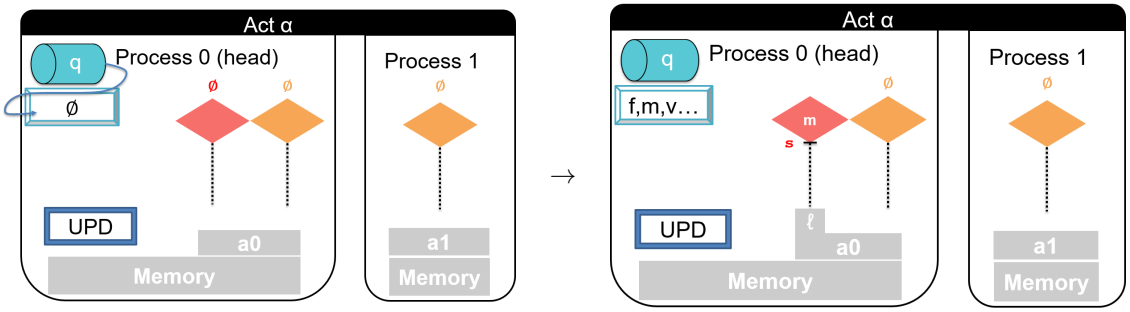
Get retrieves the value associated with a future f. If the future has been resolved with value w, the get statement is replaced by w, which is the value that was assigned to the future in the Return-Value rule. If the future was not resolved, then this rule can not be applied, and the α active object cannot execute the `get` statement as this operation is not defined for unresolved futures: the active object is blocked.

Invk invokes the method `m` of an existing active object and creates a future associated to the result. This rule requires v to be evaluated into an active object β then enqueues a new request in this object.

A new unresolved future f is added to the configuration. Parameters $\overline{v}$ that are passed to the method are evaluated locally, into $\overline{w}$. The request queue of the active object β is then appended with a triplet containing a new future identifier f associated to the request, the method m to call and the parameters $\overline{w}$.

Note that the execution of this rule requires the called active object β to be different than the current active object which makes the call. The Invk-Self rule described allows the current active object to make an active object call on itself

Invk-Self invokes the method `m` of the calling active object and creates a future associated to the result. This is a simple adaptation of the Invk rule above; the purpose of this rule, compared to the previous one, is to allow self-invocation. The variable v must be evaluated to the current active object (and not a different one as the Invk rule). The fields are updated as in the previous rule, except that these are the fields of the calling active object.



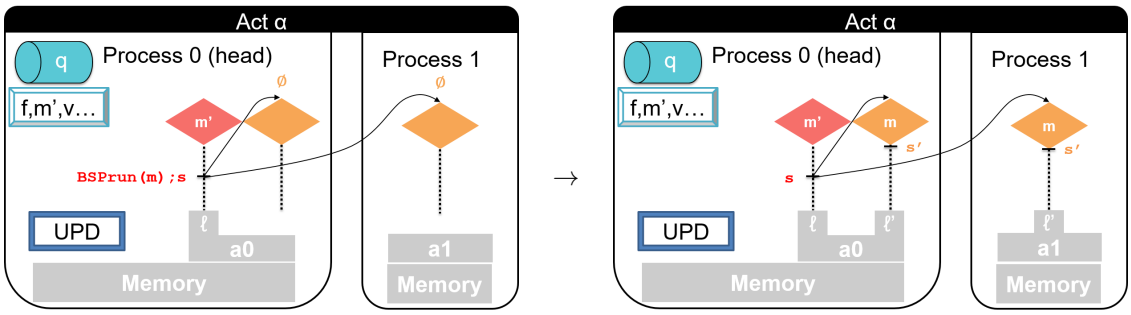
(a) Initial configuration

(b) Result configuration

Figure 4.6 – Serve rule

Serve processes a queued request. Figure 4.6 illustrates this rule. To prevent concurrent execution of different requests, the active object is required to be idle (with the current request field empty). A request $(f, m, \overline{v})$ is dequeued to build and execute a new sequential environment $i \mapsto \{\ell|s\}$, while the parallel execution context is initialized empty. BSP active objects start executing functions

sequentially on the head process, so the sequential environment is assigned to the head process of the α active object ($i = head(N)$). The bind function initializes the environment by building a task, i.e. local environment ℓ and statement s, from the method name m and argument list $\overline{v}$. This rule enables an active object to process a request, as opposed to the Invk rules that are for creating and queuing a request.



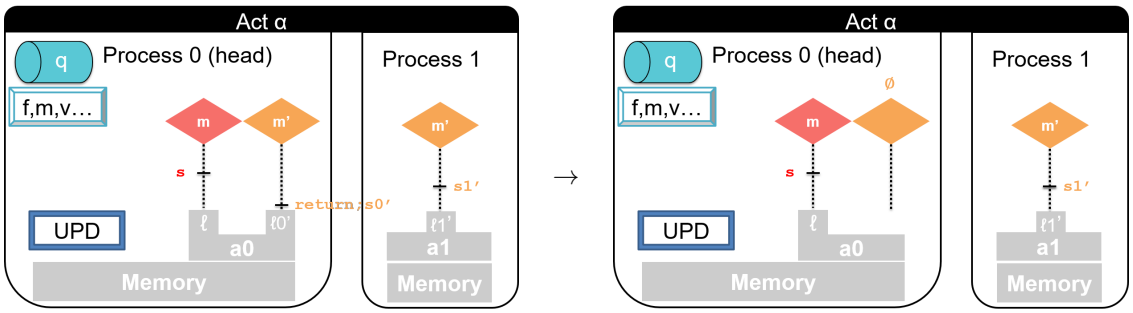
(a) Initial configuration (b) Result configuration

Figure 4.7 – BSPrun rule

BSPrun starts a new parallel environment from the current active object α and the method m. This rule is for turning a sequential environment into a parallel environment through a function call. Figure 4.7 illustrates this rule, where we can see the parallel execution context being created for running method m in parallel with all the processes of actor α. Every process of the active object is going to be responsible for executing one instance of the same task $\{\ell'|s'\}$. All parallel processes start with the same local variable environment ℓ' and the same statement s' to execute. This task is used to initialize the parallel environment of the active object α. Every process is going to be responsible for executing one instance of this task. This is why all parallel processes start with the same local variable environment and the same statement to execute. Like the previous rule, *BSPrun* starts a method execution, but of a parallel function with no parameter.

Return-Value resolves a future. The expression v is first evaluated into a value w that is associated with the future f. The current request field is emptied, allowing a new request to be processed. This rule describes how an active object function returns a value from a return statement; it rule only applies to the sequential context. The expression v is first evaluated into a value w. Since

we did not use a reduction context, we use $A(i)$ to select the object variable environment of pid i. Then the future associated to the request is given the w value. The current request element is then set to empty so that other requests can be served.



(a) Initial configuration (b) Result configuration

Figure 4.8 – Return-Sub-Task rule

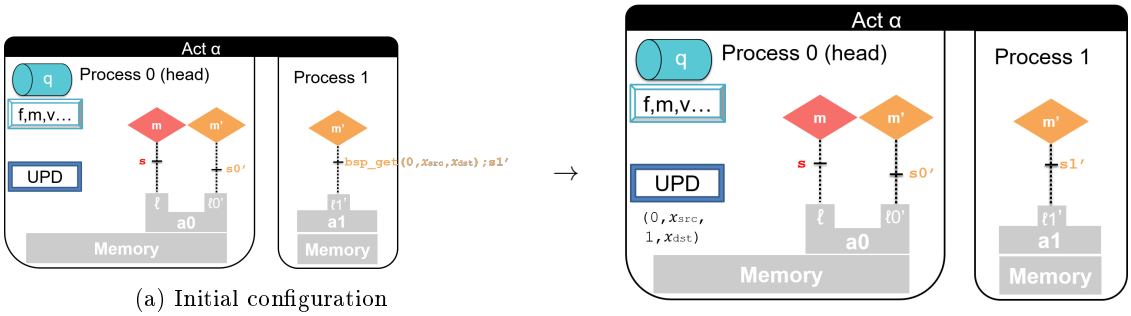
Return-Sub-Task terminates one parallel task. Figure 4.8 illustrates this rule. The process that performs the `return` is removed from the set of tasks running in parallel. When the last process is removed from this set, the sequential context can be evaluated.

Figure 4.8 shows the head process returning from its parallel execution context. When that happens, its associated parallel execution context is removed. A `return` statement by one process only terminates this process's execution context. Here, process 1 is still active, so the sequential execution context may not resume, even if the head process finished its parallel execution context (executing the parallel function). When all processes have returned, with their parallel execution contexts set to $\varnothing$, by this rule and the triggering `return` statement, sequential execution may resume. This rule is the opposite of *BSPrun*, it is used to switch from parallel to sequential execution by exiting from the function called by *BSPrun*. Note that we do not allow sub-processes to return a value from the parallel execution context, because we do not want to define what it would mean to have return values from parallel to sequential execution contexts.

Sync ends the current superstep, the `sync` statement must be reached on every pid k of the parallel execution context before this rule can be reduced. DRMA operations that were requested since the last superstep and stored in the *Upd* field as (i, v, j, y) quadruplets are taken into account. They

are used to update the object variable environment A into A' such that variable y of pid j is going to take the value v as evaluated in process i, for every such quadruplet. As *Upd* is an unordered set, these updates are performed in any order.

BSP-Get requests to update a local variable with the value of a remote one. We write a DRMA quadruplet such that the variable x_{src} of the remote process i is going to be read into the variable x_{dst} of the current pid k during the next synchronization step. For example, if the statement `bsp_get(1,x,y)` was executed from process `2`, it is going to create the quadruplet $(1, x, 2, y)$, which means the value of variable `x` on process `1` has to be written into the variable `y` of process `2`.



(a) Initial configuration

(b) Result configuration

Figure 4.9 – BSP-Get rule

Figure 4.9 illustrates the evaluation of this rule. Process 1 executes `bsp_get` to assign the value of the variable x_{src} of process 0 be copied into its variable x_{dst}. This operation is simply remembered as a quadruplet of the UDP field. Note that here, process 1 executes `bsp_get` in the parallel execution context, but it could have been process 0 and even in the sequential execution context.

BSP-Put requests to write a local value into a remote variable. The value to be written is evaluated into v, and a new update quadruplet is created in *Upd*. Just as `bsp_get`, it will be taken into account upon the next `sync`. For example, if the statement `bsp_put(1,x,y)` was executed from process `2` and its `x` variable has `42` as value, it is going to create the quadruplet $(2, 42, 1, y)$, which means the value `42` has to be written into the variable `y` of process `1`. Note that here, remembering that process `2` issued the operation is not important, but this generic structure allows us to keep

the get and put operations into the same UPD field.

Determinism While race conditions exist in ABSP, like in active object languages and in BSP with DRMA, the language has no data race. Indeed, the only race conditions are message sending between active objects, and parallel emission of update requests. The first one results in a non-deterministic ordering in a request queue, and the second in parallel accumulation of update orders in an unordered set. Updates are performed in any order upon synchronization but additional ordering could be enforced, e.g. based on prioritized pids as suggested by [62].

4.4 Example

In this section, we present a simple example using our ABSP formal language. This example features a BSP active object which has an object variable `_val` on each process (with a different value for every process), and a function which does two things:

1. every process `i` sets the value of its `_val` variable to the value of `_val` of its neighboring process on the right (`i+1`),

2. returns the value of `_val` which the head process has.

We assume we have basic arithmetic operations such as addition and modulus in order to compute the neighboring process identifier.

Figure 4.10 features the OffsetActor, which has functions `offset` and `bsp_offset`. The former is meant to be called as an active object function and the latter is meant to be executed in parallel through *BSPrun*.

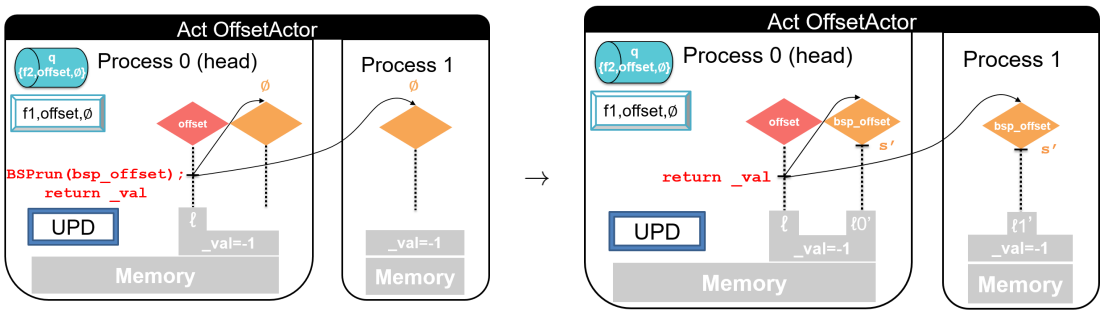
The `offset` function simply calls *BSPrun* with `bsp_offset` as parameter, and returns the integer value stored in `_val`. The `bsp_offset` parallel function first initializes the `_val` variable in the first call (we do not have sophisticated constructors to initialize different values for each process at construction-time), then each process takes the value of `_val` from its right neighbor (circularly) through a `bsp_get` call.

```
1  Act OffsetActor
2  {
3      int _val
4
5      void bsp_offset()
6      {
7          if (_val == -1)
8          {
9              _val = pid; // numbered 0..nprocs-1
10         }
11         else
12         {
13             skip;
14         }
15
16         bsp_get((pid + 1) % nprocs, _val, _val);
17         sync();
18         return;
19     }
20
21     Fut<int> offset()
22     {
23         BSPrun(bsp_offset);
24         return _val;
25     }
26 }
27
28 {
29     OffsetActor act
30     Fut<int> f1
31     Fut<int> f2
32     Int x
33     Int y
34
35     act = new OffsetActor(2,-1);
36
37     f1 = act.offset();
38     f2 = act.offset();
39
40     x = get f1;
41     y = get f2;
42 }
```

Figure 4.10 – ABSP example

The main function (that is unnamed), creates an instance of this active object with two processes and initialize the `_val` field at `-1` to mark it as uninitialized (and triggers its initialization inside `bsp_offset`). Then the `offset` function is called twice in a row and the resulting values are synchronized into variables `x` and `y`, with respectively values `1` and `2`.

Figure 4.11 illustrates the instantiation in the first call to `BSPrun` in the example of Figure 4.10



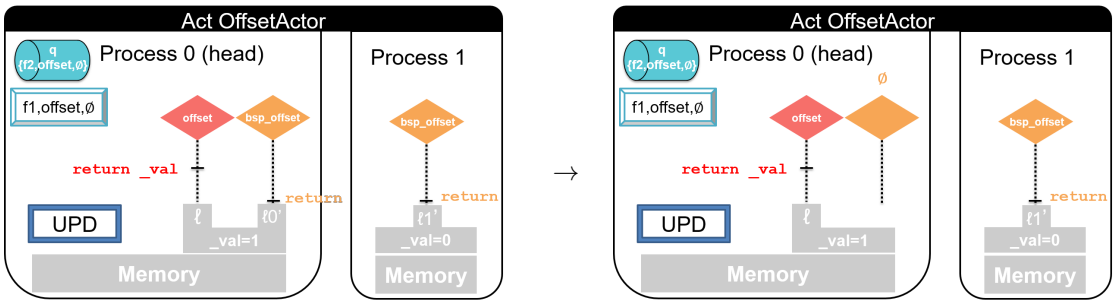
(a) Initial configuration

(b) Result configuration

Figure 4.11 – Instantiation of first *BSPrun*

(line 23). Note that for clarity in the figures illustrating this example, we only show the state of the `_val` variable in the $a0$ and $a1$ variable environment; however they also contain the *pid* and *nprocs* variables initialized in the New rule when creating the active object. At the time of this call, the `bsp_offset` function `BSPrun` has as parameter was never entered; this means the `_val` variable in the a field still has a value of `-1` that was assigned for every process at the object creation time (line 35). We can see in Figure 4.11 that the current request field is set to $(f1, offset, \varnothing)$; this means that the request being handled is associated to the `f1` future, for executing the `offset` function (the one the head process is currently in), with no parameter. We can also see that there is another request in the queue: $(f2, offset, \varnothing)$; this is a similar request, but associated to the $f2$ future. Note that due to the asynchronous nature of parallel processes, it may also be the case that another statement execution scheduling would have resulted in the main function not queuing this second request yet (if the main function did not reach line 38), but here we assume it did for illustrative purpose. In the result configuration, after the call to `BSPrun`, the parallel execution context is initialized so that the statements s' correspond to those of the `bsp_offset` function; we do not show these statements as they would take too much space in this figure. We can see that there is only `return _val` left to execute in the sequential execution context; as the parallel execution has priority, the parallel execution context has to disappear on all processes before executing this last statement.

