Università di Pisa and Scuola Superiore Sant'Anna

Department of Computer Science

Master's Degree in Computer Science and Networking

A Bulk Synchronous Parallel pattern in FastFlow

Candidate:
Orlando Leombruni

Supervisor:
Prof. Marco Danelutto

Academic Year 2018/2019

Ringraziamenti

Si chiude qui un altro importante capitolo della mia vita. Il percorso di questa Laurea Magistrale è stato lungo e a volte difficile, ma non sono pentito delle mie scelte: ho imparato tanto, sia a livello di conoscenze che a livello umano. Ho avuto al mio fianco persone con cui ho condiviso gioie, soddisfazioni, piccoli e grandi traguardi, e perché no anche un po' di tristezza[1]. Purtroppo la situazione attuale non mi permette di festeggiare con queste persone — che con ogni probabilità sono le stesse che stanno leggendo ora questa tesi — il raggiungimento di un risultato così importante: spero che queste poche righe di ringraziamento possano comunque esprimere un minimo la mia stima ed affetto verso di loro.

Un sentito ringraziamento al Prof. Marco Danelutto, per la pazienza ed il supporto dimostratomi durante il lavoro di tesi.

Un grazie gigantesco a mamma Mara, di cui mi basta sentire la voce per riacquistare tranquillità. Grazie di cuore *madreh*, mi sei sempre stata vicina anche a 500 chilometri di distanza e non mi hai mai fatto mancare nulla. (E mi regali un sacco di biglietti di concerti, cosa molto importante.)

Subito dopo — ma non per importanza, ovviamente — vorrei ringraziare Alessandra, che mi accompagna tenendomi la mano praticamente dall'inizio di questa avventura universitaria. Si dice che al fianco di ogni grande uomo vi è una grande donna: io non sono di certo un grande uomo (beh, stazza a parte), ma sono sicuro che al mio fianco c'è una donna superlativa.

Un grazie enorme a nonna Maria, che non ha mai mancato di farmi sentire il suo affetto e il suo supporto. Abbiamo superato tante difficoltà assieme, abbiamo chiacchierato approssimativamente cinquemila ore al telefono da quando sono a Pisa, e spero di poter continuare a farlo ancora per molto.

Un grazie a *fratello* Maurizio, a cui ogni tanto tocca sorbirsi le mie lamentele videoludiche e non solo.

Un grazie di cuore a Valerio e alla sua famiglia per esserci sempre, anche se ci sentiamo poco. (Un po' per colpa mia, un po' perché non so mai qual è il tuo numero di telefono, caro Zione.)

Un ringraziamento particolare va a Francesco, Maria e Marco, che mi hanno sempre riservato un'accoglienza e un calore senza pari. Sono contento che le nostre strade si siano incrociate.

Un grazie speciale ad Antonio e Mattia, amici di lunghissima data su cui so di poter contare nonostante la lontananza. (Ehi, ora posso prendervi in giro in quanto extracomunitari! Bella 'sta Brexit.)

Un grazie ad Alfredo, ~~Lamberto~~Luigi e (l'altro) Mattia, per le serate spassose e per le chat deliranti di cui non posso scrivere il nome, perché Cricca Panzoni non è un nome adatto per essere stampato su una tesi. (Ops.)

Un grazie a Marco (quello *tetesko*), perché anche se ci sentiamo poco la nostra amicizia resta sempre salda come ai vecchi tempi.

Un grazie ai "Mammut di Proust" Francesco, Lorenzo e Matteo per la compagnia e i meme sull'informatica (ma non solo).

Un ringraziamento speciale ad Alberto e Paola per la professionalità, la cortesia e la pazienza dimostratami.

Un grazie a tutti quelli che hanno condiviso con me un pezzetto del loro percorso di vita, sia pure solo virtualmente. Vorrei ringraziarvi ad uno ad uno, ma lo spazio purtroppo è poco. Confido nella vostra comprensione.

Vorrei infine dedicare questo traguardo a nonno Vincenzo, che sono sicuro sarà fiero di me ovunque egli sia. Ciao nonno, mi manchi.

Orlando Leombruni

[1] Che comunque non va vissuta come un valore negativo, come insegnano gli Elii.

Contents

List of Figures

List of Listings

Chapter 1

Introduction

Writing parallel software, even in its simplest form, is harder than writing sequential software. The amount of details that need to be taken into account is much higher, and traditional programming languages are for the most part designed for sequential programs. In these languages, support for parallel computations is not a central focus and is often limited to a set of basic mechanisms for spawning and synchronizing parallel executors.

More recently, however, parallel computing has started to become a central element of the technological landscape. Commodity processors feature lower operating frequencies and an increasing number of cores with respect to the ones available ten years ago [52]. GPUs, which feature a high number of computing units that can only perform basic arithmetic operations, are increasingly used as coprocessors to accelerate heavy numerical algorithms [15, 44]. Smart devices run on low-power multicore SoCs (often equipped with small GPUs) that greatly benefit from a balanced computational load [58]. The ability to write good parallel code is therefore becoming a basic tool in a programmer's arsenal.

We believe that there are two fundamental ingredients in efficient parallel programs: a **clear design** of the computation to be performed and a **suitable and efficient framework** for its implementation. The role of the second ingredient is often played by libraries and APIs paired with existing sequential languages. One such example is *FastFlow*, a C++ library developed at University of Pisa and University of Torino [3], which provides the programmer with easy-to-use mechanisms for the design and implementation of parallel programs.

Designing the parallel algorithm is not an easy task either. One can start assuming virtually no limitations on what a parallel computer can do, i.e. to have infinite processing elements and infinite memory without concerns to regulations of concurrent accesses. The algorithms designed under these assumptions, however, cannot be directly implemented "as-is", since the actual hardware *does have* limitations.

The Bulk Synchronous Parallel paradigm was introduced by Leslie G. Valiant in the late 1980s as a model with realistic limitations on parallel hardware features [61]. BSP algorithms are divided in *supersteps*: every processing element must wait that its peers complete the current superstep before advancing the computation to the next one. Moreover, while communications with other processing elements can be scheduled at any time, they actually only take place after the end of the superstep. Together, these two constraints impose a "superstructure" to parallel computations, which makes it easier to reason about the effects of operations.

The BSP model has not been widely adopted, at least in comparison to other parallel models like PRAM and dataflow, until recent times. For a long while its use was limited to resolution of numerical problems and niche interests in theoretical computer science. Since the early 2010s, however, the BSP model got a "second life" having been adopted in Google Pregel [36], a framework for distributed large-scale graph computing, due to its clear and effective programming style and performance model. Nowadays, it is implemented in a number of frameworks for both distributed (Apache Hama [7], Apache Giraph [6]) and parallel (MulticoreBSP [67, 66], Bulk [16]) computing.

We believe that BSP is a good model for designing efficient parallel algorithms. Its role as a "bridging model" also allows for direct execution of those algorithms on a suitable runtime support, therefore freeing the programmer from having to adapt them to specific frameworks' programming models. Unfortunately, there are few BSP implementations that offer modern features like automatic memory management on multicore, shared memory architectures. Therefore, we considered implementing a BSP library that offers the abovementioned modern features, with an API that is simple to use and a run-time support that frees users from menial tasks like having to register shared variables. We chose FastFlow as the underlying framework due to its easy-to-use algorithmic skeletons and its compositional capabilities. Since the BSP pattern is not currently provided by FastFlow, we also decided to make the library a fully-featured FastFlow component, able to be used inside a broader computation.

The objective of this work therefore consists in the design and implementation of a modern C++ parallel programming library providing the BSP model *on top of* and *within* the FastFlow framework. The library will target shared-memory multicore machines. We want to provide programmers with an easy-to-use interface that allows them to design and code efficient BSP programs. Our implementation will fully support the object-oriented paradigm and will relieve the programmer from the burden of having to manually manage memory and shared variables. The library has been designed to be

- fully **BSP compliant —** existing BSP algorithms must be able to be easily ported to our library, and
- fully **FastFlow compliant —** the whole BSP computation will act as a single FastFlow parallel component, thus being able to become part of a broader FastFlow structured computation; it will receive input data from other FastFlow nodes and send output data to other FastFlow nodes.

The library will be itself implemented as a FastFlow graph, therefore exploiting the framework's compositional capabilities. This fact has helped in the implementation design phase, as solutions based on different algorithmic skeletons for the library could be considered and prototyped quickly without having to change the internal structure of the nodes.

Our work started with an in-depth study of the state of the art, in the form of both theoretical advancements and practical implementations, to better understand the BSP model and its scope. After coming up with an initial draft, we carried on both the architectural and implementation design at the same time, one aspect influencing the other. Of course, this meant going through multiple revisions of the library, from a rough prototype – which aggressively enforced type correctness between a superstep's output and its successor's input – up to the final version presented in this thesis (Figure 1.1), sporting good scalability as well as highly expressive API and a run-time support that relieves the user from burdens like memory management and variable registration. We took inspiration for some of the features from "competitors", mainly MulticoreBSP for Java, but ultimately we designed both the implementation structure and the API from scratch. Lastly, we performed extensive tests for both correctness and performance of our implementation, achieving good results for the latter.

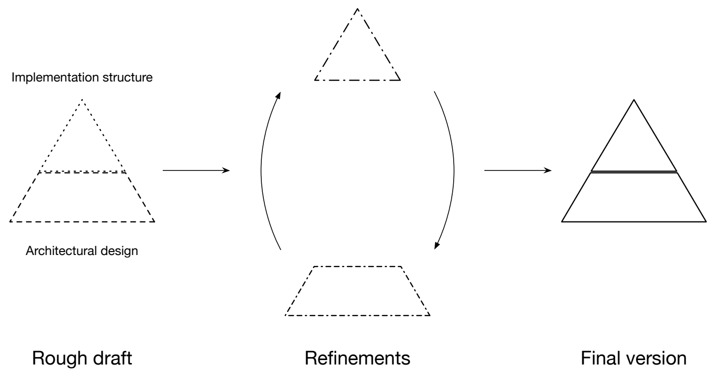


Figure 1.1: Workflow for the design and implementation of the library.

The rest of the thesis is organized as follows.

- **Chapter 2** provides the reader with the necessary concepts used throughout the work. The theory behind the Bulk Synchronous Parallel model is laid out, together with the relevant definitions and theorems that help frame the model into the bigger picture of parallel programming. The FastFlow framework is also introduced, together with its main features, to provide familiarity with some of the terminology and mechanisms presented in the rest of the thesis. A brief digression on a memory allocation strategy provided by FastFlow and used in the library closes the Chapter.

- **Chapter 3** introduces the architectural model of our implementation. This model is composed by various entities: the design of each entity is discussed, along with a description of how it fits and behaves inside the architecture.

- **Chapter 4** presents an overview of the library's implementation. Each class is described in depth, focusing on its peculiarities and relation with respect to the entities of Chapter 3. Extensive code snippets detailing the main portions of the implementation are also provided and discussed.

- **Chapter 5** contains the experimental results from the various test programs we implemented. The testing methodology and machine architecture are detailed at the start of the Chapter. For each test program we define the communication scheme and input distribution, then we provide and discuss perfomance plots confronting the results obtained from our library with the corresponding MulticoreBSP for Java implementation.

- **Chapter 6** draws the conclusions relative to the whole work. The experimental outcomes are once again commented, and an outline for possible future works is provided.
- **Appendix A** contains the documentation for the library API, together with some clarifications about advanced mechanisms such as direct access methods.
- **Appendix B** contains the source code for the library and for the test programs. The source code is also available at [41].