Figure 4.12 shows the first instantiation of the `return` statement inside the `bsp_run` function



(a) Initial configuration (b) Result configuration

Figure 4.12 – Instantiation of first *return* from *bsp_offset*

(line 18). This happens after the scenario of Figure 4.11 above and after the execution of the core of `bsp_offset`. We assume that both processes executed the rest of the `bsp_offset` function and reached the `return` statement at the same time. Firstly, and because it was the first time this function was called, the `bsp_offset` function made every process set the value of the `_val` variable to the value of their pid; since here the object has two processes, the process of pid `0` has `_val` set to `0` and the process of pid `1` has `_val` set to `1`. Secondly, all processes gave their value of `_val` to the process having a successive pid value (or to process `0` for the process of pid `nprocs - 1`); since here there are only two processes, we can consider they simply permuted the value of their `_val` variable; this means that now, after the execution of the core of the `bsp_offset` function, process `0` has `_val` set to `1` and process `1` has `_val` set to `0` in the initial state of Figure 4.12. In this figure, both processes reached the `return` statement; this means Return-Sub-Task rule is applicable to both processes, but here we chose arbitrarily to instantiate the `return` statement of the process `0`. We can see the parallel execution context disappearing in the result configuration. Now what is left is only the return statement of process `1` before going back to the sequential execution context and returning the `_val` value of `1`.

We illustrated the transition between sequential and parallel execution through `BSPrun` and the `return` statement within the parallel function that was called. The `f1` future is then associated with the value `1`. The next call to `offset`, associated to the `f2` future, will then be associated to the value `0` because the `_val` values are going to be permuted again in the `bsp_offset` function.

Through this example we illustrated the syntax of our ABSP language and the instantiation of two key rules in our semantics.

4.5 Conclusion

In this chapter, we have provided a formal semantics for our BSP active object model presented in Chapter 3. This semantics joins active object semantics and simple BSPlib-like DRMA semantics through a central *BSPrun* primitive. While we do not derive formal properties due to lack of time and a focus on the distributed futures in Chapter 5 and implementation aspects, this semantics would be a good candidate to do so. The deterministic property (under similar conditions to active objects) would be an interesting property to prove. We chose semantics close to the BSPlib specification, which as explained in Section 2.3, is not deterministic, but would be easy to implement in a deterministic way. Similar changes could be made, for example, enforcing an order on the execution of DRMA operation instead of having an unordered set.

Chapter 5

Distributed Futures

5.1 Introduction

In this chapter, we introduce the notion of distributed futures, a concept of futures that is appropriate to manipulate distributed data. They enable their efficient transfer between data-parallel tasks and provide a unified view of distributed data through a single future.

This chapter is organized as follows: we explain in Section 5.2 why the classical futures are not appropriate for manipulating distributed data, before introducing our notion of distributed futures in order to solve these issues. In Section 5.3, we then discuss some high-level implementation aspects we had to address in order to implement distributed futures within our BSP active objects implementation, including its internal structure and the primitives we introduced to manipulate distributed futures . We then give an example usage within our implementation in Section 5.4, before concluding in Section 5.5.

5.2 Motivation and principles

In this section, we explain the concept of a distributed future. We will see how distributed futures can be used to enable a more efficient communication between parallel tasks that use futures, by

parallelizing the communication. We will also see that distributed futures allow the programmer to have a more unified view of distributed data, as opposed to a solution that would be based on arrays of futures.

The idea of distributed futures is not only relevant in the context of our BSP active objects, but also to any parallel framework mixing task and data parallelisms by using futures to communicate distributed results between parallel tasks. This is why, in this section, we discuss distributed futures in a rather abstract manner; they are discussed in the context of parallel tasks: with just the concepts of multiple tasks involving multiple processes each. As we have seen in Chapter 3, each of our BSP active object can be used to run a parallel task; this is how the discussions in this section about parallel tasks can be related to BSP active objects.

We first discuss what led us to design distributed futures in Section 5.2.1, then we discuss the principles of distributed futures in Section 5.2.2.

5.2.1 Motivation

To incorporate data-parallelism inside a task-parallel framework, the most efficient solution is to use multiple processes inside a single entity to handle each task, like BSP tasks in our active BSP model. If these tasks are synchronized by futures, the result of each data-parallel computation is required to be returned as a single return value in order to store it in the future that corresponds to the task result. This single return value could be a complex collection – e.g. an array, a vector – but it must be manipulated as a single future and transmitted in its entirety upon future access. 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, even if it was distributed among processes. This is the strategy we applied in the previous chapters.

This gathering however raises a performance issue whenever the result is a large array, especially when this array is passed to another task that scatters it again to do another data-parallel processing.

Figure 5.1 illustrates this problem for two parallel tasks `Task1` and `Task2`, which may respectively correspond to two BSP active objects. `Task1` has data that is distributed among its processes, and

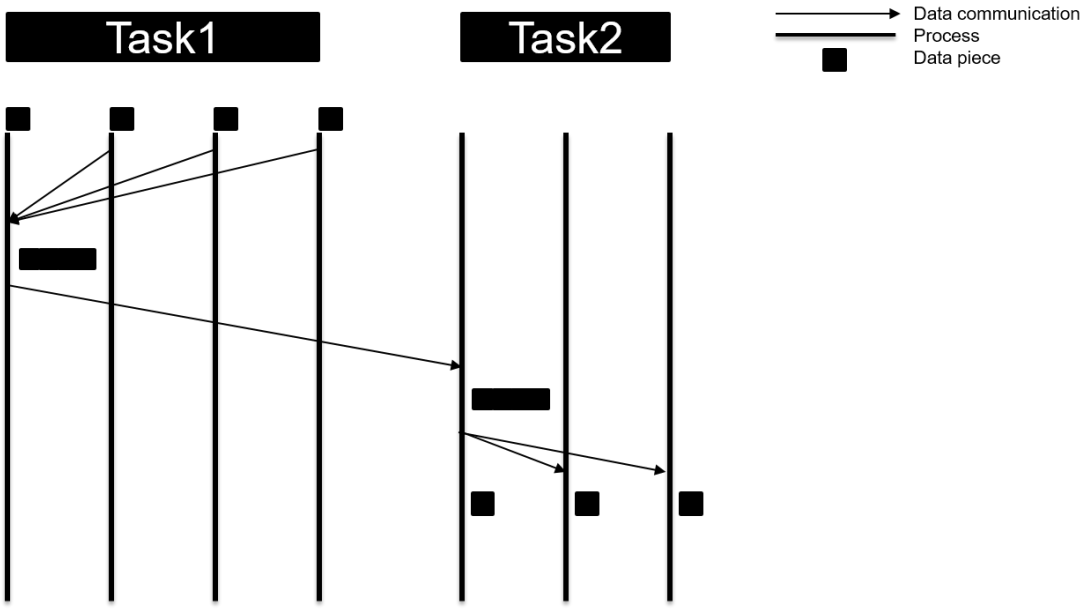


Figure 5.1 – Gathering and scattering to transfer distributed data

`Task2` requires this data to be distributed among its own processes. In order to transfer the data, `Task1` gathers all the distributed data pieces in the memory of a single of its processes. This process then communicates the full data to a single process of `Task2`, which then scatters the data so that it is distributed among all of its own processes.

A more efficient way to transfer the data between the parallel tasks would be that every data-parallel process keeps its part of the result, and transmits it directly where it is needed instead of gathering the data in order to transfer it. We will see in the following section how we propose to do so using distributed futures.

5.2.2 Principles

The purpose of a future is to *represent* the result of a task that is being computed, and to provide a way for a process to synchronize on the availability of this result. The representation of this result can be passed around to any other task if this future is a first-class future, enabling other processes to synchronize on the availability of this result and to receive its data. When the result

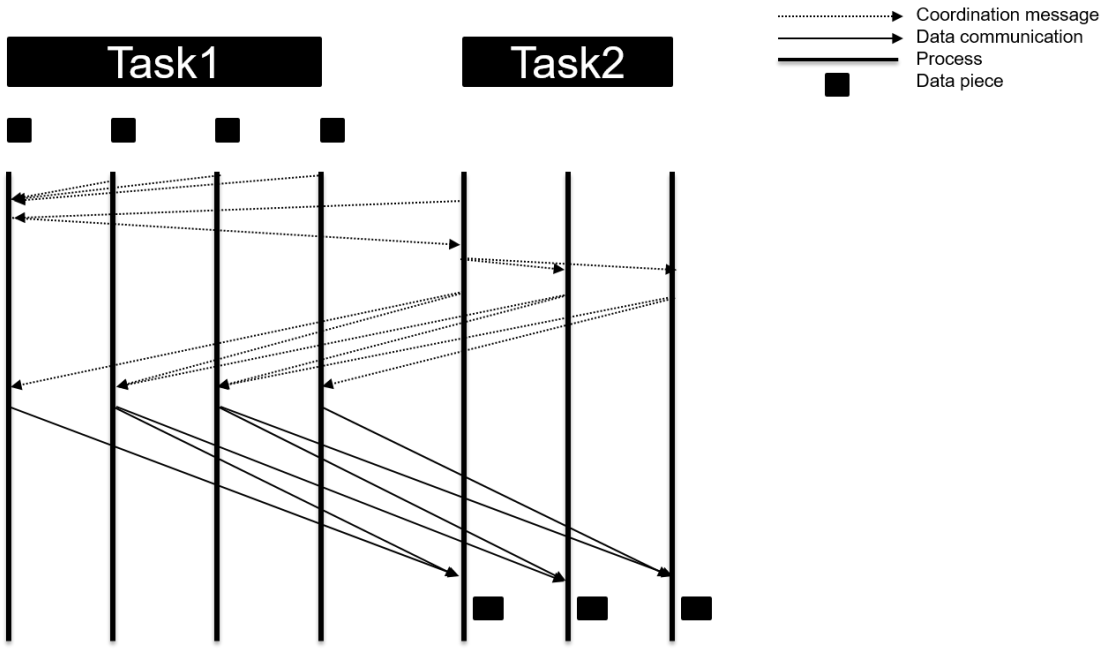


Figure 5.2 – Parallel transfer of distributed data

of a task is a distributed data, we would like to have a future that still represents this distributed data, but where we have more control over how the data itself is communicated on request, over parallel processes. What we do not want, is to move distributed data needlessly; instead of having every process in a parallel task gather its data piece to a central process, we choose to have these processes keep their data piece, but create a description of this piece. The descriptions of all the pieces are then gathered from each process of the parallel task to a single process among them in order to form a description of all the pieces: a description of the distributed collection of data. We now refer to this distributed collection of data as a distributed vector. This description of the distributed vector can then be returned as a future, and sent around between processes and/or be used to request specific data pieces (only those required).

We call this description a *vector distribution* because it *represents* a *distributed vector* being computed. This vector distribution may only be known after a data-parallel computation. This is why calling a parallel task that creates a distributed vector first returns a future of vector

distribution. We call this future a *distributed future*, because it represents a distributed data, through its vector distribution, 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 (and not the actual data of the vector). Synchronizing this distributed future enables the programmer to synchronize on the availability of the distributed data, without having to transfer the data right away. Instead of having a future which is a placeholder for the data, the future is a placeholder for the vector distribution. The details of the vector distribution are not exposed to the programmer, but its content can then be used by the programmer, through a dedicated `get_part` primitive, to fetch the data parts necessary for the computation on the current processes. This primitive is distinct from the `get` on the distributed future, and transfers only a part of the distributed data to the process that performs the `get_part`.

Provided the size of a 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 needed parts of the content of the distributed vector, directly from the processes that hold them. Moreover, to obtain the computed data, the programmer does not have to know where each part is located and on how many processes each part is distributed.

This general principle is illustrated by Figure 5.2, where `Task1`, instead of gathering all the data pieces to a single process, gathers only the description of these pieces. When a process of `Task2` synchronizes the corresponding distributed future, this description is sent to `Task2` as a vector distribution. This process then broadcasts it to all the other processes of `Task2` so that they can freely request the pieces they need. Each process, from the vector distribution, is then able to retrieve any part of the distributed vector, no matter on which process(es) this part is located.

When each process individually requests a piece, the distributed data is effectively transferred in parallel between the needed processes of `Task1` and the needed processes of `Task2`. In our implementation, this operation is not a collective: individual processes may request pieces independently, and at a different time, from the rest of the processes. Note that it is also possible that only one process requests a part (that is not even the whole distributed vector), without the other processes requesting any part. However, a different version could implement a more coordinated transfer

through a collective communication operation.

In this figure, the processes of `Task1` are more numerous than the processes of `Task2`. Because the same distributed data is evenly distributed on fewer processes in `Task2`, this means the size of each part owned by each process of `Task2` is larger than the size of each part owned by each process of `Task1`. This is why an individual data transfer initiated by a process of `Task2` does not just involve communication from a single process to another; in this case, each process of `Task2` receives data pieces from two processes of `Task1`. In the example, each processes requests a different part of the distributed data. For example, the second process of `Task2` only needs a part of the data stored by the second process of `Task1`, the other part is requested by the first process of `Task2`; the second process of `Task2` will only receive the data it needs.

In short, distributed futures are more efficient as they enable parallel communication between the different processes that handle different data parts, but also this optimisation is easy to program as the programmer does not have ot be aware of the data distribution. They remove the necessity to gather each part into a single process, transmit it in its entirety, and scatter it again. 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 the metadata that is necessary to access the content of the associated distributed vector. From this metadata, the content of the distributed data can then be transferred by requesting the right part on the right process(es), transparently for the programmer.

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. This is why a lazy synchronization strategy is best-suited, as we do not know in advance which process will need which data.

Overall, a distributed future is at the same time a distributed data structure and a unique descriptor that is both used to perform synchronization like a standard future, and to redistribute the real data to processes that need them.

5.3 Implementation of distributed futures

Our implementation of distributed futures is based on the BSP active object model of described in Chapter 3, and its implementation we will introduce in Chapter 6. To implement the concept of distributed future, two aspects have to be implemented: the future resolution as a vector distribution and the access to the distributed data. We review our solution for both of them below. We start by defining the `vector_distribution` data structure to represent a distributed vector, this data structure is stored in the distributed future when it is resolved.

5.3.1 The `vector_distribution` structure

When our distributed futures are resolved we assign them a collection of (`pid`, `local_id`, `size`, `offset`) quadruplets. We define this structure as a `vector_distribution`, which internally is a list of these 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 size of this part; *offset* is the start index of this part 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, and thus requires a distributed future to track one such part per process.

For example, if we have a block-distributed vector of 40 elements computed by processes 1, 2, 3 and 4, the value of the `vector_distribution` is the list $((1, 1, 10, 0), (2, 12, 10, 10), (3, 41, 10, 20), (4, 33, 10, 30))$, where each `local_id` value is taken arbitrarily. Here we can see that processes own consecutive parts of size 10 each. The second element of the quadruplet is a local identifier that allows each process to identify uniquely the designed part: when requiring the part identified by 12 at process 2, it will return the 10 elements of the distributed vector starting at index 10 even though process 2 does not know what part of the distributed vector this data represents. Note that this definition allows a set of quadruplets to be defined so that there are holes within a distributed vector; for example, if we remove the second quadruplet in the example above. It is useful for the

programmer to know the size of a distributed vector from a `vector_distribution`. For example, this size may be used in order to easily redistribute a distributed vector into a different process group that has a different number of processes. As we will see in Section 5.3.2, the user does not directly manipulate the set of quadruplets in a `vector_distribution` in order to communicate a distributed vector. While the communication primitive is not influenced by holes in the distributed vector, the user is not made aware that a part of a distributed vector that is received has a hole inside. In this work, we are not interested in the possibility of having such holes in distributed vector in order to simplify these aspects. This is why we assume distributed vectors are contiguous, meaning they does not have any such hole. Allowing holes would require our communication primitive to give the user information on these holes, for example through a mapping from received data to index in a distributed vector.

Instead of a part list, we could rely on pre-defined distributions; then the `vector_distribution` would refer to a distribution type along with its parameters, this would be enough to deduce the precise distribution of the distributed vector. However, the location information, i.e. the process IDs, still has to be maintained which makes the number of processes a factor of the `vector_distribution` structure size. Using a part list like ours is generic in the sense that it can represent any distribution, but is not suited when the number of parts is high. For example, the worst case is a cyclic distribution, where each vector element is round-robin distributed among the processes. Every part of the `vector_distribution` represents one element of the distributed vector. In such a case, a pre-defined distribution would involve a structure of size proportional to the number of processes (holding only the `pid` of storing processes) while the size of our distributed futures, given the way we have decided to represent them, is proportional to the size of the vector, which is bigger. Pre-defined distributions are out of scope for this work, but they should be envisioned for distributions where each process stores many parts of a distributed vector.

5.3.2 Language extension

In order to represent the type of data in a distributed vector, we provide a template parameter `T` to the `vector_distribution` structure, which becomes `vector_distribution<T>`. This has no

repercussion on the quadruplet definition defined in Section 5.3.1. This definition allows a safer use of the primitives defined below. In our implementation, distributed futures are represented as futures of `vector_distribution <T>`; we provide a more elegant definition as `DistrFuture<T>`, which is simply an alias for `Future<vector_distribution<T> >`. For example, we can have a `DistrFuture<int>` that represents a distributed vector of `int` elements.