Chapter 2

Background

This Chapter provides the reader with basic concepts relative the Bulk Synchronous Parallel (BSP) paradigm. The BSP model has been designed in the late 1980s to overcome the limitations of the PRAM model. BSP divides a program in a series of supersteps that contain both computation performed by a number of processing elements and communication between them. Each superstep is separated from the others by means of a global synchronization (e.g. barrier). Recent developments on the paradigm (e.g. MultiBSP, MulticoreBSP) are touched upon.

An overview of the FastFlow framework closes the Chapter.

2.1 The Bulk Synchronous Parallel model

The Bulk Synchronous Parallel model was introduced by Leslie G. Valiant in 1990 [61] as a "bridging model" for distributed computing. Valiant considered the main reason for the "success" and ubiquity of the sequential computation the availability of a *central unifying model*, the Von Neumann computer. This unified view serves as a connecting bridge between software and hardware: software developers can focus on writing programs that run efficiently on it – abstracting from the complexities of the hardware. Conversely, hardware designers only have to realize efficient Von Neumann machines, without having to mind about the software that will be executed on them. Valiant thought that – in order to obtain a widespread adoption of distributed computing – a model with a similar purposes as Von Neumann's one for sequential computation was needed; he then introduced the BSP model as a viable candidate for this role.

NOTE. Valiant's original article actually references *parallel computing*, hence the name *Bulk Synchronous* ***Parallel***; nevertheless, it was written at a time when the predominant source of parallelism derived from the exploitation of an interconnection of processing elements, each with its own private memory and lacking a global clock or synchronization, in which data is exchanged by messages sent and received via the network — i.e. what is called today a "distributed architecture" [50]. (In fact, Valiant even mentions implementations of a BSP computer based on optical crossbars or packet-switched networks, reinforcing this concept.) Throughout the remainder of this text we will refer to

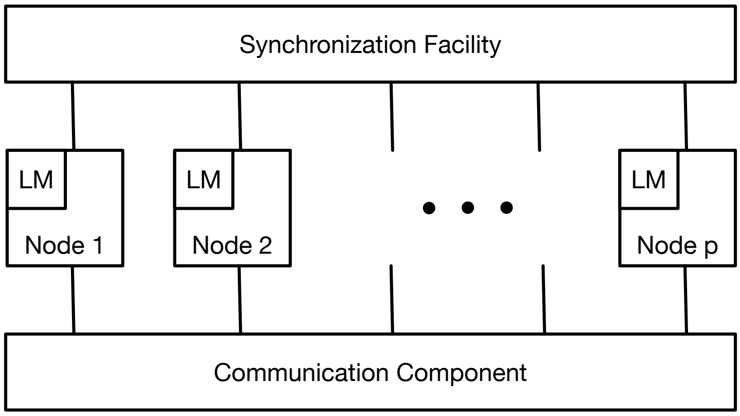


Figure 2.1: The architecture of a generic BSPC.

the modern-day definition of *parallel computing* (i.e. a single system with multiple processors and cores, with a shared memory and global clock), so Valiant's original BSP article will be treated as referencing a distributed architecture.

2.1.1 The BSP Computer

The BSP model defines – along the lines of the Von Neumann model – an abstract **Bulk Synchronous Parallel Computer**, or BSPC for short. A BSPC is the combination of three attributes:

1. a number of *nodes* for performing processing and memory operations;
2. a *communication component* that delivers point-to-point messages between nodes;
3. a *synchronization facility* that coordinates the behavior of the nodes.

A BSP computation consists in a sequence of **supersteps**. During a superstep, each node performs a task consisting of local computation and message sending primitives. The superstep ends when all nodes finish their task. The synchronization facility is the entity responsible for checking this condition. An important feature of this mechanism is that the effect of any communication primitive performed during a superstep will take place only during the successive superstep, i.e. the communication between nodes actually takes place at the end of the superstep.

The way the synchronization facility works has an impact on the performance of a BSPC. First, we define the following.

Definition 2.1. A *time unit* is the time spent by a node performing a single operation over data available in its local memory.

Definition 2.2 (Periodicity of a BSPC)**.** The *periodicity* L of a BSPC is the number of time units in the interval between two consecutive checks of the "superstep ended" condition.

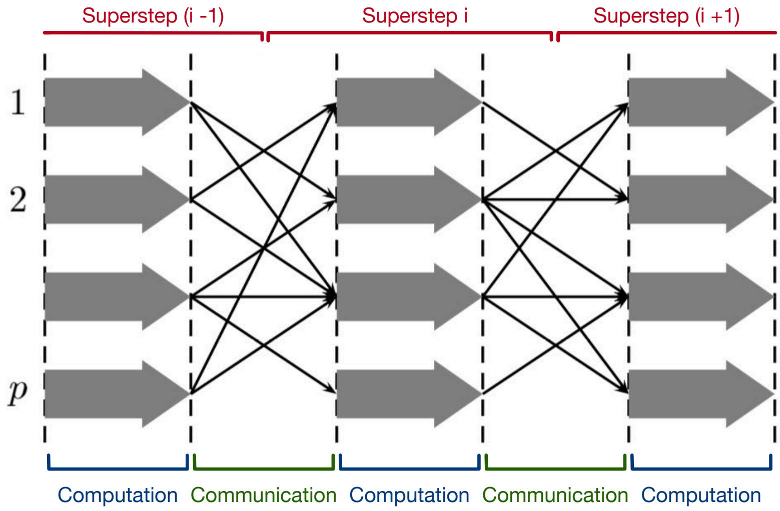


Figure 2.2: The superstep flow of a BSP computation.

In other words, the synchronization facility checks that the current superstep has ended every L time units. This definition is purposefully vague, since no assumption is made on how the synchronization facility actually works. For example, if the system can continuously check whether a superstep is completed, then L can be defined as the number of time units needed to perform the check itself. The periodicity concept also brings two interesting consequences:

- L is the minimum effective duration of a superstep (i.e. a superstep that theoretically completes in less than L time units will actually behave like a superstep that completes in L time units);
- the maximum efficiency for a BSPC of periodicity L can be achieved with a program that perfectly balances tasks in such a way that, in every superstep, every node completes its task in exactly L time units.

Depending on the synchronization facility, some portion of the periodicity L can be overlapped with local computations. In this case, with l we refer to the non-overlapping time interval of the periodicity.

The task executed by each node in a given superstep can be further divided into three distinct portions: the *input phase*, the *local computation phase* and the *output phase*. In the input phase, a node receives data in its local memory; then, some local computation is performed over this data; in the output phase, the node sends some data to other nodes. This functional partition of a task allows us to define more useful parameters for a BSPC.

Definition 2.3 (Local computation cost for a superstep)**.** Let i be a superstep. The *local computation cost* w_i is the maximum number of local operations performed by any node during superstep i. In other words, let w_{ij} be the number of local operations performed by node j during superstep i. Then we define

$$w_i = \max_j w_{ij}. \tag{2.1}$$

NOTE. Since we defined the time unit to be the time to perform a single operation it also holds that, for superstep i, w_i is the time spent in the local computation phase.

Definition 2.4 (Number of data units sent and received)**.** Let h'_{ij} and h''_{ij} be respectively the number of data units received and sent by node j during superstep i. We define the *maximum number of received data units* during i as

$$h'_i = \max_j h'_{ij} \tag{2.2}$$

and the *maximum number of sent data units* during i as

$$h''_i = \max_j h''_{ij}. \tag{2.3}$$

Definition 2.5 (h-relation)**.** Let i be a superstep. The *total data units* exchanged during this superstep by a generic node j is defined as

$$h_{ij} = h'_{ij} + h''_{ij}. \tag{2.4}$$

The maximum data units exchanged during superstep i is called the *h-relation* of i and is defined as

$$h_i = \max_j h_{ij}. \tag{2.5}$$

Sometimes it is also useful to define the *maximum h-relation* for a BSP program as

$$h = \max_i h_i. \tag{2.6}$$

We said before that the communication component is the entity responsible for delivering point-to-point messages (carrying data) between nodes. Without loss of generality, we can assume that this component will take a fixed amount of time to delivery a single data unit after reaching steady state.

Definition 2.6. Let s be the startup time of the communication component, i.e. the time needed to reach steady state, and $\bar{g}$ be its basic throughput when in continuous use, i.e. the time spent in delivering a single unit of data at steady state. Then, for superstep i, the communication component will take $\bar{g}h_i + s$ units of time.

From this definition it follows that in order to achieve optimality we need $\bar{g}h_i$ to be at least of the same order of magnitude as s, i.e.

$$\bar{g}h_i \geq s. \tag{2.7}$$

This condition is required, otherwise a lot of time is "wasted" in the startup phase of the communication component. This suggests a way to simplify the cost model.

Definition 2.7 (Gap of a BSPC)**.** Let $g = 2\bar{g}$. Then, it holds

$$gh_i = 2\bar{g}h_i \geq \bar{g}h_i + s \tag{2.8}$$

where the second inequality comes from equation 2.7. The parameter g is called *gap* or *communication throughput ratio* of a BSPC.

We now have all the necessary ingredients for a basilar cost model for a BSPC.

Definition 2.8 (Cost model for a superstep)**.** Let i be a superstep. Then, the cost for i (i.e. time spent in this superstep) can be modeled by the following equation:

$$c_i = w_i + gh_i + l. \tag{2.9}$$

Definition 2.9 (Cost model for a BSP program)**.** Let S be the number of supersteps of a BSP program. Then, the total time spent executing the program can be modeled by the following equation:

$$c = \sum_{i=1}^{S} c_i = \sum_{i=1}^{S} (w_i + gh_i + l) = \sum_{i=1}^{S} w_i + g \sum_{i=1}^{S} h_i + S \cdot l. \tag{2.10}$$

If we define

$$W = \sum_{i=1}^{S} w_i \qquad H = \sum_{i=1}^{S} h_i$$

then equation 2.10 can be simplified as

$$c = W + g \cdot H + S \cdot l \tag{2.11}$$

where W is called the *total local computation cost*, H the *total communication cost* and S the *total synchronization cost*.

Lastly, we define when a BSP program is *balanced*.

Definition 2.10 (Balanced BSP computation)**.** Let p be the number of nodes of a BSPC. We define the *local computation volume* $\mathcal{W}$ as the total number of local operations executed during the whole program. Similarly, we define the *communication volume* $\mathcal{H}$ as the total number of data units transferred between all nodes during the whole program. If both following conditions hold:

$$W = \mathcal{O}(\mathcal{W}/p) \qquad H = \mathcal{O}(\mathcal{H}/p) \tag{2.12}$$

then the BSP program is said to be *balanced*.

2.1.2 Towards an object-oriented shared memory BSP

The aim of this work is to develop a BSP implementation which focuses on *objects* as the atomic unit of data, designed for multicore shared-memory single machines. This scenario introduces its own set of complexities and challenges that have been tackled throughout the years by different people. This section presents the main concepts behind the most relevant adaptations of the BSP model for a shared-memory architecture.

PRAM simulations and the BSPRAM model

In Valiant's original article [61], particular emphasis was placed in providing an efficient way to simulate PRAM algorithms on a BSPC. The PRAM model was introduced in [64] as a way to design parallel algorithms and perform quantitative analyses on their performance, in a similar manner to Von Neumann's RAM. The PRAM model relied on a set of simplifying assumptions, namely

- any processor can access any location of the shared memory uniformly (i.e. with the same cost of access)
- there is no resource contention between processors.