In a data-parallel context where the processes of an active object each produces a part of a distributed vector, our BSP active object implementation allows the programmer to make processes keep vector parts by storing these parts on their management thread. This associates each distributed vector part with a local identifier within this thread, so that they can be queried (e.g. the part whose id is 12 in the example above). The management thread does not need to know more information. In particular, it does not need to know how different parts relate to one another, its role is just to be available for querying. Instead, metadata are assembled into a `vector_distribution`, which is given to the user; this is a list of the quadruplets, as described in Section 5.3.1. The user can use this structure to know which part(s) to query when an arbitrary subpart is required.

We provide 4 main high-level primitives for manipulating `vector_distribution`, shown in Figure 5.4. These primitives all have a template parameter `T`; as we will see below, it serves different purposes, including a better type safety and a more user-friendly computation of array positions. The following description of these primitives is illustrated by the process and communication diagram of Figure 5.3.

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 part within the global distributed vector. This step creates a quadruplet as defined in Section 5.3.1. The `size` and `offset` parameters are quantified in number of `T` elements, not in bytes as the quadruplets of Section 5.3.1. This template function will multiply these parameters by the size of an element `T` when creating a quadruplet.

A call to `gather_vd_parts` by the head process enforces that all the calls to `register_result` finished (by making sure the parallel function is exited by every process), then assembles the quadruplets these calls produced into a `vector_distribution<T>` structure on the head process.

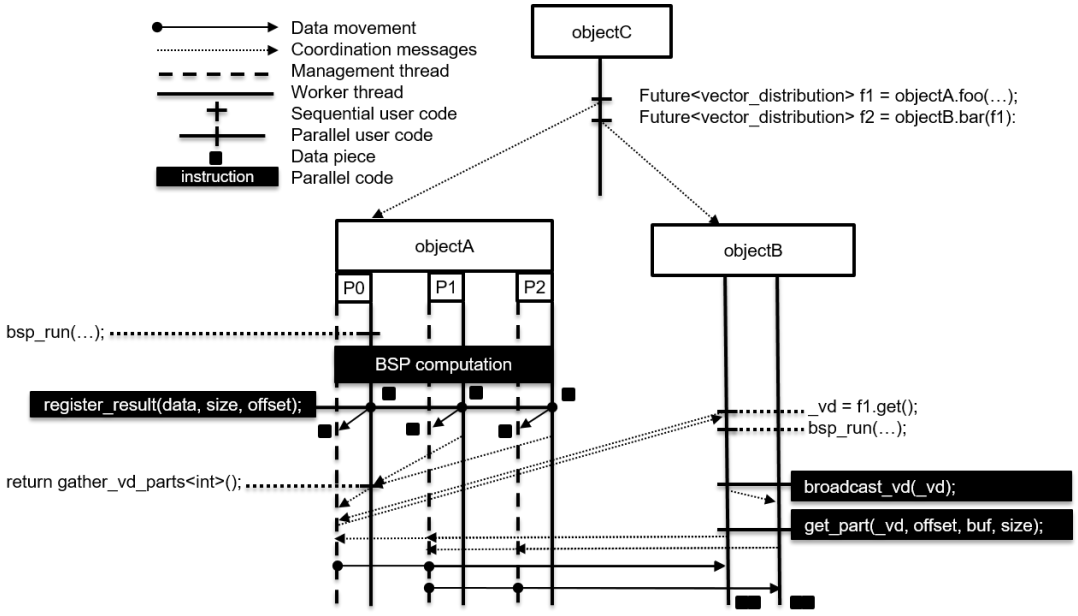


Figure 5.3 – Distributed future API example

```
1  template <class T>
2  void register_result(const T *data, size_t size, size_t offset);
3  template <class T>
4  vector_distribution<T> gather_vd_parts();
5  template <class T>
6  void broadcast_vd(vector_distribution & vd);
7  template <class T>
8  void get_part(const vector_distribution<T> & vd,
9                size_t offset, char *buf, size_t size);
```

Figure 5.4 – Vector distribution primitives

This structure is then given as return value to fulfill the associated distributed future typed as `DistrFuture<T>` (which is a `Future<vector_distribution<T> >`). Note that this is the only function where the template parameter `T` has to be given explicitly by the user, because it can not be deduced automatically by a `vector_distribution<T>` parameter.

In the example of Figure 5.3, the `vector_distribution<int>` returned by `gather_vd_parts<int>` is given to the `return` statement by `objectA`. This allows the `get` statement on `objectB` to proceed, as it requests the associated future value. At this point the synchronization provided by the future is ensured, and the vector distribution is transfered to `objectB`, but the data transfer still needs to be performed (the metadata is transfered, but not the data).

Resolving this future means that the programmer obtains this `vector_distribution<int>`. He triggers this resolution with a usual `get` on this future.

The `broadcast_vd` primitive allows a head process to send this data structure to its other active object processes; a collective call to the `broadcast_vd` primitive within a parallel function sends the `vd vector_distribution<int>` given as parameter into the same parameter of the other processes. The purpose of this primitive is to make the `vector_distribution` structure transparent to the programmer so that it can be communicated without knowing its internal content, which does not have to be exposed.

With the `get_part` primitive any of these process can request a subpart of the distributed vector, where its distribution is transparently deduced from the `vd vector_distribution<T>` given as parameter. The `offset` and `size` parameters implicitly specify the part(s) to be retrieved, without the user knowing on which process(es) the data is. Note that we do not specify any part id: they are not exposed to the programmer; instead we directly specify the range of data that is required. The retrieved part can actually be split over multiple processes, or be contained within one part in one process; `get_part` deduces from the set of quadruplets stored in `vd` from which process(es) the data is to be queried. Note that, as for `register_result`, the `offset` and `size` parameters are quantified in number of elements of type `T`.

Consider the example of Section 5.3.1, if we request a part of the distributed vector at offset 15 and of size 10, `get_part` will ask for one part of size 5 on process 2 (at offset 5 inside the part

locally indexed 12) and another on process 3 (at the beginning of the part locally indexed 41). In Figure 5.3, the first and second processes of `objectB` each requests a half of the distributed vector stored in the memory of the three processes of `objectA`. The first process of `objectB` requests the first half of the distributed vector, which is deduced to be on `P0` and `P1`; meanwhile, the second process of `objectB` requests the second half of the distributed vector, which is deduced to be on `P1` and `P2` of `objectA`. As said earlier, these two `get_part` operations are independent from each other: they are not part of a collective operation. As multiple processes can call this function at the same time, the distributed vector can be transferred in parallel from a BSP active object to another.

In order for a BSP active object to decide how to retrieve a distributed vector in a balanced manner (which process has to receive which part and of what size), it is also useful for the programmer to know the global size of a distributed vector from the `vector_distribution`. This is why we also introduce a `getVecSize` member function in `vector_distribution`. Because we assume there is no hole in a distributed vector, the global size of the distributed vector is obtained form the size and offset of the last quadruplet of a `vector_distribution` (the one with the largest offset). Because the `offset` and `size` elements of our quadruplets are always expressed in bytes, `getVecSize` obtains the number of `T` element in the distributed vector by dividing the total number of bytes by the size of a `T` element. This is so that the returned value is quantified in number of elements of `T` in the distributed vector.

We also added a primitive that enables the programmer to distribute a vector (to create a distributed vector from a normal vector). This primitive only supports block distribution, but the idea could be further extended if distribution policies would be implemented. The signature of this primitive is `vector_distribution<T> block_distribute(std::vector<T> v)`. This primitive may be useful, for example, to avoid going through the head process of an active object when the user wants to distribute a vector. Indeed, in this case, the user would have to create an active object function that takes a vector as parameter, and then scatter each part to a parallel process. With this primitive, the vector is directly scattered without having to be transmitted in full to the head process beforehand, therefore avoiding a data transfer.

5.3.3 Note on implementation choices

In our implementation, we have introduced a `vector_distribution<T>` structure that represents the distribution of a distributed vector among different processes. This structure, when given as return value of an active object function, becomes a distributed future as `DistrFuture<T>` (which is an alias for `Future<vector_distribution<T> >`). Synchronizing this distributed future with a call to `get` returns the associated `vector_distribution<T>`, which may then be used with a `get_part` primitive in order to retrieve a range of the data.

The concepts of `Future` and `vector_distribution` could have been further integrated together. For example, a single primitive called on the distributed future could both synchronize the metadata and retrieve the data, in a manner that is transparent for the user. One issue with such a solution would be that the user may not know in advance the size of the parts each process would then receive. The primitive receiving the data would then have the responsibility to allocate the buffer (or reallocate a provided buffer if it is too small) used to receive the data. Another issue when not knowing the size in advance is that it is not possible for the receiving processes to decide precisely how to distribute the data among themselves (how to cut the distributed vector). This issue may be solved by implementing distributed policies as discussed in Section 5.3.1. For example, the receiving processes may simply specify that they want to receive the data in a block-distributed manner among themselves, thanks to a pre-defined distribution policy that would be recognized by the processes that own the data. This is why exploring this option requires further investigating distribution policies. In such a case, one could imagine a primitive similar to the get_part of Section 5.3.2, but with a signature of `void get_part(const DistrFuture<T> & df, const DistributionPolicy & policy, std::vector<T> & data)`. Contrarily to the version of Section 5.3.2, this primitive directly provides the `DistrFuture<T>` among its parameters, instead of the synchronized `vector_distribution<T>` metadata. Also, this primitive does not provide a precise target memory with `offset` and `size` parameters, but only gives a `DistributionPolicy` object, which must contain information to distribute the data such as the pid of each process. Also notice that the `data` parameter is an `std::vector` object, which may be resized by this `get_part`

primitive if its initial size is not enough to retrieve each part.

The study and implementation of a merged concept of `Future` and `vector_distribution` requires a careful study of distribution policies and communication patterns in order to minimize the synchronizations. While the further development of this idea is beyond the scope of this thesis, we think it would be an interesting improvement and we provided an analysis of its requirement in this section.

5.4 Illustrative example

In this section, we give a simple code example showing the usage of distributed futures within our BSP active object implementation (that will be fully described in Chapter 6).

This example is split into Figure 5.5 and Figure 5.6 for size concern. Figure 5.5 shows a `PipeActor` class with two active object methods : `produce_dv` and `fetch_dv`, the former creates a distributed vector and returns it as a distributed future, and the latter fetches an input distributed vector (in parallel) and returns it as a distributed future. The `fetch_dv` function could do something with the distributed vector once it has it, but the purpose of this example is to show the parallel transfer and the usage of distributed futures, as simply and clearly as possible. The `produce_dv` method is not shown because `fetch_dv` is enough to show the usage of distributed futures and their associated primitives. Figure 5.6 shows a main function instantiating three PipeActor active objects (`a`, `b`, and `c`) with two processes each. A distributed vector is created on the first object `a`, which is successively forwarded to `b`, and then to `c` using the `fetch_dv` method.

Figure 5.5 shows the core of this example, the `PipeActor` class and its `fetch_dv` method. Let us first take a look at the object variables declared at lines 4-5; a vector `_v` is declared for storage and a `vector_distribution<int>` is declared for sharing a `vector_distribution<int>` parameter between different functions; as explained in Chapter 3, we use the object scope so that different functions have access to the same variables. The `fetch_dv` function itself is rather simple, it first synchronizes the `_vd` distributed future it received as parameter with a `get` operation on the `Future` object, and assigns the resulting `vector_distribution<int>` object to the `_vd` object variable.

```
class PipeActor : public ActorBase
{
private:
    std::vector<int> _v;
    vector_distribution<int> _vd;

    void bsp_fetch_dv() {
        int s,p;
        size_t part_size, part_offset, size;

        s = bsp_pid();
        p = bsp_nprocs();

        broadcast_vd(_vd);

        // Get the size of the whole data
        size = _vd.getVecSize();

        // Compute the data part indexes this process is inderested in
        part_size = size / p;
        part_offset = s * part_size;

        // Resize vector _v so that it has enough space to receive the data part
        _v.resize(part_size);

        // Receive the right data part into _v
        get_part(_vd, part_offset, _v.data(), part_size);

        // ... Here we could work on the data part stored in _v

        // Store the data part
        register_result(_v.data(), part_size, part_offset);
    }

public:
    vector_distribution<int> produce_dv(size_t size) { ... }

    vector_distribution<int> fetch_dv(DistrFuture<int> vd) {
        _vd = vd.get();
        bsp_run(&PipeActor::bsp_fetch_dv);

        // Assemble and return the registered metadata(instead of the full data)
        return gather_vd_parts<int>();
    }
};
```

Figure 5.5 – PipeActor forwarding distributed data

```
int main()
{
    // ...
    Proxy<PipeActor> a = createActiveObject<PipeActor>({1,2});
    Proxy<PipeActor> b = createActiveObject<PipeActor>({3,4});
    Proxy<PipeActor> c = createActiveObject<PipeActor>({5,6});

    DistrFuture<int> f1,f2,f3;

    f1 = a.produce_dv(1000000);
    f2 = b.fetch_dv(f1);
    f3 = c.fetch_dv(f2);

    //...

    return 0;
}
```

Figure 5.6 – main function showing three PipeActor objects forwarding a distributed vector through distributed futures

The `bsp_run` function is then called with function `bsp_fetch_dv` as parameter. This makes all the processes of the actor enter the `bsp_fetch_dv` function. As we will see, this function produces a distributed vector and each process registers the description of each part it owns; these descriptions are then gathered into another `vector_distribution` object through the `gather_vd_parts` primitive, which is returned right away. Let us now take a look at the `bsp_fetch_dv` function; after declaring and initializing some variables, the first thing it does is calling the `broadcast_vd` function, which makes the head process share the `vector_distribution<int>` object it received as parameter through the `DistrFuture` object that was synchronized, as we explained above. Every process is interested in receiving a distinct range of the corresponding distributed vector: this is done simply by dividing the size of the distributed vector by the number of processes `p` to obtain the part size (we assume the size of the distributed vector can be divided by `p`). The offset is the obtained by multiplying the size by the pid (that ranges from `0` to `bsp_nprocs` -1), resulting in a block-distribution. Then, every process calls `get_part` to receive its part of the distributed vector inside the `_v` vector object, after it is resized so that it has enough room to hold it. From this point, every process has a part of the distributed vector and a computation could take place. Here we only put it aside using the `register_result` primitive, using the same range and size. The

parallel function is then exited and the distributed future is returned as explained above.

5.5 Conclusion

In this chapter, we presented the concept of distributed futures, which is an unification of futures and distributed arrays: a distributed future is a future that 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 are interested in from the processes that computed them. A distributed future is a programming abstraction that makes programming easier and in particular synchronization and data transfer; it also makes the communication between data-parallel entities more efficient than with standard futures (as we will validate in Chapter 7).

We implemented this notion in the context of the BSP active objects presented in Chapter 3, that allows several BSP entities to interact in a task parallel and asynchronous manner. We will come back on implementation aspects in Chapter 6 and evaluate our implementation in Chapter 7.

Improvements can be envisioned, such as a declaration of distribution policy that specifies how to redistribute data according to pre-defined or user-defined distribution type. In our implementation, the user manually specifies the ranges of distributed data needed, which may lead to programming mistakes being made on these parameters. A declared distribution policy would be easier to manipulate since primitives could be created in order to redistribute data according to a distribution policy, without the user having to compute the offsets in an error-prone manner. This is however not critical as BSPlib (which we use for our implementation of intra-actor communications) programmers are already expected to know how to manipulate data in this manner anyway.

This concept of distribution policy would allow further improvements such as allowing the user to specify a pre-fetching of data according to a particular distribution type, before the computation requiring this data is started. This would allow an active object to trigger this data transfer between BSP processes earlier. Instead of pulling data when the BSP computation starts, by invocation of the `get_part` primitive, the idea would be 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. Such a pre-fetching strategy is outside the scope of this work.

Chapter 6

Implementation

6.1 Introduction

In the previous chapters, we presented BSP active objects (Chapter 3) along with a distributed future extension (Chapter 5). In order to show how these concepts can be implemented, we developed our own implementation. We have already introduced some key implementation aspects in the previous chapters; here, we go into further detail concerning our implementation in order to better describe it. While this chapter is not a full description of our implementation and does not cover every detail, we show the aspects that we deem interesting and worth mentioning.

In this chapter, we start in Section 6.2 by describing the environment on which we chose to base our development work. To present our BSP active object implementation, we describe it as if we first implemented normal active objects in Section 6.3, and then turned them into parallel BSP active objects in Section 6.4. We then explain further implementation aspects for distributed futures in Section 6.5 before concluding in Section 6.6.

6.2 Environment

In this section, we describe the technical environment that we used for implementing BSP active object.

To implement our hybrid BSP active object model, we chose to rely on the BSPlib implementation for the implementation of the BSP part our model. We made this choice because, as we have seen in Chapter 2, BSPlib is a direct implementation of the BSP model which makes it possible to program directly BSP communications. This also gives us the possibility to reuse an existing BSPlib implementation. Since the BSP part of an active object is meant to speed up a task, reusing an existing implementation allows us to benefit from previously optimized implementations. Moreover we can reuse existing BSPlib user code within our implementation.

We chose to use a BSPlib implementation that runs on the top of MPI, because it is relatively easily to modify this library and add aspects specific to our imlpementation of active objects. Another reason is that Wijnand Suijlen, a supervisor of the author of this thesis, developed several BSPlib implementations over MPI and could quickly confirm it would not be too difficult to modify the libraries in order to fit our model.

Along our development work, we did use several BSPlib implementations running on the top of MPI, but we did not decide this from the start. We started with BSPonMPI v0.2, because it appeared easy to integrate it. The author of this library later developed a professional BSP implementation for Huawei that supports BSPlib; this implementation was later used within our implementation as it was more stable and efficient than BSPonMPI v0.2; but since the code of this Huawei implementation is not open source, we had to keep relying on BSPonMPI v0.2 for some time. Later, the same author redeveloped BSPonMPI from scratch into a more stable version that remains open-source, (starting from v1.0), which we integrated into our implementation to replace BSPonMPI v0.2.

Since we wanted the active object and BSP implementations to co-exist within the same environment to avoid having to transfer data between different environments, and that there is no well-established active object library in C++, this meant we had to develop our own active ob-

ject layer for implementing our BSP active object model. While it was possible to do differently (through sockets for example), we chose to implement active objects in MPI that was already used by BSPLib. It avoids us to introduce new concepts in an already complex environment.

When we developed the worker and management thread architecture, first described in Section 3.4, we chose to rely on the pthread library for thread programming, mostly because it is a widely used and available library, with a reachable community, and because the author is familiar with this thread programming library.

Active objects are meant to be easily programmable, but with both active and passive objects for efficiency reasons. Active objects manage a thread and can receive asynchronous method invocations, while each passive objects can only be manipulated by a single active object. When a call is made to an active object, it is expected that passive objects may be passed as parameter. Therefore (to maintain the invariant that each passive object is manipulated by a single active object), it is required to serialize these objects in order to send them to a remote process. We chose to rely on the Boost serialization library in order to do so, there are two reasons for this. Firstly, because we wanted to avoid having to develop the serialization of objects ourselves. Secondly, because this library allows the serialization of objects in a rather user-friendly manner. Consequently, any object that can be serialized according to the guidelines of this library can be sent as parameters of active objects.

6.3 Active object implementation

In this section, we detail the main aspects of our implementation of active objects with C++ and MPI. In Section 6.4, we will describe the BSP part of this implementation.

We start this section by describing the challenges associated to implementing active objects in C++ in Section 6.3.1; we then show how we implemented the communications with MPI in Section 6.3.2.

6.3.1 C++ active objects

As explained in Section 6.2, the active object part of our implementation had to be developed from scratch in C++. The C++ language does not provide the greatest tools for such an implementation. For example, creating an active object from an existing class and being able to call its methods as active object methods is not an easy task. This is mainly because reflection, i.e. the ability of a program to examine its own code, is not provided by C++.

Let us first consider the example of creating an active object on an existing MPI process to show the benefits of reflection. In this example, we want to create an active object of class `Foo` on process `P1`. With reflection, this could have been implemented by sending a message to process `P1` containing the name of this class (messages are just strings over the network). The process `P1` would read this message and create this class from the string, but this is not possible in C++. This is why we had to create a mapping from string to class ourselves. The same problem arises, in a more complicated manner, for active object calls. A call to an active object method involves sending a message to a process's active object that represents the call to this method along with its parameters. Interpreting and translating this message to a call to an object's member function, with their typed parameter is also complex in C++ and a mapping has to be done in order to implement reflection-like features.

Fortunately, the C preprocessor and C++ templates allowed us to generate most of the code for the user so that the generation, from a class, of an active object can be as simple as possible for the active object programmer. Only a limited set of macros is exposed to the user in order to generate the code translating an object class to an active object class. The main macro exposed to the user is the `DECL_ACTOR` macro; it enables the user to turn a class into an active object class. From this class, an active object can be created on other processes. This macro and its syntax were developed thanks to the features of the Boost Meta Programming Library (MPL) [33].

The syntax of `DECL_ACTOR` is shown in Figure 6.1. A class name is given as first parameter, then individual sets of active object functions are given between parenthesis; the user has to give, for each active object function, its return type, its name, then the type of all of its parameters. With

```
DECL_ACTOR(ClassName,
           (return1_type, function1, arg1_type, arg1_type, ...),
           (return2_type, function1, arg2_type, arg2_type, ...),
           (...)
)
```

Figure 6.1 – Syntax of `DECL_ACTOR`

```
DECL_ACTOR(PipeActor,
           (vector_distribution<int>, produce_dv, size_t),
           (vector_distribution<int>, fetch_dv,  DistrFuture<int>)
)
```

Figure 6.2 – Example usage of `DECL_ACTOR`

this information, we can generate the necessary code to turn the class given as parameter into an active object class. The `DECL_ACTOR` macro generates:

1. **A `Proxy` class**, which enables the user to call active object functions,
2. **A `handler` function**, which a remote process enters when it is instructed to create this actor,
3. **A `createActor` method**, which creates the `ClassName` actor on a remote process,
4. **A `Maker` class**, which at construction time, assigns a mapping from the class name as string to the `handler` function, so that a remote process may find this function when it receives a message instructing to create this actor.

An example usage of `DECL_ACTOR` is shown in Figure 6.2. This code turns the `PipeActor` class, shown in Figure 5.5, into an active object class.

An additional macro has to be called by the user: the `REGISTER_ACTOR(ClassName)` macro. This macro initializes a `Maker` object (defined when calling `DECL_ACTOR`), corresponding to the class given as parameter, that creates the mapping from the class name as string to the correct `handler` function (also defined in `DECL_ACTOR`). The `DECL_ACTOR` can be called within a header file, but the operation performed when calling `REGISTER_ACTOR` may only be done in a compiled C++ code. This is why we created this `REGISTER_ACTOR` macro, so that it can be called separately from `DECL_ACTOR`. For the `PipeActor` example above, simply calling `REGISTER_ACTOR(PipeActor)` from

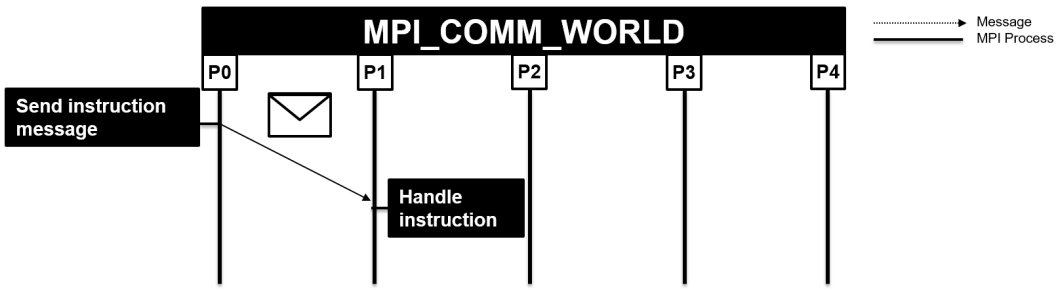


Figure 6.3 – Management of MPI processes

a compiled file (e.g. PipeActor.cpp) is enough: it registers the actor and makes it possible to create and invoke active objects oy type `PipeActor`.

6.3.2 MPI implementation of actors

As explained in Section 6.2, we use an implementation of MPI to implement communications between actors. In this section, we describe in more detail how these communications are performed.

An MPI program starts with all processes entering the `main` function of an SPMD program, with all of its processes belonging to an `MPI_COMM_WORLD` group, with each process numbered from `0` to the total number of processes in this group minus one. This statically defined parallel structure is already contrary to the active object style of programming, this is why we impose that the first function the user must call is an initializing function called `activebsp_init`. Aside from initialization purposes, this function only lets the process `P0` exit this function to proceed executing the rest of the `main` function. All the other processes are then stuck in this function, waiting to receive further instructions; these instructions, for example, may create an active object, destroy it, call an existing active object or retrieve the value of a future. The MPI process of pid `0` is then a special case, because it executes the main function and it is not associated to any particular object. We call this process the main process or the coordinator process.

This base architecture is illustrated by Figure 6.3. This environment has five MPI processes, including the coordinator process (`P0`) and four processes (`P1`, `P2`, `P3` and `P4`). In this figure, an instruction message is sent from `P0` to `P1`. This message may for example instruct the process to

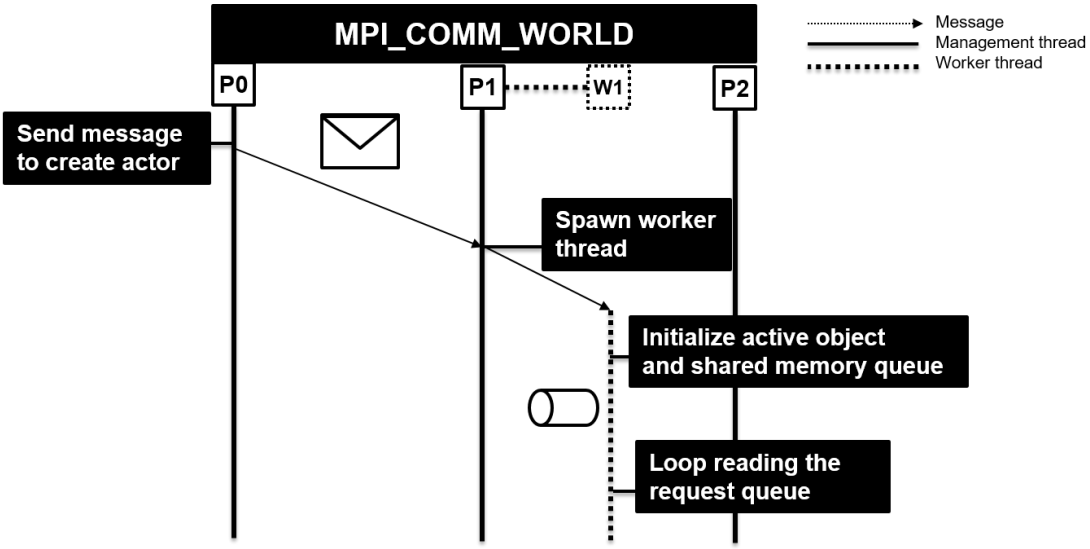


Figure 6.4 – Creating an actor

instantiate an active object, or to call an active object if one is already attached to `P1`. Note that while active objects may be created and destroyed dynamically on these processes (all except `P0`), there can be processes that remain idle throughout the life of an application, if the programmer does not use them to create an active object.

From this description, the reader may remember the discussion about management thread presented in Section 3.4; each of the MPI processes (except `P0`), is in fact one such management thread, even though they are not attached to any active object yet.

The examples of creating an active object, and calling one are important to understand our architecture, we will now focus on each of them.

Figure 6.4 shows process `P0` initiating the creation of an active object on process `P1`. As explained above, a message describing the creation of the actor is sent to `P1`. A new thread `W1` is then spawned by `P1`; this thread is the worker thread of this actor as described in Section 6.4. This thread initiates the active object by instantiating the corresponding object and creating the request queue in a memory that is shared by the two threads. The worker `W1` then reads the request queue, which is for now empty, meaning this worker is now blocked. Note that when a thread is created,

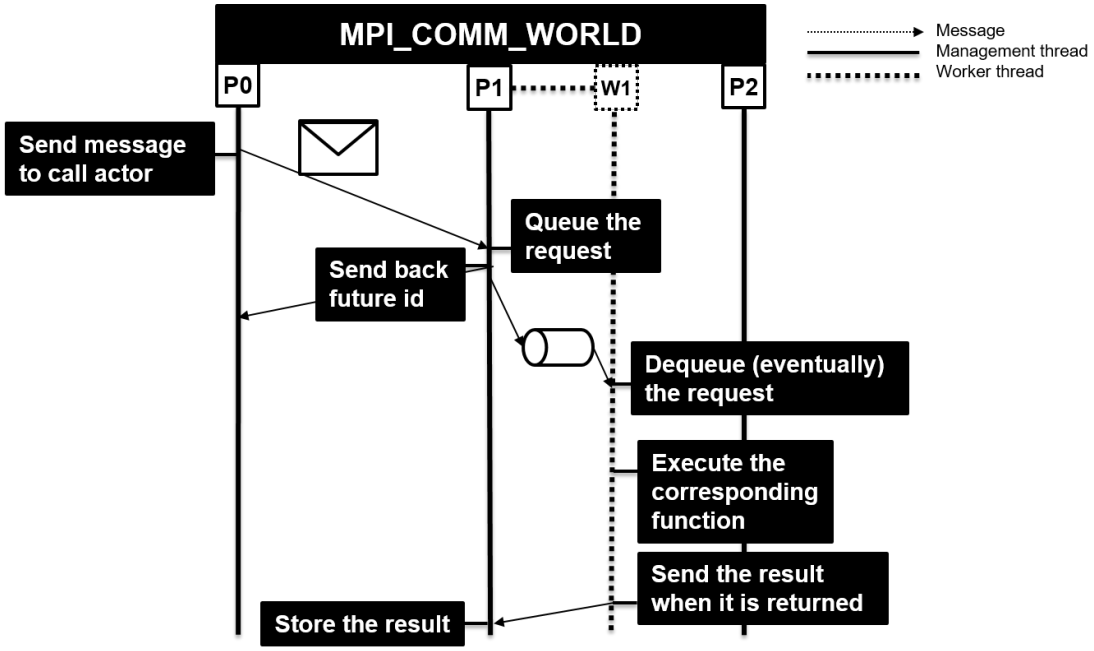


Figure 6.5 – Calling an actor

it is associated to the same MPI pid as the process that created this thread; this means that `W1` is associated to the same pid as `P1`.

In this first example, we showed how we implement the creation of actor in an MPI environment, where all processes already exist in an application. In an environment that is able to spawn processes easily, the actor creation implementation would have been slightly different. Indeed, the process allocated to the new actor would have to be spawned before sending the creation message, instead of this process already existing and being blocked in the `activebsp_init` function waiting for a message.

Figure 6.5 then shows process `P0` calling an actor hosted on `P1`, which may be the same as the one created in Figure 6.4. Process `P0` sends a message to `P1` describing this call, along with its parameters. This message is then queued by `P1` (the management thread) into the shared memory queue belonging to the actor. After this request is acknowledged by being queued, this management thread sends a response to the `P0` calling process, with an identifier that is used to build a `Future` object; we will come back on this later in this section. The message in the queue is eventually

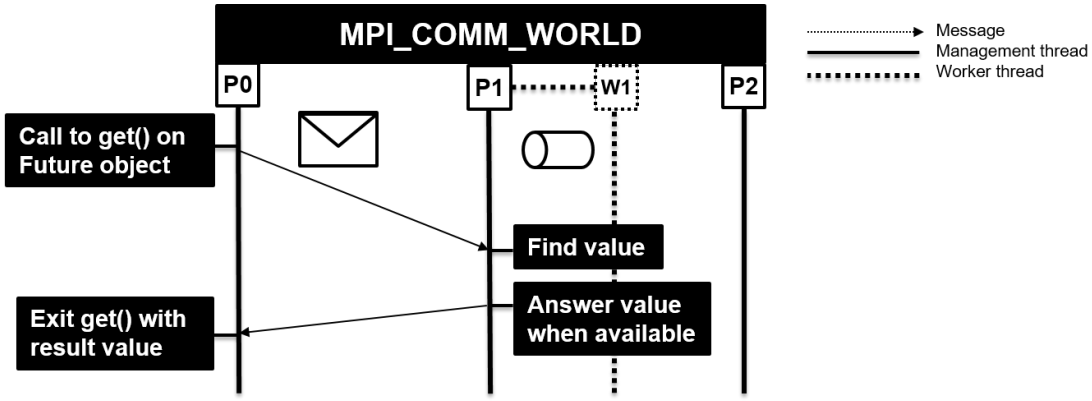


Figure 6.6 – Calling `get` on a `Future` object

dequeued by `W1` when it is not handling another message and this request is next in line in the queue. After the message is dequeued, its content is examined so that the requested function is known, and the arguments of the call are de-serialized. The function is then executed by the worker thread. When this execution finished, and the result value is returned, this value is then serialized and sent to the management thread, which just stores it.

Our implementation of futures is quite simple. The `Future` object simply contains the MPI pid of the head process of the active object that is producing the result, along with a unique key identifying the result for the called active object. This key was allocated and returned by this process when the active object call was made. Figure 6.6 shows what happens when this result is queried (by calling `get` on the corresponding `Future object`). In this situation, the management thread simply answers, when the result is available, with this serialised result. The `get` call is just left pending as long as there is no value to answer. After the result value is sent back to `P0`, the `get` call de-serializes this result so that it is available in the actor that perdormed the `get`.