A PRAM is categorized according to the strategy it uses to resolve read/write conflicts in memory:

- if a memory location can be accessed only by a processor at a time, regardless if it is a read or a write, then the PRAM is said to be *EREW* (Exclusive Read, Exclusive Write);
- if a single processor can write in a memory location at a given time, but multiple processors can read from it simultaneously, then the PRAM is said to be *CREW* (Concurrent Read, Exclusive Write);
- if multiple processors can read and write in a memory location at a time, then the PRAM is said to be *CRCW* (Concurrent Read, Concurrent Write).[1]

In [59] it is shown that PRAM algorithms can be efficiently simulated on a BSPC[2]; the single shared memory of a PRAM is implemented in a BSPC by mapping its memory positions to local memories of BSP nodes according to a hashing function. The simulation is *optimal* provided that the PRAM has more processors than the BSPC.

Definition 2.11. A model M can *optimally simulate* a model N if there exist a transformation that maps any problem with cost $T(n)$ on N to a problem with cost $O(T(n))$ on M.

Definition 2.12. Let p be the number of nodes of a BSPC. A PRAM algorithm is said to have *slackness* σ if at least σp PRAM processors perform a memory operation (read or write) at every step.

A PRAM algorithm with slackness σ has *at least* σp processors that communicate at each step. A simulation of this PRAM algorithm on a BSPC with p nodes means that at least σ processors are mapped to each BSP node. In order to achieve an optimal simulation, some constraints must be put on σ, as showed by the following theorem.

Theorem 2.1. *Suppose to have a BSPC with p nodes, with g constant (not depending on p or the problem size, i.e. $g = O(1)$) and $l = O(\sigma)$. This BSPC can optimally simulate*

- *any EREW PRAM algorithm with slackness* $\sigma \geq \log p$
- *any CRCW PRAM algorithm with slackness* $\sigma \geq p^{\epsilon}$, *with* $\epsilon > 0$.

The interested reader can find the proof of this theorem in [59].

In [55] Tiskin proposed a BSP model for shared memory systems, which replaced the communication network with a shared memory component; this allowed for the exploiting of data locality, something that was not possible with the original BSP model. Tiskin called his model *BSPRAM*; the architecture for a BSPRAM is shown in Figure 2.3.

[1] The Exclusive Read, Concurrent Write (ERCW) strategy is never considered.

[2] The paper shows a PRAM simulation on a XPRAM, which is a simple BSPC.

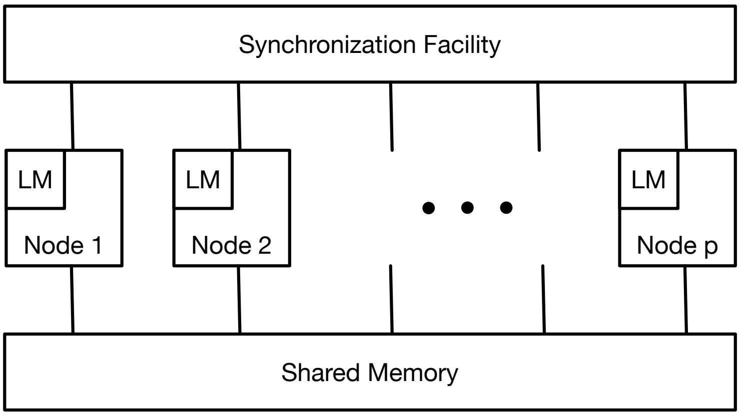


Figure 2.3: The BSPRAM architecture.

As with a standard BSPC, a BSPRAM has p processors (nodes) that perform operations on data in their local memory, a synchronization facility and a communication component, which in this case is a shared memory unit. By adapting slightly the definition of h-relation as the maximum data units *read or written into the shared memory*, the same basic cost model of equation 2.9 can be used. The BSPRAM supersteps differ a little from standard BSP supersteps: in the latter model, the input phase was mostly a "passive" one, since the communication component wrote data directly into the nodes' local memories; communication could be essentially considered concluded when the output phase ended. In the BSPRAM, instead, nodes must actively read from the shared memory in the input phase, so the three phases of a superstep are more distinct (Figure 2.4).

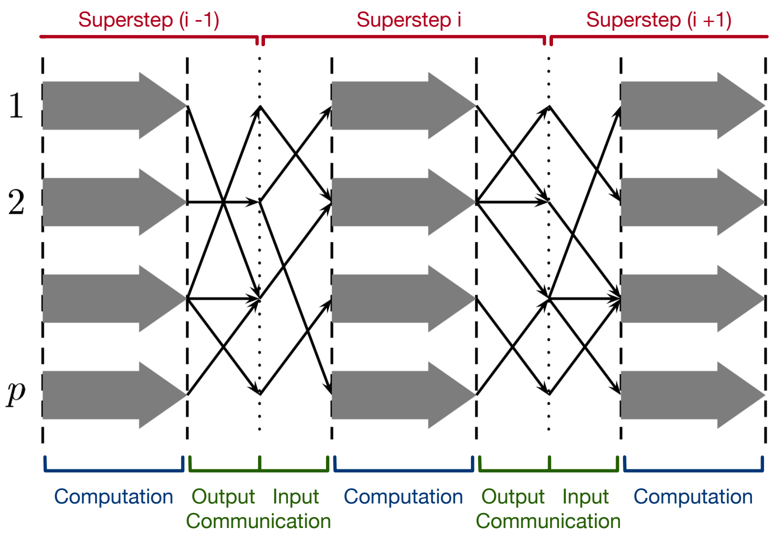


Figure 2.4: The superstep flow of a BSPRAM computation.

As Valiant did for PRAM algorithms, Tiskin proved that BSPRAM programs could be optimally simulated onto a "classical" BSPC. First, he defined the *slackness* parameter for BSPCs.

Definition 2.13. A BSP or BSPRAM algorithm is said to have *slackness* σ if for any superstep i it holds

$$gh_i \geq \sigma \tag{2.13}$$

i.e. the communication cost for any superstep is at least σ.

The following theorem directly mirrors Theorem 2.1.

Theorem 2.2. *An optimal simulation on a BSPC with parameters p, g, l can be achieved for*

- *any algorithm designed for a BSPRAM with EREW strategy and parameters p, g, l, that has slackness $\sigma \geq \log p$*
- *any algorithm designed for a BSPRAM with CRCW strategy and parameters p, g, l, that has slackness $\sigma \geq p^{\epsilon}$, with $\epsilon > 0$.*

Another important property of a BSPRAM is that it can optimally simulate generic BSP algorithms that have at least a certain level of slackness.

Theorem 2.3. *An optimal simulation on an EREW BSPRAM with parameters p, g, l can be achieved for any algorithm designed for a BSPC with parameters p, g, l that has slackness $\sigma \geq p$.*

Proof for Theorem 2.2 follows from the one for Theorem 2.1, while the interested reader can find proof for Theorem 2.3 in [55].

Tiskin's results are interesting, since they show that classical BSP algorithms can be efficiently run on shared-memory systems if the amount of data exchanged at every superstep (the *h*-relation) is at least equal to the number of BSP nodes, which is an easily achievable condition in most cases.

Object-oriented BSP and further developments

Most BSP models presented so far do not place any condition on how the memory is organized, nor on the structure and contents of messages exchanged by nodes. (An example of BSP model that explicitly considers memory hierarchies can be found at [22].) Most BSP implementations — especially the ones based on BSPlib [14, 12], but also including other independent implementations [42, 65] — require that the programmer directly manages the memory, allocating and freeing blocks of data "by hand" [27]. Nowadays, the most popular programming languages tend instead to reduce the burden on the programmer by automatically taking care of memory management. We think that our BSP implementation should adapt this philosophy by providing the user with a simple memory management scheme. We choose to embrace the *object-oriented* paradigm [51] and place *objects* as the atomic unit of both BSP messages (i.e, each message carries a single object) and memory management.

The idea of an object-oriented extension to the BSP model is not a new one. In [31] Lecomber proposed a distributed memory object-oriented BSPC and BSP++, its C++ implementation. Some of the ideas proposed in this work derive from Lecomber's solution, such an emphasis on easy-to-use mechanisms and primitives, while others – for example, tightly coupling an object with the set of processes (nodes) allowed to use it, as a mean to express whether the object is in a node's local memory – only make sense in a distributed environment.

There exist, of course, many implementations of the BSP model for modern languages, some of which support the object-oriented paradigm [25, 26, 33]. The library which bears the closest similarity to our work is, however, *MulticoreBSP for Java* by Yzelman and Bisseling [39, 66]. This library targets shared memory, multicore machines and place *objects* as the atomic unit of data; it provides the user with few, clear-cut memory operations (mainly variants of *put* and *get*) and an interface similar to the language's Collections framework. The definition of a BSP program in MulticoreBSP for Java also follows the object-oriented paradigm: not only the whole computation is encapsulated into a class that inherits from `BSP_PROGRAM`, but also variables are accessed and manipulated via methods of the object that represents them. We will emphasize our solution's similarity to MulticoreBSP for Java in more detail in Chapter 4; here we highlight the fact that most programs written using the MulticoreBSP for Java library are very easily ported to this work's C++ library, once taken care of the differences between the two languages.

Over the course of the years, new refinements of the BSP model have been proposed. In [60], BSP's original author Valiant proposed an extension of its model to account for multicore machines and memory hierarchies called *Multi-BSP*.

A Multi-BSPC is organized in a tree structure where the leaves (level 0 nodes) are processors with no local memory; every node at tree level i contains a portion of level $i - 1$ nodes and acts as a sub-BSPC with some memory space m_i and internal synchronization, where internal level $i - 1$ nodes act as the processing elements of the sub-BSPC. For example

- a multicore processor is a level 1 node with cache memory m_1 where each core is a level 0 node;
- a multiprocessor machine that uses the above processor is a level 2 node with shared random access memory m_2;
- a LAN with a small number of such multiprocessor machines is a level 3 node, and so on.

One key element of the model is the ability to have *nested BSP runs*: Multi-BSP programs can have subroutines which are themselves Multi-BSP programs, and if the tree structure has enough depth the execution of those subroutines can be performed as a BSP computation over lower-level nodes. The *MulticoreBSP for C* library [67], by the same authors of MulticoreBSP for Java, is an implementation of the Multi-BSP model for shared-memory machines using the C language. Although a C++ interface is provided, the library does not focus on the same object-oriented principles as its Java counterpart and the two projects are mostly independent [38].

Lastly, we mention some of the most relevant distributed frameworks that are inspired or directly based on BSP. This model gained popularity in the recent years due to its inclusion in a highly influential proposal for a large-scale graph processing system, Google Pregel [36], in which graph computation follows the bulk-synchronous pattern and is organized in supersteps. Pregel is a

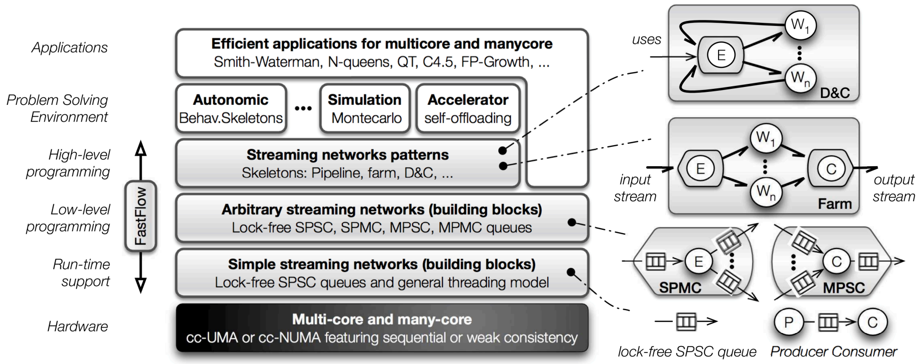


Figure 2.5: The FastFlow framework layered design (from [2]).

proprietary Google technology, but an open-source counterpart called Giraph has been developed at Apache [6, 45, 17]. A more general example of distributed BSP framework, also by Apache, is the Hadoop-based Hama [7, 47].