6.4 BSP active object implementation

In the previous sections, we have described how we implemented basic active objects within our C++/MPI environment. We will now explain how we turned these classical active objects into BSP

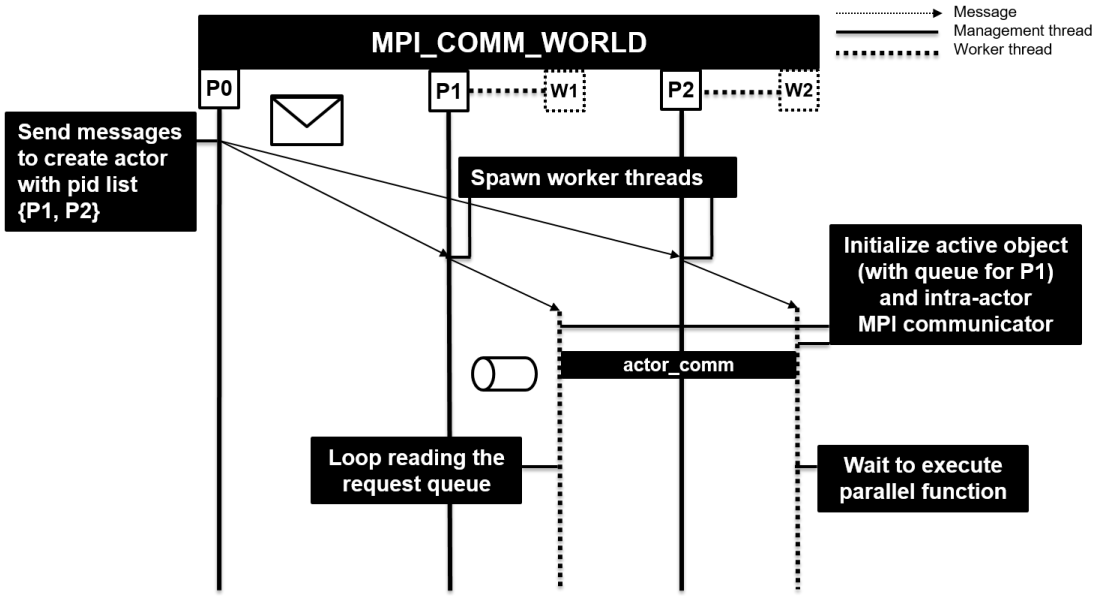


Figure 6.7 – Creating a multi-process actor

active objects: active objects with multiple processes which are able to execute data-parallel code in a BSP-style.

6.4.1 Giving multiple processes to active objects

In Section 6.3.2, we have shown the creation of a single-process actor. Here we will show how we implemented actors with multiple processes. We start from the example of Figure 6.4, where we showed how we created a single-process actor using MPI. Figure 6.7 now shows a multi-process actor being created. In this figure, process `P0` initiates the creation of a single actor, on processes `P1` and `P2`, by sending a message to each of these processes with the list of pids that will belong to the active object (for each process). The first process in the list will serve as the head process of the active object; here `P1` is the head process. All the active object processes start by creating a worker thread (here `W1` and `W2`) and then perform some initializations, but now only the head process creates a request queue. A key difference in the initialization step of this parallel actor is that all the worker threads create an `actor_comm` MPI communicator (a new group of processes).

All the specified active object processes now belong to this MPI communicator. This communicator allows calling MPI collective operations where only the active object processes (`P1` and `P2` in the example) participate instead of all the processes of the `MPI_COMM_WORLD` global communicator that contains all the processes in the MPI environment. This `actor_comm` communicator is particularly important in the integration of an existing BSPlib library, because our modified BSPlib implementation initializes the library from this communicator, which allows us to call the BSPlib `bsp_sync` barrier operation involving only the processes in `actor_comm`. From this point, the management threads continues running without interfering with the rest of the actor creation, we now focus on the behavior of the rest of the active object The `W1` worker thread, now reads its request queue to start processing requests, like a single-process actor. What is different is that the `W2` worker thread expects a communication from the `W1` worker thread requesting a call to a parallel function. This communication is made through a call to the `bsp_run` primitive, which `W1` may initiate while handling a request. We will further describe the implementation of `bsp_run` in Section 6.4.3.

6.4.2 BSP implementation within parallel actors

In our implementation of BSP active objects, we used BSPlib to implement the data-parallel part of our active objects. More precisely, we used an existing BSPlib implementation over MPI that we slightly modified so that it can be initialized with an MPI communicator, and not `MPI_COMM_WORLD`. We explain below how we integrated BSPlib within our implementation.

BSPlib itself has primitives that dictate how a BSP program is structured. Namely, the `bsp_init(void (*spmd)())` function must be called before calling any other BSPlib primitive to instruct all MPI processes to enter the designated `spmd` function; the process of pid `0` is an exception: it does not enter the `spmd` function through `bsp_init`. This process instead continues executing and must call `spmd` directly on its own. Before it makes this call, this process continues sequentially and may, for example, handle program arguments. This `bsp_init` function may be called only once, this is also the case of `bsp_begin(int p)` primitive (and the corresponding enclosing `bsp_end()` primitive). The `bsp_begin` function requests at most `p` processes for the whole duration of the BSP program. This program structure is incompatible with BSP active objects be-

cause we want to handle several requests and thus create several BSP parallel computation during an active object lifetime. Also we want to be able to create and destroy active objects with possibly different process subsets. Consequently, we must remove the restriction that these primitives may be called only once per program lifetime. To do so, the BSPlib implementations themselves had to be modified.

The changes to the BSPlib implementations were required so that the `bsp_init`, `bsp_begin`, and `bsp_end` functions may be called multiple times without terminating the program. Another main reason for modifying the BSPlib libraries was to initialize them from a process subset (not all available processes in the environment). For example, in MPI implementations of BSPlib, the `MPI_COMM_WORLD` global communicator is often used when calling communication primitives. This was an issue because we do not want to involve every process in the environment when a BSP active object calls BSPlib, but only the processes of this BSP active object. To solve this issue, we created a custom initialization function called `bsp_plug_mpi(MPI_Comm comm)`, which allows us to initialize a BSPlib library with a specified subset communicator, so that multiple active objects can live in the same MPI environement using different processes. We then modified the BSPlib libraries used by our framework so that they use this `comm` MPI communicator instead of `MPI_COMM_WORLD` when they communicate.

In our BSP active object implementation, we chose not to expose the `bsp_init`, `bsp_begin` and `bsp_end` primitives to the user. The `bsp_run` primitive already imposes a structure where the function given as parameter is entered in parallel. The purpose of `bsp_init` is also to trigger the parallel execution of the function given as parameter (of all processes except the one with pid `0`). If `bsp_init` would be exposed to the user, it would co-exist with `bsp_run` while serving a similar purpose. Instead, the modified BSPlib library is initialized by the BSP active object library when creating the BSP active object itself. From the point of view of the BSPlib library, all processes called `bsp_init` (also `bsp_begin`) and entered the `spmd` function given as parameter from the moment the BSP active object is created. The `bsp_end` function is called when the BSP active object is destroyed.

From the BSP community's point of view, this model can be seen as a high-level encapsulation of

subset synchronization. While we have chosen to implement the model with subset synchronization (of MPI communicators), the subsets are all disjoint in nature because a process can only participate in one active object (process subset). Because of this aspect of our implementation, and as seen in Section 2.3, the downsides of subset synchronization apply to our implementation, including sacrificing the accuracy and simplicity of the BSP cost model. However, another implementation making the choice of interleaved threads of BSP computations as seen in [65] could be envisioned in a different implementation of this model. Our model does not fit with a global BSP cost model by principle. Indeed, BSP active objects rely on several functionally different active objects with different synchronization patterns. Consequently, such a system is not be as perfectly balanced as divide and conquer algorithms. With an implementation similar to the one proposed in [65], BSP active objects are bound to suffer from imbalanced communications and computations before global barrier synchronizations if they were to be synchronized altogether. This is why our implementation does not feature global barrier synchronizations, but only subset synchronizations (of active object processes, not synchronizing all the processes in the environment at the same time).

6.4.3 Implementation of `bsp_run`

We have seen that `bsp_run` takes a parallel function as argument, and that every process of the active object then enters this function collectively As we will see in this section, our implementation of the `bsp_run` function is quite simple.

Because the nature of `bsp_run` is to ask remote processes to execute a given function, the same problem as described in Section 6.3 about executing a remote function by name occurs. To solve it, we have to specify the mapping beforehand between function name and function pointer; we do so through a `register_spmd` function that must be called within the object constructor because it is called in parallel at the object creation time. This `register_spmd` function associates an identifier to the function given as parameter.

As shown in Figure 6.8, the implementation of `bsp_run` is then a simple broadcast of this identifier from the head process to the other processes of the same active object. Every other process has to be ready for this broadcast, and once they receive this identifier, they enter the

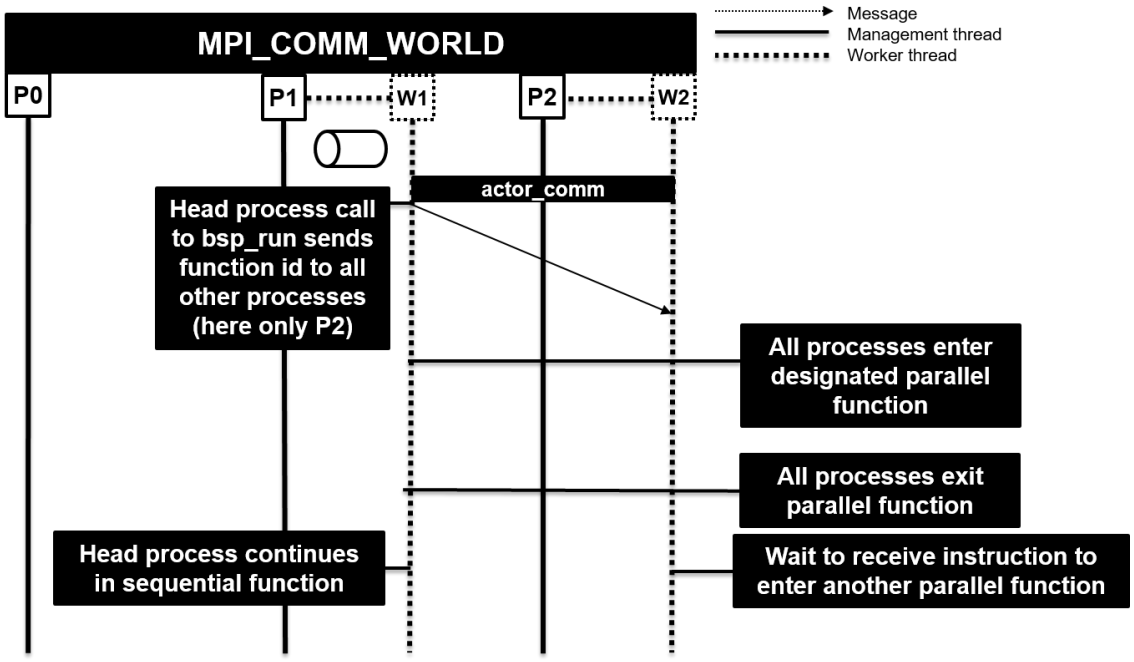


Figure 6.8 – Implementation of `bsp_run`

corresponding function, after looking it up in a table that mapped it to the right function pointer thanks to the previous call to the `register_spmd` function.

In this section, we have shown the main aspects of our BSP active object implementation over MPI. We have seen that all MPI processes (except the one with pid `0`) are management threads that are able to instantiate BSP active objects by spawning a worker thread. For parallel BSP objects, these worker threads are linked together by a dedicated MPI communicator. This communicator is then used to initialize a BSPlib library that only includes the processes in this communicator. A BSP active objects is then able to call our `bsp_run` primitive, which executes a parallel function on all the processes of an active object. We have seen that to do so, this `bsp_run` primitive is simply broadcasting an integer to these processes. In short, this BSP active object is able to execute code in parallel thanks to `bsp_run` and to call BSPlib primitive thanks to the BSPlib library dedicated to these processes.

6.5 Distributed future implementation

As we have seen in Chapter 5, distributed futures allow the parallel transfer of distributed data between two parallel entities. The processes of a parallel entity keep their parts of some distributed vector, which can be queried by another group of processes through a distributed future and a `vector_distributed` structure. When we presented the requirement of having a management thread in Chapter 3, we pointed out that only one management thread was required for each active object. Because our distributed future concept makes processes keep part of distributed data, they must also be available in order to send these parts when it is asked of them. As we have seen in Chapter 5, this means that we then need to have one management thread per active object process instead of just one for the whole active object.

The `get_part` function, introduced in Chapter 5, is a central part of our implementation of distributed futures. This primitive allows the retrieval of any subpart of a distributed vector from a `vector_distribution` structure given as parameter.

Algorithm 1 get_part pseudocode

```
Require: vector_distribution vd, size_t offset, char * out_buf, size_t size
 1: req_end ← offset + size
 2: for each {vd_pid, vd_local_id, vd_size, vd_offset} ∈ vd do
 3:     part_end ← vd_offset + vd_size
 4:     if not (part_end <= offset or vd_offset >= req_end) then
 5:         if offset >= vd_offset then
 6:             req_offset ← offset − vd_offset
 7:         else
 8:             req_offset ← 0
 9:         end if
10:         req_size ← min(vd_size, req_end − vd_offset) − req_offset
11:         dst_buf = out_buf + req_offset + vd_offset − offset
12:         requestPart(vd_pid, vd_local_id, req_offset, req_size, dst_buf)
13:     end if
14: end for
```

Algorithm 1 shows a simplified pseudo-code for the implementation of `get_part`. This algorithm relies on our `vector_distribution` implementation as a list of quadruplets, as described in Chapter 5. The *requestPart*(*pid*, *local_id*, *offset*, *size*, *buf*) function, called in line 12, requests *size*

bytes from process *pid*, from position *offset*, of the part it identifies by *local_id*; the result is then written in the *out_buf* buffer.

This algorithm iterates all the quadruplets inside the `vd vector_distribution` structure given as parameter. For each of these quadruplets, the algorithm checks (at line 4) if the part this quadruplet describes overlaps with the part requested by the *offset* and *size* parameters of `get_part`; if it does, it means there is something in this vector part that is required (all of it or just a subpart). The following lines from 5 to 11 consist of calculating exactly what is required, and at which spot this part should be written inside the *out_buf* buffer. From this information, the *requestPart* function described above is called to initiate the communication of exactly the data that is required.

6.6 Conclusion

In this chapter, we have described key aspects of our implementation of BSP active objects and of distributed futures. We have also discussed our reasoning for the different choices we made. We tried to keep a middle ground between performance and programmability. For example, reusing an existing BSPlib implementation was a choice for performance; while enabling flexibility and responsiveness through the introduction of a management thread was a choice in favor of programmability for the user. Because of this, our implementation could be improved in either direction by sacrificing the other aspect. Nevertheless, our implementation serves as a usable and effective proof of concept implementation for our BSP active object model, as well as distributed futures.

Chapter 7

Experimental evaluation

7.1 Introduction

In Chapter 3, we have introduced our hybrid BSP active object model, along with a distributed future extension in Chapter 5. We implemented these concepts in C++ over MPI as described in Chapter 6. In this chapter, we are now interested in evaluating the performance of this implementation through benchmarks.

We start in Section 7.2 by describing our execution environment, we then provide some small communication benchmarks in Section 7.3 followed by a more elaborate benchmark scenario in Section 7.4. We then give a conclusion for our experiments in Section 7.5.

7.2 Experimental Setting

For these experiments, we have at our disposal ten Huawei RH2288v2 servers, each with two Intel Xeon E5-2690v2 CPUs that have ten cores each. This kind of machines is what our lab has in biggest number, and we prefer to have a group of identical machines in order to better compare results; this is why we do not use more nodes for our experiments.

We use the Intel C++ compiler version 18.0.1. For our communication benchmarks in Sec-

tion 7.3, we use one process per node in order to better observe performances in a simple environment. However, extra parallelism can be achieved in the scenario of Section 7.4; this is why we will use more processes per node. In this scenario, and 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 nodes.

7.3 Communication benchmarks

In this section, we show the result of two small benchmarks involving our BSP active object implementation. We show what we call the vector call benchmark in Section 7.3.1, which focuses on active object calls and `get` operation. We then present our relay vector benchmark in Section 7.3.2, which focuses on evaluating data transfers between parallel actors with different kinds of futures.

7.3.1 Vector call

In this section, we are interested in measuring the performance of an active object method call, and of the future resolution. We design a scenario where an active object method is called with a vector as parameter. This method simply returns the input vector, and the process that called it calls `get` on the corresponding `Future` object.

Figure 7.1 illustrates this scenario. This is basically combining the scenarios of Chapter 6 involving an active object method call (Figure 6.5) and a future resolution (Figure 6.6). The vector is first serialized, then sent to the management thread of the `object` active object. After the message is dequeued, the vector is de-serialized by the worker thread, which returns it (and re-serializes it) to the management thread. This management thread then sends the result to the coordinator process, because it requested the resolution of the associated `Future` object. The result is then de-serialized in order to retrieve the associated `vector` object. For this scenario, we measure the time it takes to make the active object call, and then the times it takes after the call to retrieve the resulting data. We measure two successive `get` operations, because the first one is executed right after the call.

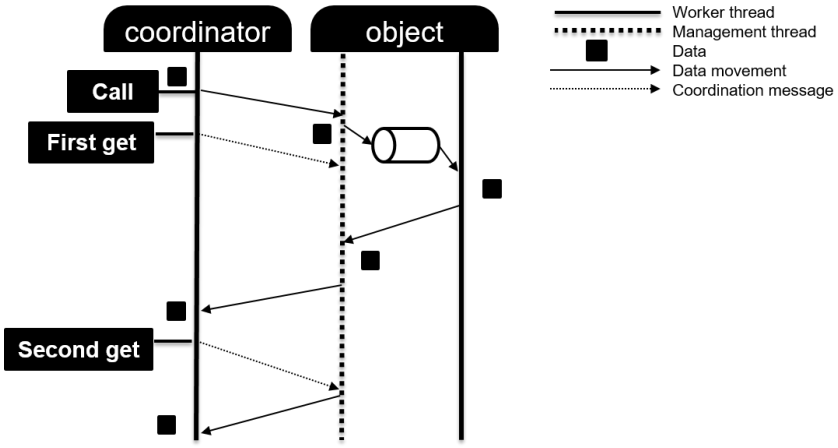


Figure 7.1 – Vector call benchmark scenario

Because of this, the first `get` also measures the time it takes for the request to be dequeued and processed. This is also interesting to measure, but it does not exclusively measure a `get` operation. However, a second get gives us a better indication of the time it takes for only executing the `get` operation, because the result associated to the future is then known to be produced.

Our implementation of `Future` keeps the serialized data when it is first synchronized, so any `get` operation made after a first one does not require communication; this is why we make sure that we clear this data before executing the second `get` operation, in order to really perform and measure the communication for this `get` operation.

We vary the size of the input vector and observe the resulting execution time for the call and `get` operations. We make sure the active object queue is empty before each execution. We execute each scenario ten times and we take the average running time to report each result. All the measure we show have a Coefficient Of Variation (COV) of less than 10% (the COV is calculated as the standard deviation divided by the mean).

Figure 7.2 shows these measures according to the vector size. We can see that the time for the second `get` is slightly faster, but about equivalent to the time for the call. We can also notice that it takes about three times more time to execute the first `get` (when the result is not ready yet) than to execute the first `get` (when the result is ready). This is because, the input vector must

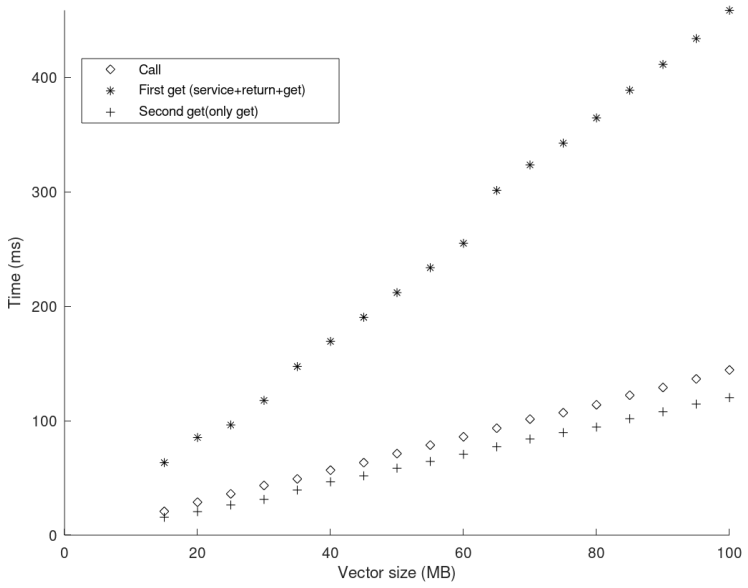


Figure 7.2 – Vector call benchmark results

be dequeued by the worker thread after the call, then sent to the management thread and then sent back to the coordinator process. This process involves more serializations and data movement, which is why we observe more time for executing the second `get` than executing the first one.

7.3.2 Relay vector

In this section, we are interested in comparing the performance of different kinds of future communication. To do so, we designed a scenario where a coordinator process sends a vector to a parallel object `objectA`. This object then distributes this vector among its own processes, for a possible computation in parallel (which we do not perform because we are only interested in communication time), then the same vector is returned (simulating a result data vector of the same size) to the coordinator. The coordinator process then sends the result (we will see how below) to a second parallel object `objectB`, which has the same behavior as `objectA`. The coordinator process then synchronizes and receives this result data.

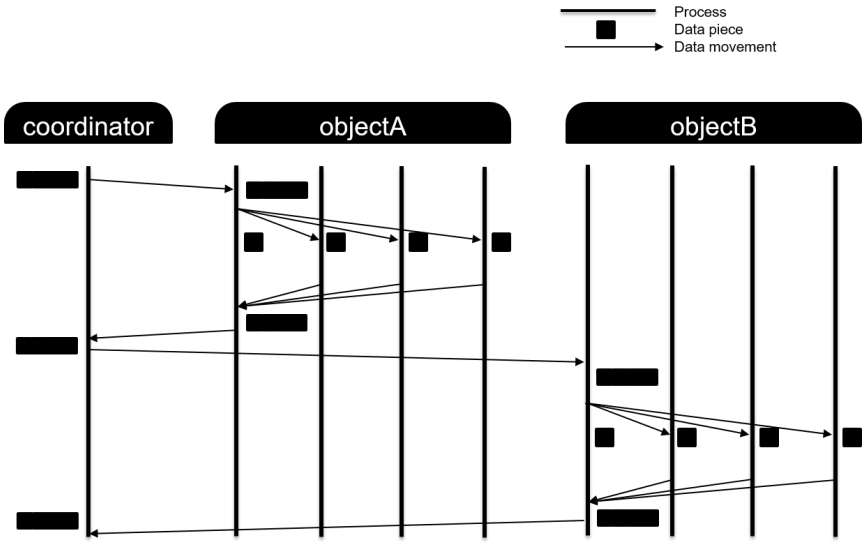


Figure 7.3 – Communications in Relay vector scenario for future

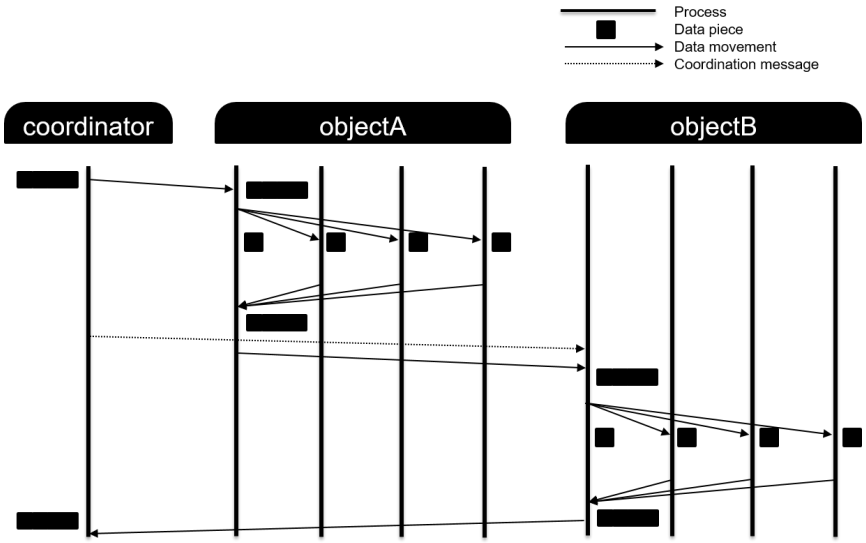


Figure 7.4 – Communications in Relay vector scenario for first class future

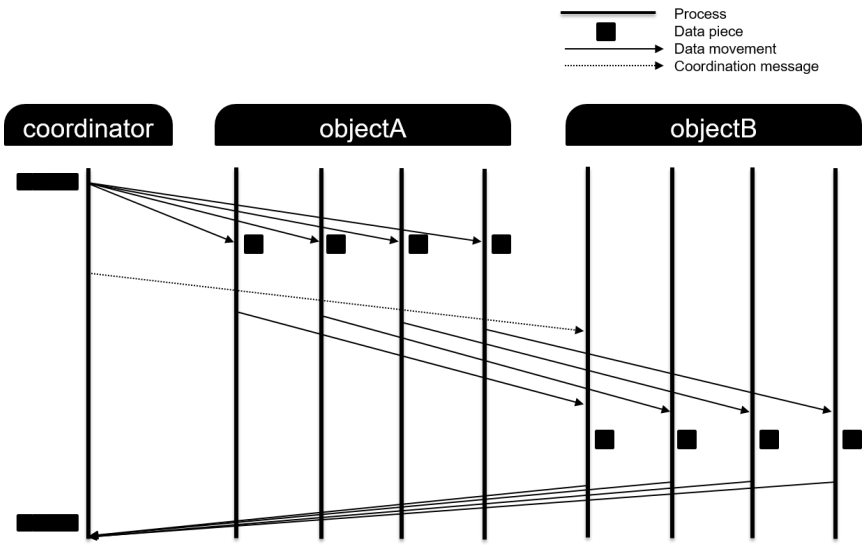


Figure 7.5 – Communications in Relay vector scenario for distributed future

We are interested in comparing the performance of different futures, so this scenario, described in an abstract manner so far, is not exactly the same for all futures. We compare:

1. **Normal futures:** The data is sent from `objectA` to `objectB` by having the coordinator process synchronize the associated future, and send the full data to `objectB` as shown in Figure 7.3.

2. **First class futures:** The data is sent from `objectA` to `objectB` by having the coordinator process send the first future directly to `objectB`, which synchronizes it without requiring the coordinator process to receive it at all. This is shown in Figure 7.4.

3. **Distributed futures:** the data is distributed directly from the coordinator process to the parallel processes of `objectA`. To do so, we use the `block_distribute` primitive we introduced in Chapter 5. The coordinator process then sends the distributed future to `objectB` by calling it. `objectB` uses this distributed future to receive the data in parallel from `objectA`. The distributed future that was created from calling `objectB` is then used by the coordinator process to receive the data through `get_part`. This is shown in Figure 7.5.

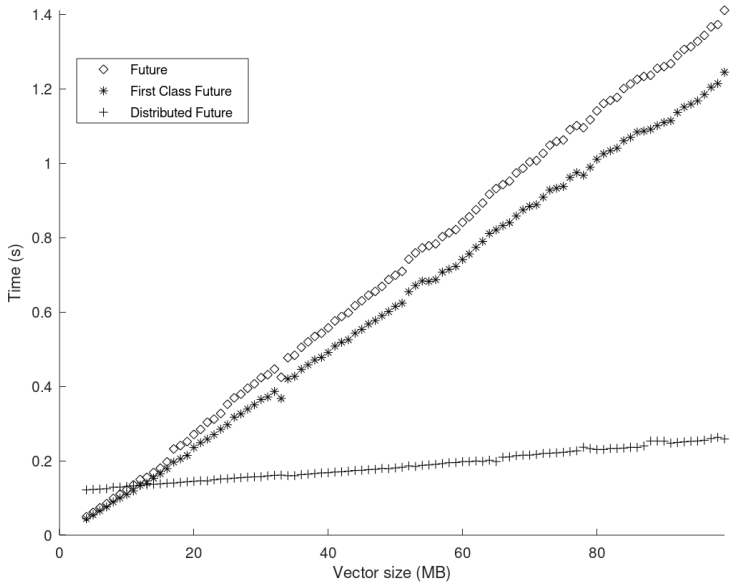


Figure 7.6 – Relay vector benchmark results

We execute this scenario using nine nodes in total, with one process per node. One node is dedicated to the coordinator process, and four nodes are allocated to `objectA` and four to `objectB`. This setting means that every process is on a remote node and all communications between processes go through the network. In this setting, the redistribution through distributed futures (and `get_part`) is simple because there is the same number of process on `objectA` and `objectB`.

As with the vector call benchmark, we vary the size of the input vector and observe the resulting execution time for each kind of future communication. We execute each scenario ten times and we take the average running time to report each result. The COV for these measures is always lower than 3%.

Figure 7.6 shows the result of this execution. We can see that for smaller input sizes, both normal futures and first class futures have a better performance, but that distributed futures scale much better as the vector size increases. The first class future always has an advantage over the normal future as a vector communication is avoided. Because distributed futures rely on communication of metadata, they perform a larger number of small communications. Thus when latency is the

limiting factor, i.e. for very small sizes, it is predictable that they are less efficient. However, we expected the distributed futures to perform better for small vectors, but this is probably due to the fact our implementation is not optimized.

Overall, the better performance as the input vector size increases clearly shows the benefits of distributed futures to communicate large amount of data.

7.4 Image comparison benchmark

In this section, we demonstrate the performance gain of distributed futures when used between parallel actors. To do so, we design a scenario that consists of a pipeline of three active objects. When using pipelines for task parallelism, the slowest stage dictates the throughput of the whole pipeline. So we show how we can choose the number of processes given to each stage according to performance, memory, or disk capacity goals. In our experiments, we will show that we can fine-tune the performance of pipeline stages by varying the number of processes assigned to each stage, and we compare the performance of distributed futures against a version without distributed futures.

7.4.1 Scenario

We consider a scenario of a web backend system that indexes and stores large numbers of images from its users that are in competition for a best picture prize. However, some users are gaming the system by uploading pictures multiple times, so that they get a better chance of winning. For that reason the backend filters all duplicates from the stored images. To avoid duplicates, we check if an image is already in the collection when it is being added. This may be time and memory consuming depending on the number and size of the images in the collection. To speed up the process, the backend maintains a thumbnail database as index. A thumbnail is a downscaled version of an original image obtained through lossy compression. Instead of comparing original images when we add a new image in our collection, we then simply compare thumbnails (we place ourselves in a setting where comparing thumbnails is sufficient and using hash-code is not useful because we

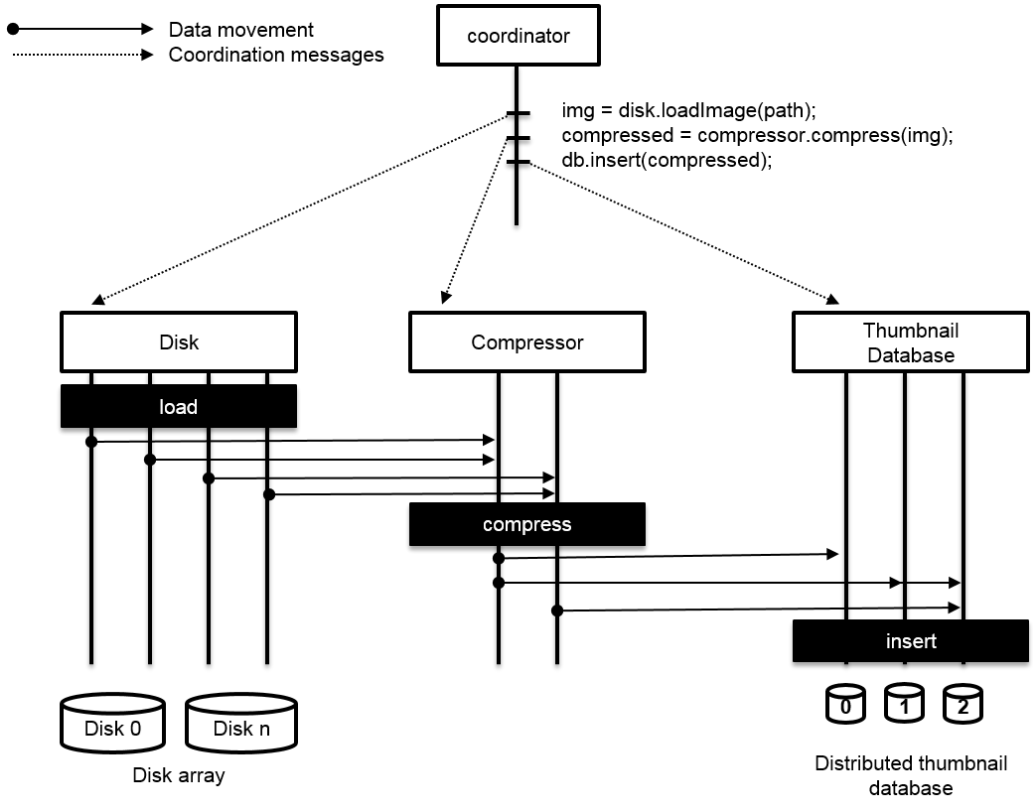


Figure 7.7 – One pipeline sequence

cannot afford to compare original images). Our setting also allow for an advanced comparison of thumbnails, e.g. not too sensitive to light and color balance. Comparing thumbnails, in contrast to comparing original images, is faster and requires less memory because less data is compared, but it requires compressing the original images into these thumbnails.