2.2 FastFlow

FastFlow is a parallel programming framework for shared-memory multicore machines, developed by the Parallel Processing research groups at University of Pisa and University of Torino [3]. It aims to be an easy-to-use yet highly efficient platform for the development of parallel applications targeting shared memory multi/many-cores. It has been designed according to four fundamental principles to achieve those goals:

- a **layered design** that allows progressive abstraction from lower-level hardware concepts up to high-level programming constructs (Figure 2.5). The core of the framework is the *run-time support* layer, built on top of a threading library like POSIX. The run-time support layer features an extremely efficient implementation of one-to-one, Single-Producer Single-Consumer (SPSC) FIFO queues [57]. These queues are then used as building blocks for the above *low-level programming* layer, which provides support for other types of queues (one-to-many, many-to-one, many-to-many) used in creating lock-free and wait-free data-flow graphs. Lastly, the *high-level programming* level provides programmers with an interface for building parallel programs based on the *parallel patterns* concept. Users are not required to build their applications only on top of the high-level programming layer, but can instead use constructs from the full stack of layers;

- a **focus on base mechanism efficiency**; the SPSC queue structure that forms the base mechanism of FastFlow is wait-free and lock-free, with different optimizations for different underlying architectures. Those queues are used as synchronization mechanisms for pointers, in a producer/consumer pattern, and can be used to build networks of entities (threads) that communicate according to the data-flow pattern;

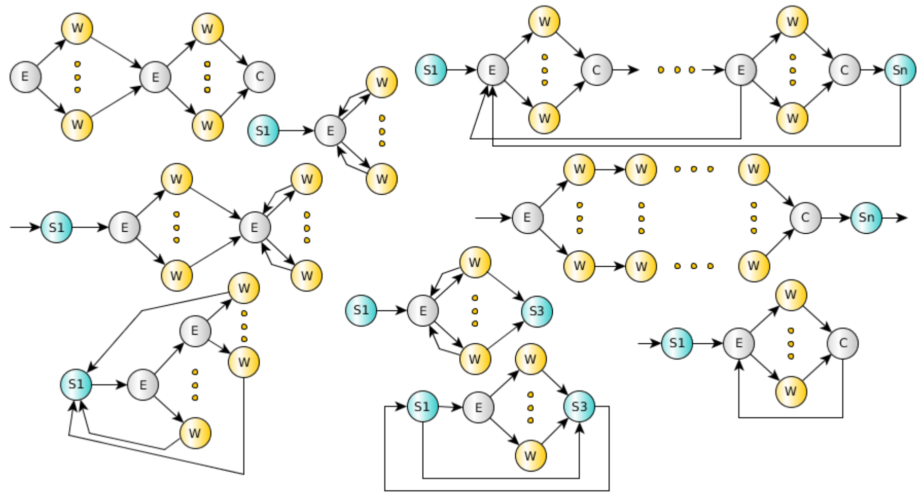


Figure 2.6: An example of arbitrary compositions of FastFlow computational graphs (from [21]).

- the adoption of **stream parallelism** as the main supported form of parallelism. This reduces the complexity of the library and is not restrictive, as other forms of parallelism can be implemented via stream parallel patterns. A stream program is a graph of independent stages that communicate via data channels; the streaming computation is a sequence of transformations on data streams done by nodes in the graph. The data streams can either be generated and consumed entirely inside the streaming application (*endo-streams*) or entirely outside (*exo-streams*);

- the **algorithmic skeletons** approach to parallel programming. The skeletons encompass common parallel programming concepts (e.g. Divide and Conquer, map-reduce, ...) and make them available to the user as high-level constructs with proper semantics.

FastFlow is designed to be portable over different architectures and to support different kinds of accelerators like GPUs and FPGAs. The authors have also started to work on a distributed-memory implementation of the library [4].

FastFlow is provided as a C++11 header library, and its mechanisms are available as template classes. The main component of the framework is the FastFlow *node*, an object of the `ff_node` class; in its simplest form, a `ff_node` contains the portion of sequential computation to be executed by a single runnable entity (thread). The parallel computation can be expressed as a graph of `ff_nodes` and, in true compositional fashion, can be considered a node itself, thus becoming a portion of a broader computation. According to this principle, the library provides algorithmic skeletons such as *pipeline* (`ff_pipe`) and *farm* (`ff_farm`) that are also `ff_node` objects. This supports the construction of graphs of any complexity to be built (Figure 2.6). In Chapter 4 we will see how to implement a BSPC using FastFlow components.

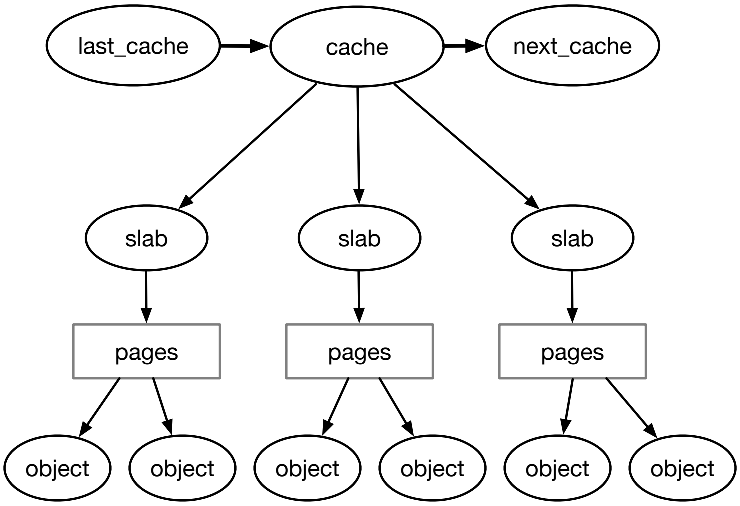


Figure 2.7: The slab allocation memory organization.

2.2.1 Slab Allocator

The FastFlow framework contains a custom allocator which may achieve higher performance when the allocation structure is composed of small memory areas [56]; this allocator is based on the *slab allocator* introduced by Bonwick in [13] and that will be briefly discussed here.

When dealing with frequent allocations and deallocations of small data objects, the CPU time for initialization and destruction of those objects may outweight the proper allocation and deallocation cost, thus degrading the overall performance. A solution to this problem is *object caching*: when an object is released, the system keeps it in memory instead of releasing it; when another object with the same type is needed, the previous one is re-used and the cost of initialization is spared.

A memory managed with slab allocation is organized into multiple *caches* with no shared state and thus able to be locked independently. Each cache is composed by several *slabs* (Figure 2.7); a slab consists in one or more portions of contiguous memory divided into equal-sized chunks, with a reference count (of already-requested chunks) and a free list. When a chunk is requested, it is removed from the free list and the reference count is updated; the opposite happens when releasing it. A slab is the minimum memory size that a cache can grow or shrink; when a cache is full and a new object is requested, an entire slab is allocated and the object is built from a chunk of this new slab, and conversely,when the last chunk of a slab is released, the whole slab gets deallocated and removed from the cache.

The FastFlow implementation of the slab allocator is optimized for the following scenarios:

- a single thread allocates memory, and a single thread (not necessarily the same one) deallocates it;
- a single thread allocates memory, and multiple threads deallocate it.

FastFlow provides two interfaces for this allocator: the low-level one, `ff_allocator`, exclusively supports the allocation/deallocation patterns above, while the "high-level" one, `FFAllocator`, can be used by any thread regardless of the memory operation to be performed. A C++11 "wrapper" for the `FFAllocator` class is provided in the BSP library (see Section 4.9).

Chapter 3

Architectural design

After having introduced the necessary background topics in the previous Chapter, here we provide an overview on how the BSP library for FastFlow is logically organized and how the various tasks are assigned to different entities. One important thing to note is that the actual implementation of the library will not necessarily map these logical entities to "physical" ones, as discussed later in Chapter 4.

3.1 Overview

In the previous Chapter we discussed the three main components to the BSPC, the abstract machine that will execute BSP programs in an analogous way to how the abstract Random Access Machine executes sequential programs. These components are:

- the *nodes* that will provide computing capabilities and local memory;
- the *communication component* that will deliver point-to-point messages between any two nodes;
- the *synchronization facility* that will periodically check if all nodes exhausted their computation and communication tasks for the current superstep, allowing the global computation to advance to the next one.

These components form the basis of the structure of many BSP implementations, and the BSP library for FastFlow makes no exception. Since the library is designed for shared-memory systems, another logical component is needed:

- the *shared memory*, which will host the data needed by multiple BSP nodes.

Note that this BSPC is slightly different from the BSPRAM: in a BSPRAM, the shared memory effectively replaces the communication component, while in the BSPC presented here both entities coexist. Due to its specialized usage, the communication component for this BSPC is also called *memory manager* (MM).

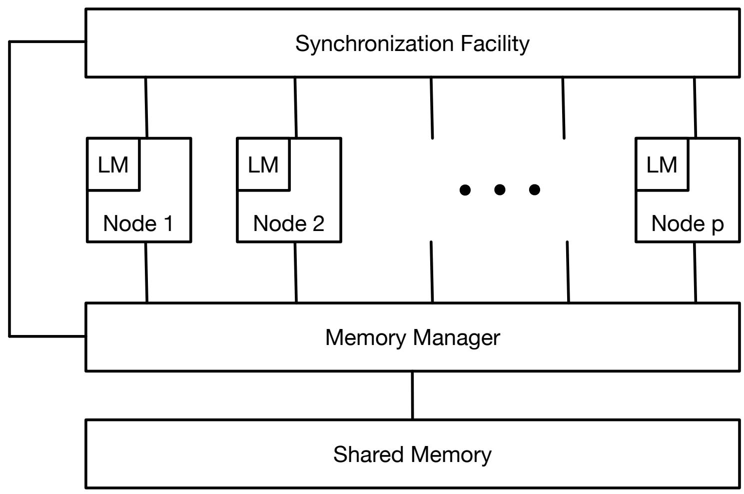


Figure 3.1: The BSPC architecture presented in this work.

3.2 Nodes

BSP nodes are the entities responsible for requesting memory operations and performing computation over data that is already available in local memory. A BSP node can be thought of being composed by four sub-components:

- a *processing unit* (PU) that performs arithmetic and logical operations over local data;
- a *local memory unit* (LMU) that holds data available to be used exclusively by the PU of the same node. This memory cannot be accessed directly from other BSP nodes, but can be written to by the memory manager (see Sections 3.3, 3.5);
- a bidirectional *communication channel* (CC) with the memory manager. The node can perform read/write *requests* for shared memory locations over this channel, and conversely the memory manager will write data on the node's local memory through it;
- a bidirectional *synchronization channel* (SC) with the synchronization facility. The node will signal the end of the local computation phase of the current superstep through this channel, while the facility will signal the start of the next one.

Note that in this BSPC the shared memory component is not accessible directly from the nodes. This avoids memory contention in case of simultaneous accesses, which are instead mediated by the memory manager.

3.3 Memory manager

The memory manager (MM), as its name implies, is the entity that provides BSP nodes with access to the shared memory. In the BSPC we're considering, it

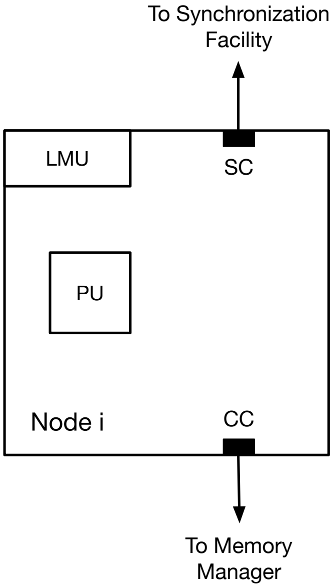


Figure 3.2: The sub-components of a BSP node.

is also the logically centralized entity that effectively realizes the interconnection network between nodes via communication channels (Section 3.2). For this reason, it corresponds to the original BSP's definition of *communication component*.

The MM is connected to *all* BSP nodes via communication channels. On those channels, during the computation phase of the superstep, the MM receives read/write requests from nodes. Since these memory operations are required to be asynchronous in the BSP model (i.e., they immediately appear as completed but must actually take place at the end of a superstep), these requests are *stored* inside the MM. When the computation phase ends, the MM executes each request's operation on the shared memory. The memory contention resolution strategy is not fixed; for our purposes, the MM uses the *CRCW* strategy, but it's conceptually simple to enact the *EREW* or *CREW* strategies. After performing a memory operation, the MM sends the result of that operation to the relevant node (see Section 3.5).

Lastly, the memory manager has a bidirectional channel that connects it to the synchronization facility. The facility will notify the end of the local computation phase of all nodes to the MM, while the latter will notify to the former the end of all memory operations and – thus – of the whole superstep.

3.4 Synchronization facility

The behavior of the synchronization facility is heavily implementation-dependent, so its architectural specification is loosely defined.

The synchronization facility is the entity that enforces the computation structure of the BSP model, i.e. the local computation and communication phases of each superstep. It communicates with the BSP nodes via the *synchronization*

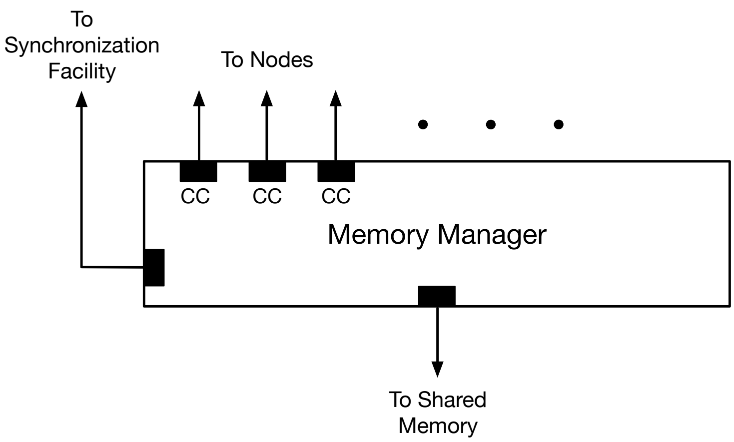


Figure 3.3: The Memory Manager component.

channels (SCs, see Section 3.2), and with the memory manager via a single bidirectional channel.

The synchronization facility periodically checks (via the SCs) if every node has finished its local computation for the current superstep. As soon as this condition is true, it asks the memory manager to perform the end-of-superstep memory operations and waits for it to signal the end of those operations. When the synchronization facility receives this signal, it broadcasts the superstep advancement message to all BSP nodes.

3.5 Shared memory

The defining characteristic of the BSPC that we are implementing is the shared memory component. Unlike its role in the BSPRAM model, where it was effectively the entity responsible for communication between nodes, here the shared memory takes the more passive role of "merely" being the memory space where the data to be used by multiple nodes is stored.

As previously discussed (Section 3.3), nodes can only access the shared memory by performing asynchronous requests via the MM. This means that data to be used in local computation inside a node must be copied in that node's local memory, in order to avoid having to wait for the shared memory operation to conclude. The MM must perform this copy as efficiently as possible and maintain coherence between both types of memory (local node and shared). Thus, an adequate organization of the data in both the shared and local memories is needed.

First of all, we remind that the whole BSP computation is in fact a combination of smaller local computations which – as the name implies – are performed over *local data*. The result of a local computation is also stored in the local memory of the node that performs it. In order for a local computation to use data that belongs to other nodes (i.e. *global* data), that data must have been requested in advance and already copied inside the node; conversely, the result of a local computation can be sent to other nodes only after it has been written

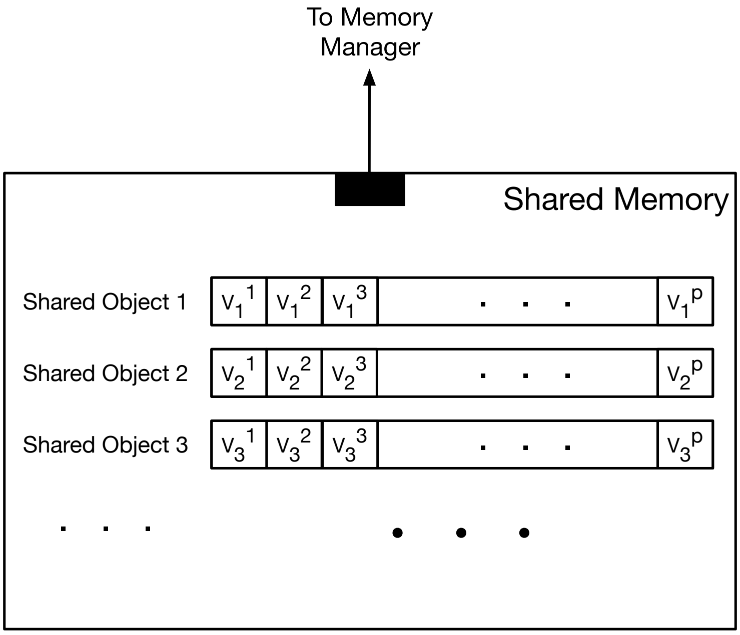


Figure 3.4: The Shared Memory entity.

in local memory. There is, therefore, a certain level of *data redundancy* that must be present in order for the BSP computation to take effect; this applies to any BSPC, not only to the one subject of this work.

The main ideas for data management in the architecture of our BSPC are the following:

1. each global data element is replicated on the shared memory p times (where p is the number of nodes). Basically, each node has its own private copy of the data element on the shared memory;

2. a write request $write(element, payload, recipient)$ is fulfilled by the MM by writing the request payload in both the recipient's local memory and private copy of the element in the shared memory. The MM can perform both writes in parallel (one on the CC with the recipient node, one on the shared memory channel);

3. a read request $read(element, source, destination)$ is fulfilled by the MM in the same way as a write request, i.e. the payload is the source's private copy of the element in the shared memory, which is copied into the destination's private copy and local memory in parallel.

The first idea trades off some shared memory space that may go unused (a private copy for every node is created for each data element, even if that element is not used at all in some nodes) for an increase in performance for read operations, since the element to be read is already present in the (fast) shared memory and there is no need to copy it from the source's local memory. The second and third ideas guarantee coherence between the data in the shared memory and their copies in the nodes' local memories.

3.6 Summary

In this Chapter we introduced an architectural design for an object-oriented BSP model on shared memory machines. We defined *nodes* as the computing entities, with fast local memory and a series of channels for communication and synchronization. We described a shared memory organization that allows for efficient retrieval of information and the network component in charge of communication and memory management. Finally, we detailed the tasks of a synchronization facility that regulates the BSP supersteps.

In the next Chapter we will describe a C++ implementation of a BSPC that follows this design. Each of the components described above will be represented by (one or more) C++ objects.

Chapter 4

Implementation

This chapter describes the implementation design and choices for the BSP library for FastFlow. The architecture of the BSPC described in Chapter 3 is realized over common shared-memory, multicore machines using the FastFlow framework discussed in Section 2.2. The proposed implementation fully embraces the object-oriented paradigm by having the *object* be the atomic data element of the BSP computation. The programmer is therefore relieved of having to manage memory space manually.

4.1 Library structure

The library is provided as a header-only library, for consistency with the FastFlow framework. Programmers who wish to use it can simply include the `bsp_program.hpp` file in their code.

Here is a list of the library files, together with a short description of their contents.

- `bsp_program.hpp` (Section 4.3)

  The FastFlow core of the BSP library. Defines the underlying FastFlow structure, i.e. a farm with custom emitter and collector, and provides the user with means to build the BSP computation by providing `bsp_node` objects and (optionally) functions to be called before and after the computation.

- `bsp_node.hpp` (Section 4.4)

  Specialization of `ff_node` to work as the computing entities of a BSPC. Users must write their own classes that inherit from `bsp_node` and specialize the `parallel_function` method.

- `bsp_internals.hpp` (Section 4.5)

  Forward declarations and definitions for the communication component, `bsp_container` and its specializations (`bsp_variable` and `bsp_array`).

- `bsp_communicator.hpp` (Section 4.6)

  Implementation of the communication component, including the requests mechanism and the methods responsible for the creation of new shared data elements.

- `bsp_variable.hpp` (Section 4.7)

  Implementation of the data structure that represents a shared element.

- `bsp_array.hpp` (Section 4.7)

  Specialization of the `bsp_variable` structure to handle multiple elements of the same type.

- `bsp_barrier.hpp` (Section 4.8)

  Implementation of a simple, reusable barrier that partially fulfills the role of the synchronization facility in a BSPC.

- `stl_allocator.hpp` (Section 4.9)

  C++11 wrapper for the `FFAllocator` class (FastFlow's version of the Slab Allocator, see Section 2.2.1).

4.2 Architecture

The BSPC architecture discussed in Chapter 3 forms the basis for this implementation, which targets single shared memory multi-core machines and relies on the FastFlow framework. It is easy to see that the simplest way to realize the BSPC is to map every entity on a FastFlow node, except for the shared memory entity which is mapped on the program's virtual memory. This mapping is unfortunately inefficient: the memory manager node remains idle while "processing" (BSP) nodes perform their local computation, and vice-versa all BSP nodes must wait that the MM performs its end-of-superstep routine without doing nothing.

A more reasonable mapping, which is used in this work, *distributes* the MM and synchronization facility tasks to FastFlow nodes, effectively mapping the two entities onto the latter.

4.3 `bsp_program.hpp`

The BSP implementation presented in this work uses FastFlow as the target parallel programming framework, namely it is implemented *on top* of it. The `bsp_program` class encapsulates the whole BSP computation. It is a specialization of the `ff_node` class and, as such, it can be used as the component of a broader pattern. It should be noted, however, that in the current state of the library the support for FastFlow composition is still basic, althoug fully functional: the BSP nodes can access the object received in the `bsp_program` FastFlow node as input, and can forward objects to the next stage. The actual BSP computation can be performed either via the `.start()` method, which is completely unrelated to the broader FastFlow computation, or by sending a token to the `bsp_program` node.