We place ourselves in a context where we have the thumbnail database in the main memory of a set of dedicated machines. The task of compression is assigned to another set of dedicated machines. Finally, we also dedicate a parallel file system component to reading source images on a set of disks. This forms a pipeline of three components, a parallel file system component that we call the disk component, a compressor component, and a database component.

For our experiments, we have implemented this pipeline scenario using the BSP active object implementation described in Chapter 6 that includes our distributed futures. Each component is

```
std::string path;
std::ifstream is(img_list_file);
while (getline(is, path)) {
    DistrFuture<char> img = disk.loadImage(path);
    // img.get();
    DistrFuture<char> compressed = compressor.compress(img);
    // compressed.get();
    Future<int> inserted = db.insert(compressed);
    // inserted.get();
}
```

Figure 7.8 – Main part of the coordinator process

implemented as a BSP active object, and it communicates data to other components in parallel through distributed futures. We have added a coordinator component to feed the pipeline, as shown in Figure 7.7. For each image, the coordinator instructs the disk object to read it; then the image is sent to the compressor object; after compression, the resulting thumbnail is sent for indexation to the thumbnail database; and then the image is inserted in the database if this is a new occurrence.

The insertion operation does this check; it inserts this thumbnail if it was not already in the database; otherwise, the insertion operation does not insert the image. While the thumbnail database component does not allow duplicates, the images that were sent into the pipeline for insertion were read from the parallel file system component. When an image's thumbnail is refused for insertion because it is a duplicate, the original image on the disk should also be cleaned. For simplicity, de do not show this duplicate removal process on the disk. A slightly simplified version of the code of the coordinator object is shown in Figure 7.8.

7.4.2 Results

We start by evaluating the individual performance of each active object in the pipeline to identify what resource allocation avoids bottlenecks. For measuring the time each object takes, we take the loop of Figure 7.8, but we synchronize the futures after each call, as shown in comments. This makes the execution block on the completion of each function and allows us to measure the time of each active object function individually. Without blocking, we would only measure the overhead

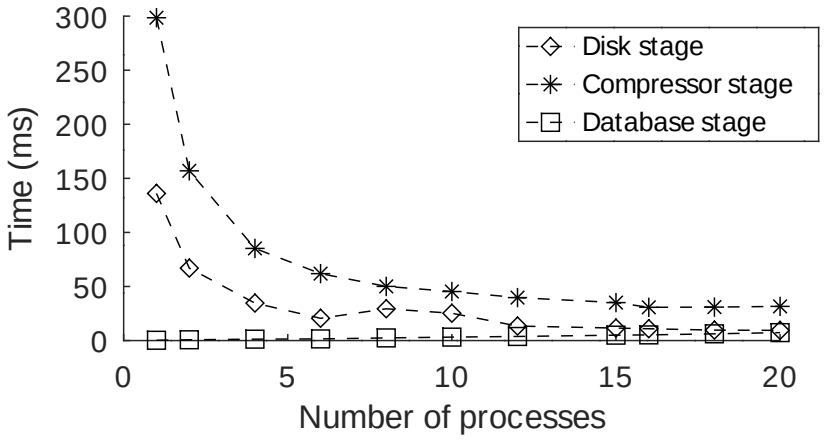


Figure 7.9 – Time for each stage of the pipeline with distributed futures

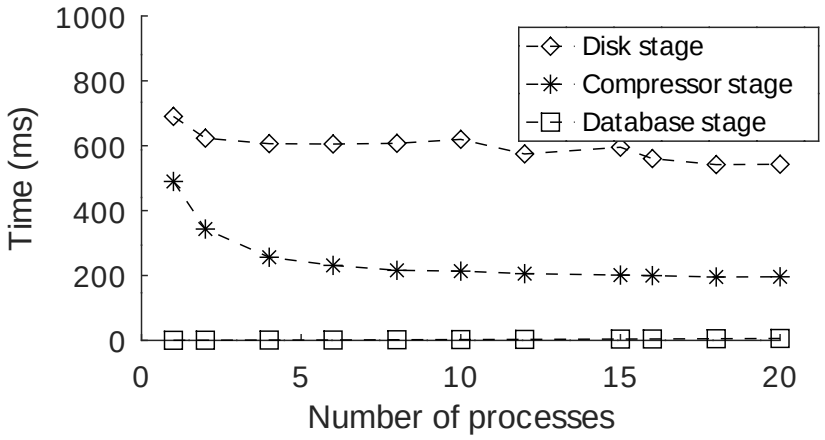


Figure 7.10 – Time for each stage of the pipeline without distributed futures

of an asynchronous call, or we would need a global clock to compare the time of invocation on the coordinator wiht the time of completion on another object. Note that resolving a future is more costly for normal futures than for distributed futures. Indeed, the former requires communication of all data while the latter only requires exchange of metadata. To minimize the communication overhead between components, we choose the same number of processes for all active objects, which avoids a communication bottleneck on either side of the distributed future. 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 and plot the average time each pipeline stage takes, Figure 7.9 uses our distributed futures and Figure 7.10 does not.

We can clearly see that the components scale better with distributed futures. The database has very little data to receive and very little work that can be performed in parallel. However, its memory use increases as more images are indexed, which would necessitate, in large deployments, its distribution over multiple servers. The disk and compressor stages benefit the most from distributed futures, because they deal with large data streams. Without distributed futures, the disk stage has to gather the whole image in one process to transmit it to the compressor object through a regular future. Figure 7.10 shows that both stages are overwhelmed by this transfer. Adding more resources for reading and compression does not show significant gain from 4 processes per stage. The experiment also shows that the performance of the distributed future version keeps noticeably improving up to 16 processes.

We now need to decide the number of components we want To pick a resource allocation, we compare the time taken by each pipeline stage depending on the number of processes used by each active object. We first observe that the pipeline's potential is related to the slowest stage of the pipeline. We thus place ourselves in conditions that minimize the time taken by the slowest stage. For the distributed future version, this point is at sixteen processes for the compression stage, while it is at eighteen processes for the disk stage for the version without distributed futures. Considering the number of processes for the slowest stage that minimizes its duration, we can take a lower number of processes for each of the other stages, as slowing down other stages does not

change the time taken for the whole pipeline to process an image. For distributed futures however, as parallelism is also achieved in the communications between the pipeline sections, the approach described above is only approximate and thus needs to be adapted. As there is no systematic methodology, we augment the number of processes used by the more data intensive stages, even if they are faster. Consequently, taking also sixteen processes for the disk object is a safe choice because reducing it will increase the time the compressor stage will take to receive its input. The choice of number of processes for the database component is guided by the amount of required memory.

We now run the same pipeline scenario, but we measure the performance of the whole pipeline (we thus remove the intermediate synchronizations after each call on code shown of Figure 7.8). We now take a configuration of twenty database processes, sixteen disk processes, and we only vary the number of compressor processes. Figure 7.11 shows the performance of these pipeline configurations. Because of the pipeline effect, the performance improvement until sixteen compressor processes with distributed futures confirms both our resource allocation choice and that the compressor component is the bottleneck of this pipeline. This experiment clearly shows the advantages of distributed futures, illustrating the gain brought by the parallel data transfers.

We also evaluate scalability with respect to the image size. We still take 1000 of them. Following our previous reasoning, we assign sixteen processes to the disk object, sixteen processes to the compressor object and twenty database processes to our database object to use two nodes. We have seen that these are interesting settings both for distributed futures and for standard futures. Figure 7.12 shows the execution time of our pipeline, Figure 7.13 shows a focus on smaller image sizes. As expected, the total time is proportional to the image size for both versions (with and without distributed futures). The version with distributed futures benefits from a much higher throughput. We attribute this again to the parallel transfer of data. We also see that, on small image sizes, where there is not much data to be transferred, distributed futures do not bring any performance gain, and are even a bit slower because data transfers involve more communications (to ask for the data) and consequently, the communication latency is higher for distributed futures than for standard ones. Further improvement concerning latency is left for future work. In particular, a bet-

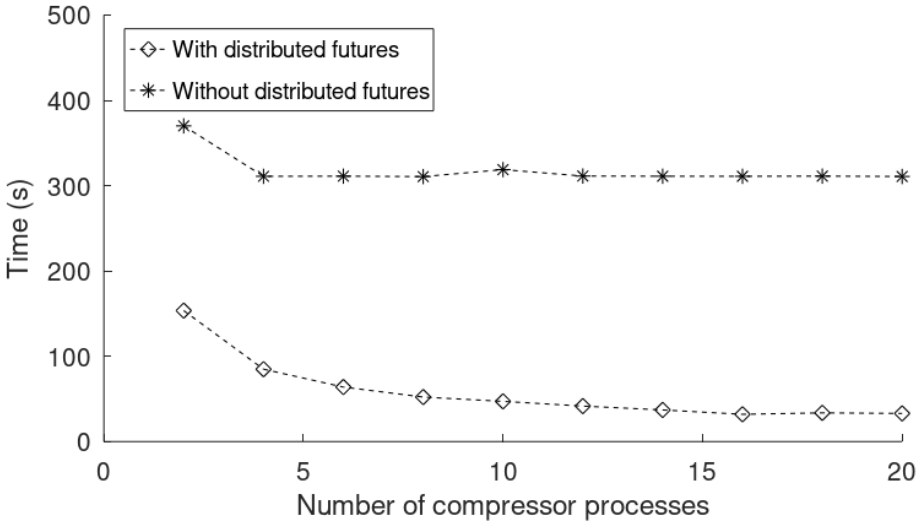


Figure 7.11 – Execution time for inserting 1000 images as function of the number of compressor processes, with 16 disk processes and 20 database processes

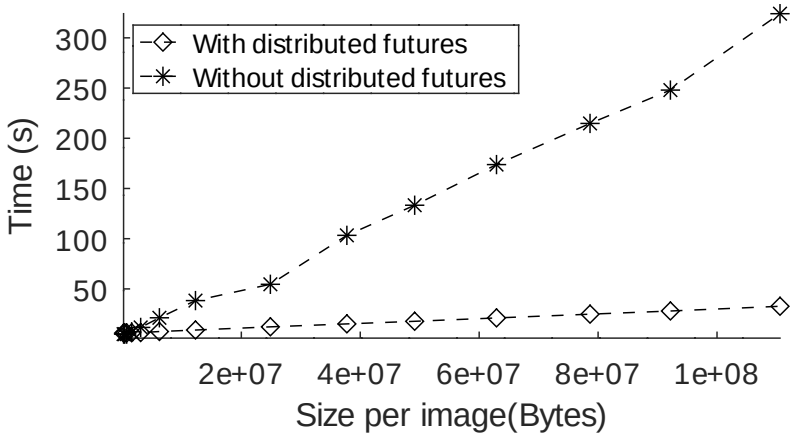


Figure 7.12 – Time for inserting 1000 images as function of the image size, with 16 disk processes, 16 compressor processes and 20 database processes

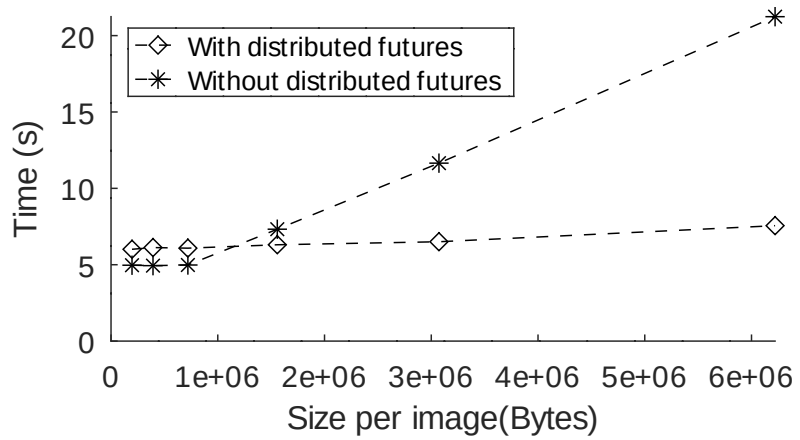


Figure 7.13 – Time for inserting 1000 images as function of the image size, with 16 disk processes, 16 compressor processes and 20 database processes

ter integration of `Future` and `vector_distribution` could reduce the number of synchronizations required before obtaining the data, which would reduce the latency and thus improve performance for smaller image sizes. Also, we have made a preliminary investigation on a prefetch strategy for distributed futures, but as no conclusive result have been obtained, this research direction is kept as future works.

7.5 Conclusion

In this chapter, we have shown the performance of our BSP active object implementation, including distributed futures. While we showed basic performance measurements through small communication benchmarks, we also showed a more complex scenario.

In particular, our experiments allow us to conclude that our distributed future implementation performs better than our futures. We note that this is especially the case as the data size increases, but our distributed futures perform worse for smaller data sizes. This is expected as distributed futures require more communications, but the fact that our implementation is not fully optimized could also explain this result. This validates the distributed future approach, at least for communicating large sets of data between data parallel computations.

Our benchmarks also show that our implementation of BSP active objects performs reasonably well and can be considered as a promising first implementation of a programming model combining data and task parallelism while ensuring good properties for the programmer.

Chapter 8

Conclusion

8.1 Summary

In this thesis, we have presented our unified BSP active object model. We study this model both from a theoretical and a practical point of view. We have seen that this model may be used to express a combination of task-parallel and data-parallel algorithms. The strength of our model is thus to allow to program applications that can benefit from these two main models of parallelisation, benefiting from the advantages of the two models. Our hybrid model relies on active objects and BSP because these two programming paradigms are both relatively high-level and easy to program compared to others in their categories. These models also come with interesting properties, such as determinism under certain condition and absence of data-race. We thus provide a programming framework that also have some interesting safety and programmability properties, as illustrated by the examples we provide in this manuscript.

We provided a formal language for BSP active objects and a clear semantics that may be understood independently of our implementation. This formalisation precisely specify the behaviour of our model and is precise enough to allow for formal proofs, even if proving properties of our model is outside the scope of this thesis.

In order to better integrate BSP and active objects, we designed distributed futures. Our

distributed future concept allows a single future to represent a distributed vector; this allow the efficient parallel transfer of distributed data, while providing a synchronisation pattern based on futures.

In order to evaluate our model from a practical point of view, we developed an implementation in C++ that communicate through MPI in a distributed environment. We used this implementation to observe the performance improvement brought by distributed futures over classical futures. Our experiments were satisfactory in the sense that our model scales well, and even if our implementation could be optimised, it is sufficient to show the effectiveness of our approach. Our model allows the easy coupling of data-parallel and task-parallel computation in an easy-to-use and efficient manner.

8.2 Concluding remarks

Through the study of our hybrid model between BSP and active objects, we have realized that a naive integration of these two models raised some issues. In particular, our base model could produce results in parallel, but it was not appropriate for communicating them efficiently between different active objects. Futures, which are usually employed in actor and active object languages, formed an obstacle to the efficient communication of the distributed data produced by our parallel BSP active objects. A solution based on array of futures is usually employed in active object programs in order to have several active objects produce result parts in parallel. Since our aim was to have a single active object work in parallel to produce a result, this solution was not appropriate. This is why we opted for what we believe is a better integration of data-parallelism: distributed futures which represent a distributed result with a single future.

In our model and implementation, we aimed to strike a middle ground between programmability and performance. Because of this, an implementation could be rethought in a context where an application has specific needs. For example, an application requiring more performance could sacrifice the responsiveness of active objects by eliminating the management threads we used in our implementation. This would indeed require fewer data transfers between threads, but it would require the programmer to have a precise knowledge of communications between active objects, as

one could remain blocked when calling another at the wrong time. Another way to eliminate the management thread would be to implement BSP active objects in a shared-memory-only setting, thus sacrificing the distributed aspect of our implementation. Programmability could also be improved, for example by implementing our model in a higher level programming environment than the C++/MPI environment we used. While our implementation kept a middle ground between performance and programmability, it enabled us to better study our BSP active model and to demonstrate the increase in performance of our distributed futures over classical futures.

8.3 Perspective

The works and contributions of this thesis may be used to base further studies. This section lists the leads that are immediately clear to us.

While we formalized our hybrid BSP active object model into our core *ABSP* formal language, we did not prove that the properties of BSP and active object still remain in our hybrid model. This would be a valuable addition, and our formal language should provide a good basis for doing so. Also, the formalization of our BSP active object model does not include distributed futures. We focused on the implementation aspects of distributed futures and we did not have the time to formalize them into our core *ABSP* language. It would be interesting to extend our formal language and operational semantics in order to reason about distributed futures in a more formal context and thus enable the proof properties for programs that use them.