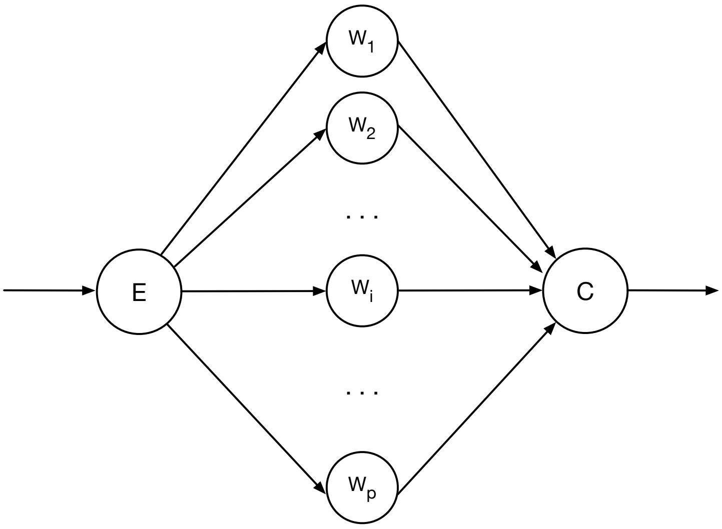


Figure 4.1: The FastFlow graph of the implementation.

The `bsp_program` is internally organized as a `ff_Farm` with custom Emitter and Collector nodes, organized in a `ff_Pipe`. The Worker nodes of the farm properly represent the BSPC, i.e. implement all non-memory entities of the BSPC discussed in Chapter 3. The specialized Worker nodes, objects of the `bsp_node` class, will be discussed in Section 4.4.

```
1  bsp_program(std::vector<std::unique_ptr<bsp_node>>&& _processors,
2                        std::function<void(void)> _pre = nullptr,
3                        std::function<void(void)> _post = nullptr):
4      nprocs{static_cast<int>(_processors.size())},
5      comm{nprocs},
6      barr{nprocs},
7      E{std::move(_pre), nprocs},
8      processors{std::move(_processors)},
9      C{std::move(_post), nprocs} {
10     for (size_t i{0}; i < nprocs; ++i) {
11         processors[i]->nprocs = nprocs;
12         processors[i]->id = i;
13         processors[i]->barrier = &barr;
14         processors[i]->comm = &comm;
15         C.master = this;
16     }
17 };
```

Listing 4.1: The `bsp_program` constructor.

The user is responsible for the creation of the worker nodes, which must then be provided to the `bsp_program` object by passing an `std::vector` of `std::unique_ptr`s to the nodes. The constructor for the `bsp_program` class will inject the necessary information into the `bsp_node` objects: in Listing 4.1, lines

13–14, it is shown how the logical Communication Channels and Synchronization Channels (see Section 3.2) are realized, i.e. as pointers to respective objects. Optionally, the user can also provide two functions `pre` and `post` that will be executed respectively before and after the BSP computation. These functions cannot take arguments and cannot return values. Internally, they are executed respectively by the `ff_Farm`'s Emitter and Collector nodes (Listing 4.2, line 6 and Listing 4.3, line 7).

```
1  struct emitter: ff::ff_node {
2
3      [...]
4
5      void* svc(void* in) override {
6          if (preprocessing != nullptr) preprocessing();
7          for (int i = 0; i < count; i++) {
8              // ENDCOMP is a special value to stop the
9              // FastFlow execution after the BSP
10             // computation ended
11             ff_send_out(ENDCOMP);
12         }
13         return EOS;
14     }
15 };
```

Listing 4.2: The emitter's `svc` method.

```
1  struct collector: ff::ff_node {
2
3      [...]
4
5      void* svc(void* in) override {
6          f (in == ENDCOMP) {
7              if (++count == threshold) {
8                  if (postprocessing != nullptr) postprocessing();
9              }
10             return GO_ON;
11         } else { // Request to forward onto later stages
12             master->forward(in);
13         }
14         return in;
15     }
16 };
```

Listing 4.3: The collector's `svc` method.

Listing 4.4 shows the method that actually builds and runs the whole BSP computation. First, the FastFlow graph is built by moving the `bsp_nodes` into the Workers entities of the `ff_Farm` (lines 2–6, 11). Then, the eventual FastFlow token is forwarded to the communicator so that it can be available to all BSP

```
1  void start(void* in = nullptr) {
2      std::vector<std::unique_ptr<ff::ff_node>> workers;
3      for (size_t i{0}; i < nprocs; ++i) {
4          auto d = static_cast<ff_node*>(processors[i].release());
5          workers.emplace_back(std::unique_ptr<ff_node>(d));
6      }
7  
8      // Let all BSP nodes see the FastFlow input
9      comm.set_fastflow_input(in);
10 
11     ff::ff_Farm<> farm(std::move(workers), E,C);
12 
13     if (farm.run_and_wait_end() < 0)
14         std::cout << "error in running farm" << std::endl;
15 
16     comm.end();
17 }
```

Listing 4.4: The `start` method of the `bsp_program class`.

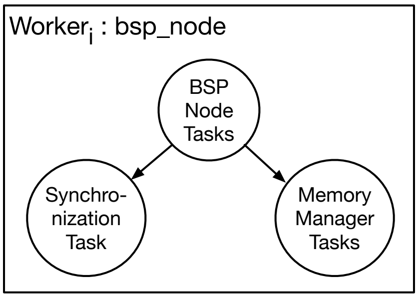


Figure 4.2: Logical tasks performed by a runnable `bsp_node` object.

nodes (line 9). The `ff_Farm` is then started (line 13) and at the end (line 16) the cleanup function of the `bsp_communicator` class is called, in order to properly deallocate all relevant data structures (see Section 4.6).

4.4 bsp_node.hpp

The central unit of a BSP computation is the BSP node, an entity that is capable of processing data stored inside its own local memory and that can send and receive data to and from other nodes. The BSP node abstraction is realized into the `bsp_node` class, which is itself a specialization of the FastFlow `ff_node` and as such is the entity that is mapped onto the actual machine's processing elements, in this case POSIX threads.

As it will be discussed later (Sections 4.6 and 4.8), the communicator and synchronization entities are not implemented into runnable, standalone entities. Their tasks will, instead, be executed by the processing elements that run the `bsp_node` objects. In particular, communication requests are done by calling the *put* and *get* methods of the bsp_variable and bsp_array classes (see Section 4.7). The end-of-superstep operations, be it signaling the end of local computation or processing the memory requests, are executed when the user calls the bsp_sync method (see Listing 4.5).

```
void bsp_sync() {
    barrier->wait();
    comm->process_requests(id);
    barrier->wait();
}
```

Listing 4.5: The bsp_sync method.

The bsp_node class is quite simple. It provides protected methods to access information such as the number of BSP nodes or request new BSP variables. The intended usage for this class is for the programmer to create a class that inherits from bsp_node and implements the virtual parallel_function method. An example of inherited class can be found in the program mwe.cpp that has been provided with the library.

4.5 bsp_internals.hpp

This file mainly provides forward declarations and interfaces for other classes, namely bsp_communicator (Section 4.6), bsp_variable.hpp and bsp_array.hpp (Section 4.7). Nevertheless, two important data structures are defined here: communication *requests* and containers that implement the first idea for data management explained in Section 3.5, i.e. for every global data element the shared memory holds a private copy for each node.

4.5.1 Requests

Requests are represented by a record-like struct that holds the relevant fields as shown in Listing 4.7. The request_type enum (Listing 4.6) is used for discerning how the request itself must be processed at the end of the superstep. Note that not all actual request categories are represented — in particular, the "variable get" and "array element get" requests are stored with the same type as their "put" counterparts, but with the source and destination fields reversed.

```
enum request_type {
    var_put,
    arr_put_el,
    arr_put,
    arr_get
};
```

Listing 4.6: Possible request types.

In the "get" request types, the data object is gathered from the relevant memory element by the communicator. In the "put" ones, the data is instead provided by the node that performs the request. In both cases, the data must be *copied* to allow the original element to be modified by its owner without repercussions on the memory operation. The `request` object contains a pointer to the copy, wrapped in a `std::shared_ptr` object to allow for simpler dynamic memory management.

```
struct request {
    request_type t;
    int reference;
    int source;
    int destination;
    int src_offset;
    int dest_offset;
    int length;
    std::shared_ptr<void> element;

    // Constructor...
};
```

Listing 4.7: Fields of a `request` data structure.

4.5.2 Shared memory containers

As outlined in Section 3.5, the ideal memory organization for shared memory elements in our BSPC consists in duplicating each element p times, where p is the number of BSP nodes. This allows for efficient memory operations at the cost of increased space usage[1].

As it will be discussed further in Section 4.7, the BSP library for FastFlow explicitly place *C++ objects* as the unit of data exchanged in communication between nodes. `bsp_variables` and `bsp_arrays` are therefore template classes whose type arguments are essentially without restraints. This allow maximum flexibility from the user point of view, but provides the additional challenge to having to store heterogeneous data in the single data container that implements the shared memory entity. This would not be a problem in a language that supports type erasure and reflection like Java, where a container like `ArrayList<Object>` would have sufficed and type information could be saved in a `Class` object, but unfortunately C++ does not provide such mechanisms. We decided to implement our own type erasure variant by using `void` pointers and closures to obtain respectively storage of heterogeneous data and a way to save relevant type information.

The `inner_var` (Listing 4.8) and `inner_array` (Listing 4.9) are private inner classes of the `bsp_communicator` class. They both contain a vector of p `void*`

[1] Actually, in this implementation we avoided the redundancy by making nodes hold *pointers* to their private data, instead of having to copy entire elements to the nodes' local memories (represented as class fields in the implementation). These pointers are furthermore invisible to the user, so "BSP-unsafe" modifications of the shared object are impossible unless explicitly requested (Section 4.7).

```cpp
struct inner_var {
    // vector of nprocs elements
    std::vector<void*, ff::FF_Allocator<void*>> element;
    // bookkeeping functions needed to work with void*
    // function to replace a variable with another value
    void (*swap)(void* el, void* other);
    // function to safely free memory occupied by an element
    void (*free_el)(void* el);

    // Constructor...
};
```

Listing 4.8: The `inner_var` class.

objects and functions that will enclose some type information. Objects of any type can be stored in an `inner_var` or `inner_array` object and, since the latter classes are not templated, their objects can be stored in a regular container like `std::vector` regardless of what they contain.

```cpp
struct inner_array {
    // vector of nprocs arrays
    std::vector<void*, ff::FF_Allocator<void*>> element;
    // bookkeeping functions needed to work with void*
    // function to replace an element of the array with another value
    void (*put)(void* el, void* toput, int pos);
    // function to replace a portion of the array
    void (*replace)(void* el, int srcof, int dstof,
                    int len, void* toput);
    // function to safely free memory occupied by an element
    void (*free_el)(void* el);

    // Constructor...
};
```

Listing 4.9: The `inner_array` class.

A `inner_var` object that stores an element of type `T` must be provided with a `swap(element, other)` function that copies the value pointed by `other` into the variable pointed by `element`. Both `element` and `other` are passed as `void*` and must be converted to `T*` inside the function, which effectively saves the type information for `T` inside the function. The `put(element, other, position)` and `replace(element, offset1, offset2, length, other)` functions of the `inner_array` class have the same purpose. Both `inner_var` and `inner_array`, moreover, have a `free_el(element)` function to deallocate an element. The task of creating these functions is not left to the user and are instead generated by the `bsp_communicator` class when a new BSP variable or array is requested.

4.6 `bsp_communicator.hpp`

As said in the introduction to this chapter, the implementation for the BSPC discussed in Chapter 3 does not assign every logical entity to a physical one. This is especially true for the `bsp_communicator` class, which not only represents two logical entities — the Memory Manager and the Shared Memory — but is also distributed over the FastFlow nodes instead of being an autonomous centralized node.

4.6.1 Implementation of the Shared Memory entity

The Shared Memory entity is implemented simply as a vector of `inner_vars` or `inner_arrays` that represent shared elements (see Section 4.5.2). To obtain a higher performance when accessing those containers, lookup maps – which associate *hashes* of the variables (or arrays) to their position in the vector – are used. The hashing function used for this purpose is the one commonly known as `djb2` ([53, 62], shown in Listing 4.11), which is a simple yet effective [29] hash that takes a string and an integer and returns another integer. In our case, the string argument is an unique representation of the variable's type obtained with the `typeid` builtin, while the integer argument is obtained from the current superstep number and the total number of variables of type `T` requested by the current node, for reasons that will become clear shortly. The hash result is then used as the key for a lookup map, whose values are positions in the vector of `inner_vars`. C++17 `std::shared_mutexes` regulate concurrent access to the shared storages: those mutexes allow for a single writer-multiple readers pattern, so any number of nodes can access (and modify!) already-existing shared elements, but only a single node at a time can create a new variable or array (lines 10 and 12 of Listing 4.10).

One of the problems that arised when devising the implementation was how to allow users to request access to the *same* shared variable[2]. For example, take this pseudo-code:

```
bsp_variable v = get_variable<double>();
if (node_id == 0)
    v.put(payload=5.0, destination=1);
```

Suppose we execute this code on a BSPC with two nodes. The communicator entity receives a request for a `bsp_variable` of type `double` from both nodes, but does not know if they are asking for access to the same variable or for the creation of two separate variables. There are two ways to solve this problem:

- let the user specify unique IDs when requesting variables (e.g. as strings or integers), and treat variables of the same type with the same ID as the same shared object;
- let the library manage variable requests automatically, providing the user with a clear rule for defining when a fresh shared object is created.

We chose the second option, according to the library's philosophy of relieving the user from the burden of explicitly managing memory aspects. We use the following proposition.

[2]Wherever we refer to variables in this section, the same concept applies for arrays.

```
// number of current superstep
int generation = 1;

// counts the variables requested by each worker in this superstep
std::map<std::string, int>* var_count;
// counts the arrays requested by each worker in this superstep
std::map<std::string, int>* arr_count;

// mutex for multiple readers-single writer access to the variables
mutable std::shared_mutex var_mutex;
// mutex for multiple readers-single writer access to the arrays
mutable std::shared_mutex arr_mutex;

// dictionary for quick retrieving of variables based on their hash
std::map<int, int> variable_dict;
// dictionary for quick retrieving of arrays based on their hash
std::map<int, int> array_dict;

// actual storage of variables
std::vector<inner_var> variables_storage;
// actual storage of arrays
std::vector<inner_array> arrays_storage;
```

Listing 4.10: Fields of the `bsp_communicator` class used to implement the Shared Memory logical entity.

```
static int get_hash(const char* s, int seed) {
    int hash = seed + 5381;
    while (*s) {
        hash = hash * 33 ^ (*s++);
    }
    return hash;
}
```

Listing 4.11: The hashing function for variables and arrays.

Proposition 1. During the *i*-th superstep, for any node the *n*-th request for a variable of type `T` will return a handle to the *n*-th shared element of type `T`.

In practice, this means that when a node requests a variable of type `T` for the first time in a superstep, and it is the first node in chronological order to do so, a new shared element of type `T` is created. When any other node request their first variable of type `T`, they receive a handle to (their private data of) this shared element, and so on. The method to request a variable is shown in Listing 4.12, and the one to request an array is similar. Lines 2–12 show how the hash value used in the lookup map is obtained: the method then checks if a variable with the same hash is present (lines 13–17) and, if not, a new `inner_var` shared element is created. Lines 22–28 show how type information is captured inside the functions passed to the `inner_var` constructor. The newly-created shared element is then pushed into the shared memory (line 30) and the lookup map is updated (line 31). The shared element's private data – relative to the node that requested it – is updated with the provided value (line 37), and a `bsp_variable` containing that pointer is returned to the caller (line 39).

4.6.2 Implementation of the Memory Manager entity

The `bsp_communicator` class also fulfills the role of the Memory Manager, i.e. the entity that collects communication requests during the local computation phase of a superstep and executes them at the end of it. As we said before, there is no autonomous entity that performs the Memory Manager tasks in a centralized way. FastFlow nodes – the same that run BSP nodes – will handle both request management and execution. The Memory Manager implementation shown here (Listing 4.13) has p request queues, one for each BSP node. Concurrent access to these queues is regulated by a series of mutexes, one mutex per queue. The *i*-th request queue will hold all requests that have the *i*-th node as *destination*. In this way, at the end of the superstep, the FastFlow node that is running BSP node i can process all requests directed to it with no concerns over concurrent accesses.

Listing 4.14 shows an example of how requests are inserted and managed. First of all (line 4), exclusive access to the destination node's request queue is obtained by locking the relevant mutex. Then, the request proper is created and enqueued (lines 5–6), and finally the mutex is released. `m_shared_ptr` is a macro to simplify the creation of a `std::shared_ptr` using the custom `stl_allocator` described in Section 4.9.

Listing 4.15 shows instead how requests are processed by the various FastFlow nodes at the end of a superstep. Each node processes the requests which feature it as destination (i.e. the ones who have effect on the node's private data of shared elements). In this way, concurrent access to the same portion of shared memory is avoided and no locking mechanism is needed. Each request is then processed according to their type; lines 5–11 of Listing 4.15 show, for example, how a variable `put` request is managed. The "type information-holding" functions of the `inner_var` and `inner_array` classes are used here: since the data is stored as `void*` pointers, these functions are needed to cast it back to its proper type before it can be modified (Section 4.5).

```
bsp_variable<T> get_variable(int holder, T* initial_val) {
    auto tname = typeid(T).name();
    // no. of variables of type T requested by the current worker
    int get_count = 0;
    try {
        get_count = var_count[holder].at(std::string(tname));
    } catch (const std::out_of_range&) {
        var_count[holder].insert({std::string(tname), 0});
    }
    // hash on type, superstep and number of vars
    int hash = get_hash(typeid(T).name(),
                        (generation * 5000000) + get_count);
    int ref;
    // try to find a variable w/ the same hash
    try {
        std::shared_lock lock(var_mutex);
        ref = variable_dict.at(hash);
    } catch (const std::out_of_range&) {
        std::unique_lock lock(var_mutex);
        auto iter = variable_dict.find(hash);
        if (iter == variable_dict.end()) {
            inner_var var{nprocs, [](void* el, void* other){
                auto tptr = static_cast<T*>(el);
                *tptr = *(static_cast<T*>(other));
            }, [](void* el){
                auto tptr = static_cast<T*>(el);
                delete tptr;
            }};
            ref = variables_storage.size();
            variables_storage.push_back(var);
            variable_dict.insert({hash, ref});
        } else {
            ref = iter->second;
        }
    }
    // initialize with the value requested by the worker
    variables_storage[ref].element.at(holder) = initial_val;
    var_count[holder].at(std::string(tname))++;
    return bsp_variable<T>{ref, holder, this, initial_val};
}
```

Listing 4.12: The `get_variable` method.

```
std::mutex* mutexes;
std::vector<request, ff::FF_Allocator<request>>* requests;

std::atomic_int process_count{0};
```

Listing 4.13: Fields of the `bsp_communicator` class used to implement the Memory Manager logical entity.

```cpp
template <typename T>
void variable_put(int what, int source,
            int destination, const T& element) {
    mutexes[destination].lock();
    requests[destination].emplace_back(request_type::var_put,
        what, source, destination, 0, 0, 0, m_shared_ptr(T, element));
    mutexes[destination].unlock();
}
```

Listing 4.14: The variable put request handler of the `bsp_communicator`.

```cpp
void process_requests(int id) {
    // Request filtering by worker id
    for (const auto& req: requests[id]) {
        switch (req.t) {
            case request_type::var_put: {
                auto ptr = variables_storage.at(req.reference)
                            .element[req.destination];
                variables_storage.at(req.reference)
                        .swap(ptr, req.element.get());
                break;
            }
            case request_type::arr_put_el: { ... }
            case request_type::arr_put: { ... }
            case request_type::arr_get: { ... }
        }
    }
    if (++process_count == nprocs) {
        generation++;
        delete[] arr_count;
        arr_count = new std::map<std::string, int>[nprocs]();
        delete[] var_count;
        var_count = new std::map<std::string, int>[nprocs]();
        delete[] requests;
        requests = new std::vector<request,
                ff::FF_Allocator<request>>[nprocs]();
        process_count = 0;
    }
}
```

Listing 4.15: Request processing.

Lines 17–27 are only executed by the last node that finishes processing its requests. They “clean the state” relative to the current superstep, i.e. reset the variable and array counters to 0 and empty all request queues.

```
void bsp_communicator::end() {

    for(auto& var: variables_storage) {
        auto del = var.free_el;
        for (auto& el: var.element) {
            del(el);
        }
    }

    for(auto& arr: arrays_storage) {
        auto del = arr.free_el;
        for (auto& el: arr.element) {
            del(el);
        }
    }

    delete[] arr_count;
    arr_count = nullptr;
    delete[] var_count;
    var_count = nullptr;
    delete[] requests;
    requests = nullptr;
    delete[] mutexes;
    mutexes = nullptr;
}
```

Listing 4.16: The `bsp_communicator` final cleanup function.

Finally, Listing 4.16 shows the `end()` method of the `bsp_communicator class`. It is a cleanup function that gracefully deallocates all shared objects and data structures used by the Shared Memory and Memory Manager entities. It will be called by the encompassing `bsp_program` object at the very end of the BSP computation, after all nodes have exhausted their local computations (see Section 4.3).

4.7 `bsp_variable.hpp` and `bsp_array.hpp`

As outlined in Section 3.5, even in our shared-memory BSPC each node must have all data needed for computation in its local memory, and this includes shared objects. There is therefore a need for a representation of those shared elements inside the BSP node. The `bsp_variable` and `bsp_array` fulfill this task. A `bsp_variable<T>` is a template class that holds the local copy of a shared object of type `T`, and it also represents the handle of the shared object that the user can employ to perform memory requests on it. Due to limitations in the STL functions and data structures used in the implementation (in particular the `std::allocate_shared` function, see [49]), the type `T` *cannot* be a C-style array type, e.g. `E[]`. This restriction is lifted in the C++20 version of the language. The `bsp_array<T>` class is similar to the previous one, except for the fact that the shared object is a resizable vector of elements of type `T`. Therefore, the class provides methods for vector manipulation like iterators and access to elements. Unlike `bsp_variable`, in the `bsp_array<T>` class `T` can be a C-style array type.

```
1  template <typename T>
2  class bsp_variable: public bsp_container {
3  
4      // The template type must be copy-constructible and copy-assignable
5      static_assert(std::is_copy_constructible<T>::value &&
6          std::is_copy_assignable<T>::value,
7              "Type of bsp_variable must be "
8                  "copy-constructible and copy-assignable");
9  private:
10     ...
11     T* element; // Pointer to the actual element
12     ...
13 public:
14     // Replaces the element on another worker
15     void bsp_put(const T& elem, int destination) {
16         comm->variable_put<T>(reference, holder, destination, elem);
17     }
18 
19     ...
20 
21     // Returns the element in another worker's memory
22     // directly (without waiting for a superstep sync)
23     T bsp_direct_get(int source) {
24         return comm->variable_direct_get<T>(reference, source);
25     }
26 
27     // Allows direct access to the element
28     // in this worker's memory
29     T& direct_access() {
30         return *element;
31     }
32 };
```

Listing 4.17: The `bsp_variable` class.