Our distributed futures require the user to trigger two synchronizations: one for synchronizing the metadata (through `get`) and another for synchronizing the data itself (`through get_part`). A better integration of our `vector_distribution` structure and futures could allow these two steps to be merged. However, the size of a distributed vector is provided to the programmer through this `vector_distribution` structure. Not providing this structure to the programmer through `get` would require the programmer to specify how the data is to be distributed into a parallel active object without knowing the size of this data. Pre-defined distribution types would allow the user to do so. The user would then only need to specify what distribution type a parallel object should use

when receiving the data associated to a distributed future. This would reduce the synchronization requirement before obtaining the data, and also make distributed futures easier to program thanks to a modified API integrating these distribution policies.

Active objects enable the programming of pipelines of components that are wired together through method calls and future synchronizations. When a first component in a pipeline produces its result faster than a second component requiring it, this result could be sent right after it is produced even if the second component is still processing a request. This would save communication time when this second component requests this data as it would already be communicated. This idea is called prefetching and it is not new in the context of active objects. However, doing prefetching in the context of our parallel actors producing distributed results through distributed futures would raise additional challenges. When prefetching, the data communication is not initiated while handling a request. This is why prefetching distributed futures would require a different way to specify how the distributed data is to be redistributed from a BSP active object to another. For example, the aforementioned distribution policies, would be interesting candidates for doing so. We conducted initial experiments with prefetching and hard-coded distribution policies but could not reach interesting improvements in the performance nor decide whether better performance could be reached or not. This is why these investigations are not presented in the present manuscript and prefetching is left for future works.

While our formalization and implementation are specific to a particular active object type (close to ProActive) and BSPlib, our key concepts are independent of them. For example, one could get inspiration from our model to develop a CUDA active object model for GPU programming. In this context, active objects could be used to coordinate GPU parallel programs. This particular example would require to rethink some aspects of our model, including the role of the head process. Indeed the head process would most likely not participate in a GPU computation in this context, as it might be more appropriate to associate it to a CPU coordinating this GPU execution. Extra synchronization mechanisms may then need to be introduced in order to implement this idea. This is because an implementation of this design could have the head process run asynchronously from the GPU processes it coordinates. The head process would then need to coordinate the exchange

of data that the GPU threads produce, which might raise interesting problems.

All along the development of our BSP active object implementation, we have attempted to integrate more convincing code examples than the scenarios we have presented in this thesis. From a general point of view, such attempts are time and energy consuming because parallel applications belong to huge projects and their study also require expertise in specific domains. We have tried, for example, to explore parallel solver coupling frameworks such as [3]. But the attempts we have made were inconclusive due to lack of time and expertise in these domains. It would be interesting to find and implement richer applications that would be suited to our model. This would allow us to better evaluate our model, and possibly highlight the need for extensions or improvements.

Bibliography

[1] Akka. `https://akka.io/`.

[2] Apache giraph. `http://giraph.apache.org`.

[3] precice homepage. `https://www.precice.org/`.

[4] Gul A Agha. Actors: A model of concurrent computation in distributed systems. Technical report, Massachusetts Inst of Tech Cambridge Artificial Intelligence Lab, 1985.

[5] Keyvan Azadbakht, Frank S. de Boer, and Vlad Serbanescu. Multi-threaded actors. In *Proceedings 9th Interaction and Concurrency Experience, ICE 2016, Heraklion, Greece, 8-9 June 2016.*, pages 51–66, 2016.

[6] Olivier Ballereau, Frédéric Loulergue, and Gaétan Hains. High level bsp programming: Bsml and bslambda. In *Heriot-Watt University*, pages 29–38. Intellect Books, 2000.

[7] F. Baude, D. Caromel, C. Dalmasso, M. Danelutto, V. Getov, L. Henrio, and C. Pérez. Gcm: a grid extension to fractal for autonomous distributed components. *annals of telecommunications - annales des télécommunications*, 64(1), 2009.

[8] Françoise Baude, Denis Caromel, Nathalie Furmento, and David Sagnol. Optimizing remote method invocation with communication–computation overlap. *Future Generation Computer Systems*, 18(6):769 – 778, 2002.

[9] P. H. Beckman, P. K. Fasel, W. E. Humphrey, and S. M. Mniszewski. Efficient coupling of parallel applications using paws. In *Proceedings. The Seventh International Symposium on High Performance Distributed Computing (Cat. No.98TB100244)*, pages 215–222, July 1998.

[10] P. A. Bernstein and S. Bykov. Developing cloud services using the orleans virtual actor model. *IEEE Internet Computing*, 20(5):71–75, Sept 2016.

[11] Phil Bernstein, Sergey Bykov, Alan Geller, Gabriel Kliot, and Jorgen Thelin. Orleans: Distributed virtual actors for programmability and scalability. Technical report, 2014.

[12] R.H. Bisseling. *Parallel Scientific Computation: A Structured Approach Using BSP and MPI.* OUP Oxford, 2004.

[13] 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. A survey of active object languages. *ACM Comput. Surv.*, 50:76:1–76:39, 2017.

[14] Lars Ailo Bongo. Bulk synchronous visualization. In *Proceedings of the 2013 International Workshop on Programming Models and Applications for Multicores and Manycores*, PMAM '13, pages 21–30, New York, NY, USA, 2013. ACM.

[15] Olaf Bonorden, Ben Juurlink, Ingo von Otte, and Ingo Rieping. The paderborn university bsp (pub) library. *Parallel Comput.*, 29(2):187–207, 2003.

[16] W. Bousdira, F. Gava, L. Gesbert, F. Loulergue, and G. Petiot. Functional parallel programming with revised bulk synchronous parallel ml. In *2010 First International Conference on Networking and Computing*, pages 191–196, 2010.

[17] William W Carlson, Jesse M Draper, David E Culler, Kathy Yelick, Eugene Brooks, and Karen Warren. Introduction to UPC and language specification. Technical Report CCS-TR-99-157, IDA Center for Computing Sciences, 1999.

[18] Denis Caromel, Ludovic Henrio, and Bernard Paul Serpette. Asynchronous and deterministic objects. In *Proceedings of the 31st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '04, pages 123–134, New York, NY, USA, 2004. ACM.

[19] Denis Caromel, Wilfried Klauser, and Julien Vayssière. Towards seamless computing and metacomputing in java. *Concurrency: Practice and Experience*, 10(11-13):1043–1061, 1998.

[20] Avery Ching, Sergey Edunov, Maja Kabiljo, Dionysios Logothetis, and Sambavi Muthukrishnan. One trillion edges: Graph processing at facebook-scale. *Proc. VLDB Endow.*, 8(12):1804–1815, 2015.

[21] Murray Cole. *Algorithmic Skeletons: Structured Management of Parallel Computation*. MIT Press, Cambridge, MA, USA, 1991.

[22] Leonardo Dagum and Ramesh Menon. Openmp: An industry-standard api for shared-memory programming. *IEEE Comput. Sci. Eng.*, 5(1):46–55, 1998.

[23] Kostadin Damevski and Steven Parker. M x n data redistribution through parallel remote method invocation. *IJHPCA*, 19:389–398, 11 2005.

[24] Jeffrey Dean and Sanjay Ghemawat. Mapreduce: Simplified data processing on large clusters. In *6th Symposium on Operating System Design and Implementation (OSDI 2004), San Francisco, California, USA, December 6-8, 2004*, pages 137–150, 2004.

[25] Kiko Fernandez-Reyes, Dave Clarke, Elias Castegren, and Huu-Phuc Vo. Forward to a promising future. In Giovanna Di Marzo Serugendo and Michele Loreti, editors, *Coordination Models and Languages*, pages 162–180, Cham, 2018. Springer International Publishing.

[26] Kiko Fernandez-Reyes, Dave Clarke, and Daniel S. McCain. Part: An asynchronous parallel abstraction for speculative pipeline computations. In *Coordination Models and Languages*, pages 101–120. Springer International Publishing, 2016.

[27] Christian Foisy and Emmanuel Chailloux. Caml flight: A portable SPMD extension of ML for distributed memory multiprocessors. In A. P. Wim Bohm and John T. Feo, editors, *High Performance Functional Computing Proceedings*, pages 83–96, Apr 1995.

[28] F. Gava and J. Fortin. Formal semantics of a subset of the paderborn's bsplib. In *2008 Ninth International Conference on Parallel and Distributed Computing, Applications and Technologies*, pages 269–276, 2008.

[29] Frédéric Gava. Formal proofs of functional bsp programs. *Parallel Processing Letters*, 13(03):365–376, 2003.

[30] David Gelernter. Generative communication in linda. *ACM Trans. Program. Lang. Syst.*, 7(1):80–112, 1985.

[31] David Gelernter and Nicholas Carriero. Coordination languages and their significance. *Commun. ACM*, 35(2):97–107, 1992.

[32] Simon Gibbs. Composite multimedia and active objects. In *Conference Proceedings on Object-oriented Programming Systems, Languages, and Applications*, OOPSLA '91, pages 97–112, New York, NY, USA, 1991. ACM.

[33] Aleksey Gurtovoy and David Abrahams. The boost c++ metaprogramming library. *cit. on*, page 22, 2002.

[34] Reiner Hähnle. *The Abstract Behavioral Specification Language: A Tutorial Introduction*, pages 1–37. Springer Berlin Heidelberg, 2013.

[35] Gaétan Hains. Subset synchronization in bsp computing. In *PDPTA*, volume 98, pages 242–246, 1998.

[36] Robert H. Halstead, Jr. Multilisp: A language for concurrent symbolic computation. *ACM Trans. Program. Lang. Syst.*, 7(4):501–538, 1985.

[37] Ludovic Henrio, Fabrice Huet, and Zsolt István. Multi-threaded active objects. In Rocco De Nicola and Christine Julien, editors, *COORDINATION*, volume 7890 of *Lecture Notes in Computer Science*, pages 90–104. Springer, 2013.

[38] Ludovic Henrio and Justine Rochas. Declarative scheduling for active objects. In *Proceedings of the 29th Annual ACM Symposium on Applied Computing*, SAC '14, pages 1339–1344, New York, NY, USA, 2014. ACM.

[39] Jonathan M. D. Hill, Bill McColl, Dan C. Stefanescu, Mark W. Goudreau, Kevin Lang, Satish B. Rao, Torsten Suel, Thanasis Tsantilas, and Rob H. Bisseling. Bsplib: The bsp programming library. *Parallel Comput.*, 24:1947–1980, December 1998.

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

[41] E. B. Johnsen and O. Owe. An asynchronous communication model for distributed concurrent objects. In *Proceedings of the Second International Conference on Software Engineering and Formal Methods, 2004. SEFM 2004.*, pages 188–197, 2004.

[42] Terry Jones. Linux kernel co-scheduling for bulk synchronous parallel applications. In *Proceedings of the 1st International Workshop on Runtime and Operating Systems for Supercomputers*, ROSS '11, pages 57–64, New York, NY, USA, 2011. ACM.

[43] K. Keahey, P. Fasel, and S. Mniszewski. Paws: collective interactions and data transfers. In *Proceedings 10th IEEE International Symposium on High Performance Distributed Computing*, pages 47–54, 2001.

[44] Christoph W. Kessler. Neststep: Nested parallelism and virtual shared memory for the bsp model. *The Journal of Supercomputing*, 17(3):245–262, 2000.

[45] Christoph W. Kessler. Managing distributed shared arrays in a bulk-synchronous parallel programming environment. *Concurrency and Computation: Practice and Experience*, 16(2-3):133–153, 2004.

[46] Muhammad Uzair Khan and Ludovic Henrio. First Class Futures: a Study of Update Strategies. Technical Report RR-7113, INRIA, 2009.

[47] Jin-Soo Kim, Soonhoi Ha, and Chu Shik Jhon. Relaxed barrier synchronization for the BSP model of computation on message-passing architectures. *Information Processing Letters (IPL)*, 66(5):247–253, 1998. Attempt to remove global synchronization in BSP, exchange it with process-to-process exchanges. Requires using only PUT operations.

[48] R. Greg Lavender and Douglas C. Schmidt. Pattern languages of program design 2. In John M. Vlissides, James O. Coplien, and Norman L. Kerth, editors, *Pattern languages of program design 2*, chapter Active Object: An Object Behavioral Pattern for Concurrent Programming, pages 483–499. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1996.

[49] B. Liskov and L. Shrira. Promises: Linguistic support for efficient asynchronous procedure calls in distributed systems. In *Proceedings of the ACM SIGPLAN 1988 Conference on Programming Language Design and Implementation*, PLDI '88, pages 260–267, 1988.

[50] Frédéric Loulergue, Gaétan Hains, and Christian Foisy. A,calculus of functional bsp programs. *Science of Computer Programming*, 37:253–277, 05 2000.

[51] Grzegorz Malewicz, Matthew H Austern, Aart JC Bik, James C Dehnert, Ilan Horn, Naty Leiser, and Grzegorz Czajkowski. Pregel: a system for large-scale graph processing. In *Proceedings of the 2010 ACM SIGMOD International Conference on Management of data*, SIGMOD '10, pages 135–146, New York, NY, USA, 2010. ACM, ACM.

[52] W. F. McColl. Parallel algorithms lecture notes. 1997.

[53] W.F. McColl. Scalable computing. In Jan van Leeuwen, editor, *Computer Science Today*, volume 1000 of *LNCS*. Springer, 1995.

[54] Mohan Nibhanupudi, Charles D. Norton, and Boleslaw K. Szymanski. Plasma simulation on networks of workstations using the bulk-synchronous parallel model. In *in Proceedings of the International Conference on Parallel and Distributed Processing Techniques and Applications*, pages 13–22, 1995.

[55] Christophe Scholliers, Éric Tanter, and Wolfgang De Meuter. Parallel actor monitors: Disentangling task-level parallelism from data partitioning in the actor model. *Sci. Comput. Program.*, 80:52–64, Feb 2014.

[56] K. Siddique, Z. Akhtar, E. J. Yoon, Y. S. Jeong, D. Dasgupta, and Y. Kim. Apache hama: An emerging bulk synchronous parallel computing framework for big data applications. *IEEE Access*, 4:8879–8887, 2016.

[57] D. B. Skillicorn. *Predictable Parallel Performance: The BSP Model*, pages 85–115. Springer US, Boston, MA, 2002.

[58] J. L. Sobral and C. A. Cunha. An annotation-based framework for parallel computing. *16th Euromicro Conference on Parallel, Distributed and Network-Based Processing (PDP 2008)*, pages 113–120, 2007.

[59] Wijnand Suijlen. Bsponmpi. `https://github.com/wijnand-suijlen/bsponmpi`. Accessed: 2020-02-26.

[60] Wijnand Suijlen and A. N. Yzelman. Lightweight parallel foundations: a model-compliant communication layer. *ArXiv*, abs/1906.03196, 2019.

[61] Samira Tasharofi, Peter Dinges, and Ralph E. Johnson. Why do scala developers mix the actor model with other concurrency models? In *Proceedings of the 27th European Conference on Object-Oriented Programming*, ECOOP'13, pages 302–326, 2013.

[62] Julien Tesson and Frédéric Loulergue. Formal semantics of DRMA-style programming in BSPlib. In *Parallel Processing and Applied Mathematics*, pages 1122–1129. 2008.

[63] William Thies, Michal Karczmarek, and Saman Amarasinghe. *StreamIt: A Language for Streaming Applications*, pages 179–196. Springer Berlin Heidelberg, Berlin, Heidelberg, 2002.

[64] Yuanyuan Tian, Andrey Balmin, Severin Andreas Corsten, Shirish Tatikonda, and John McPherson. From "think like a vertex" to "think like a graph". *Proc. VLDB Endow.*, 7(3):193–204, November 2013.

[65] Alexandre Tiskin. A new way to divide and conquer. *Parallel Processing Letters*, 11(04):409–422, 2001.

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

[67] Leslie G. Valiant. A bridging model for multi-core computing. In Dan Halperin and Kurt Mehlhorn, editors, *Algorithms - ESA 2008, 16th Annual European Symposium, Karlsruhe, Germany, September 15-17, 2008. Proceedings*, volume 5193 of *Lecture Notes in Computer Science*, pages 13–28. Springer, 2008.

[68] A. N. Yzelman, R. H. Bisseling, D. Roose, and K. Meerbergen. Multicorebsp for c: A high-performance library for shared-memory parallel programming. *International Journal of Parallel Programming*, 42(4):619–642, 2014.

[69] Matei Zaharia, Mosharaf Chowdhury, Michael J. Franklin, Scott Shenker, and Ion Stoica. Spark: Cluster computing with working sets. In *Proceedings of the 2Nd USENIX Conference on Hot Topics in Cloud Computing*, HotCloud'10, pages 10–10, 2010.