Listing 4.17 shows a portion of the `bsp_variable<T>` class. First of all, in order to perform BSP memory operation, elements of type `T` must be copy-constructible and copy-assignable (lines 4–7); this is needed, since objects have to be copied inside requests and successively into the shared memory. The local copy of the private data of the shared object is implemented simply as a pointer to the data stored inside the `bsp_communicator` object, to save some copies. The user can call the methods `bsp_put` and `bsp_get` of the class (with different signatures according to the desired operation to be executed) to perform memory requests on the current shared object. The class provides also two *"BSP-unsafe"* functions: `bsp_direct_get` and `BSPunsafe_access`. The first one (lines 23–25) returns *a copy* of a node's private data of the shared element;. This violates the BSP condition that a node cannot access another node's local or private memory, thus the function is deemed "BSP-unsafe". Still, the fact that *a copy* of the desired data is returned means that this function can reasonably be used inside a BSP computation without having to consider unwanted side effects.

The second "BSP-unsafe" method is more problematic, since it gives access to the *actual element* in the shared memory (lines 29–31). This is a big violation of our BSPC's premise that the Memory Manager entity is the only one allowed to operate on shared memory, and in general of the whole BSP concept of only allowing communication effects to take place at the end of the superstep. Both `direct_get` and `BSPunsafe_access` functions are provided for compatibility with similar methods used in the MulticoreBSP for Java library (see [40]).

The `bsp_array<T>` class is very similar to the `bsp_variable<T>` one, so it won't be discussed as much. `T` still needs to be a copy-constructible and copy-assignable type, and the class provides specialized `bsp_put` and `bsp_get` methods for single elements of the vector. Arrays also have another BSP unsafe method, `BSPunsafe_put`, that places an element into the node's private copy without waiting for the end of the superstep.

4.8 `bsp_barrier.hpp`

The synchronization facility of our BSPC (Chapter 3) is the entity that properly manages the flow of supersteps of the whole computation: its tasks are to

1. check wherever all BSP nodes have finished their local computation;
2. wait that the Memory Manager finishes processing requests;
3. begin the next superstep, updating the BSPC's relevant structures (see Listing 4.15) and signaling the nodes to begin local computations again.

The request processing and the update portion of task 3 are implemented in our library in the `bsp_communicator` class (see Section 4.6). This leaves the implementation of the actual check and the signal to start the new superstep. A synchronization mechanism that implements them in a very simple way is the *barrier*. Nodes that finish their local computation will wait at the barrier for their peers to do the same. When all nodes reach the barrier, the latter is lifted and they can begin the local operations of the next superstep. As seen in Listing 4.5, the barrier is called twice — once when the nodes finish their local computation, and once when the Memory Manager finishes processing requests (remember that both operations are effectively executed by the physical entity that runs the underlying FastFlow node).

The barrier structure used in the implementation is shown in Listing 4.18. It's a simple structure that uses a condition variable (line 4) and its wait/notify mechanism (lines 35–38), together with a generation counter to allow for multiple uses [63].

```cpp
class bsp_barrier {
    private:
        std::mutex mutex;
        std::condition_variable cond_var;
        // Number of workers needed to release the barrier
        int threshold = 0;
        // Count of workers currently at the barrier
        int count = 0;
        // Number of times the barrier has been used
        int generation = 0;

    public:
        bsp_barrier() = delete;

        explicit bsp_barrier(int size):
            threshold{size},
            count{size},
            generation{0} {
        }

        // Copy constructor
        bsp_barrier(const bsp_barrier& other):
            threshold{other.threshold},
            count{other.count},
            generation{other.generation}{
        }

        // Barrier function
        void wait() {
            std::unique_lock<std::mutex> lock{mutex};
            auto lastgen = generation;
            if (!(--count)){
                generation++;
                count = threshold;
                cond_var.notify_all();
            } else {
                cond_var.wait(lock,
                    [this, lastgen](){ return lastgen != generation; });
            }
        }
};
```

Listing 4.18: The `bsp_barrier` class.

4.9 stl_allocator.hpp

The FastFlow allocator discussed in Section 2.2.1 performs better than the standard C++ allocator when the program has to do multiple allocations of small memory areas ([56], also discussed in Chapter 5). Oftentimes, the request mechanism of the Memory Manager exhibits this behavior, so it can be helpful to use the FastFlow allocator in place of the standard one. Unfortunately, this allocator does not provide an "STL-conforming" interface [5], so it has to be used manually (e.g. calling the `malloc` and `free` functions of an `FFAllocator` instance). The `stl_allocator.hpp` file contains a very simple interface (Listing 4.19) that allows the use of the FastFlow allocator wherever an STL allocator is required, e.g. in the `std::allocate_shared` function.

```cpp
template <typename T>
class FF_Allocator {
public:
    using value_type = T;
    using propagate_on_container_move_assignment = std::true_type;
    using is_always_equal = std::true_type;

    FF_Allocator() noexcept = default;
    template <class U>
    explicit FF_Allocator(const FF_Allocator<U>& other) noexcept {};

    value_type* allocate(std::size_t n) {
        return static_cast<value_type*>(FFAllocator::instance()
                    ->malloc(n* sizeof(value_type)));
    }

    void deallocate(value_type* ptr, std::size_t) noexcept {
        FFAllocator::instance()->free(ptr);
    }
};
```

Listing 4.19: The FastFlow allocator "STL interface".

Chapter 5

Experimental validation

5.1 Performance metrics

Before we delve into the analysis of the performance of the library presented in this work, it is important to lay the foundations for the metrics used throughout the Chapter. First of all, we formally define the most straightforward parameter of the set: the time needed for a computation to take place from start to finish.

Definition 5.1. The *total time* it takes to compute a single result is called **latency**, usually referred to with the symbol L.

The latency is a first indicator of how well the program behaves. Clearly, the lower L is, the better the program is considered. Nevertheless, if to obtain a slightly lower latency we must massively increase the resources in play, the tradeoff is not always worthwhile. We need a metric to define how "well-utilized" is the machine that performs the computation. Since we are dealing with parallel programs, the number of workers (nodes, cores, processors) used in the computation has an effect on the latency.

Definition 5.2. A parallel computation over p workers is said to have latency $T_{par}(p)$ (or T_p).

As a particular case, $T_{par}(1)$ is the latency of a computation over a single worker. Oftentimes, executing a parallel algorithm on a single worker will still build a set of data structures and mechanisms for parallel synchronization and communication which is not actually needed inside the computation itself, therefore inflating the latency. For this reason, analyses on the performance of a parallel algorithm or library are conducted with regards to the latency *of the best sequential algorithm* that solves the same problem without overheads from parallel-specific mechanisms.

Definition 5.3. A purely-sequential algorithm with no parallel overhead is said to have latency T_{seq}.

We can now introduce relevant metrics for performance evaluation.

Definition 5.4 (Speedup)**.** The ratio between the latency of a sequential computation and the latency of the corresponding parallel computation over p workers is called **speedup**:

$$sp(p) = \frac{T_{seq}}{T_{par}(p)} \tag{5.1}$$

The speedup roughly represents "how much" the latency changes when switching from a purely sequential computation to a parallel one over p workers. The best result one can hope to achieve is to perfectly cut the latency of the sequential computation down by a factor p:

$$T_{par}(p) = \frac{1}{p}T_{seq} \Longrightarrow sp(p) = \frac{pT_{seq}}{T_{seq}} = p$$

This behavior is called *linear speedup* and it is rarely encountered, due to parallel computations needing coordination and communication between them; in fact, the speedup for a single worker

$$sp(1) = \frac{T_{seq}}{T_{par}(1)}$$

is almost never equal to 1 due to the abovementioned need to build all structures needed for the parallel computation. This suggests a new metric that closely encompasses "how better" a parallel computation behaves when increasing the number of workers.

Definition 5.5 (Scalability)**.** The ratio between the latency of a parallel computation over a single worker and the latency of the same parallel computation over p workers is called **scalability**:

$$sc(p) = \frac{T_{par}(1)}{T_{par}(p)} \tag{5.2}$$

This metric is sometimes called *relative speedup*, while the quantity of equation 5.1 is called *absolute speedup* [37].

Finally, we define a measure that represents "how much" of the hardware is used during a parallel computation.

Definition 5.6 (Efficiency)**.** The **efficiency** is the ratio between the speedup of a parallel computation and the number of workers:

$$\epsilon(p) = \frac{sp(p)}{p} = \frac{T_{seq}}{pT_{par}(p)} \tag{5.3}$$

A parallel computation that achieves linear speedup $sp(p) = p$ will have efficiency

$$\epsilon(p) = \frac{p}{p} = 1$$

i.e. it fully utilizes the available hardware resources. Using the above definition of scalability, one can also define a *relative efficiency* as

$$\epsilon_r(p) = \frac{sc(p)}{p}.$$

In some cases, one can attain an efficiency greater than 1, i.e. $sp(p) > p$. This phenomenon is called *superlinear speedup* and may be due to various factors such as

- a better cache exploitation in the parallel program w.r.t. its sequential counterpart;
- a different organization of data objects for parallel programming, improving data locality even when the parallel computation uses a single thread;
- the parallel algorithm is simply more efficient than the sequential one, in the sense that it may skip unnecessary work (e.g. earlier branch pruning in search tree problems).

These situations are rarely encountered, therefore sublinear speedups are more common.

5.2 Machine architecture

Tests were conducted on a machine of the Department of Computer Science, University of Pisa. It features an Intel processor of the Xeon Phi x200 (Knights Landing) family, sporting 64 cores at 1.3 GHz [28]. Each core can have up to four threads, for a total of 256 threads. The machine has 48 GB of RAM and runs the GNU/Linux CentOS 7.2.1511 operating system, kernel version 3.10. The C++ programs were compiled using `gcc` version 7.3.0 (`libstdc++` version 6.0.24), while the Java programs (packaged into JAR files) were executed using OpenJDK 1.8.0_201. The JAR files were instead compiled with the official Oracle Java SDK version 13.0.1 (at language level 8) on a different machine with a 2.3 GHz dual-core Intel Core i5 processor, 8 GB RAM, running macOS 10.14.6. The FastFlow version used is 2.2.0, checked out from the official repository at commit `8b9105d` [20, 19].

5.3 Benchmarks

The BSP model has traditionally been used for solving numerical problems, especially in the field of linear algebra. In [9], Bisseling introduced a series of programs written in C using the BSP model, the *BSP educational package*. It features three main BSP algorithms for solving the following problems:

- *LU* decomposition of a square matrix;
- the Fast Fourier Transform of a vector of complex numbers;
- the sparse matrix-vector multiplication.

The BSPedupack also includes a program for testing the parameters of the BSPC in use and a “toy” program (used as a tutorial of sorts for the BSP model) that performs the inner product of two vectors.

The BSPedupack suite is one of the most widespread examples of BSP programs: most BSP implementations use it for testing performance and correctness. Bisseling's original version targets the BSPlib interface for C [8], but a version also exists for the MulticoreBSP for Java library.

When choosing the algorithms to use for testing our library, the BSPedupack ones seemed the obvious choice. We chose, though, to substitute one of the programs – the sparse matrix-vector multiplication – with a non-numerical algorithm, to show an example of how the BSP model is used for solving problems in other fields. The "replacement" program implements a parallel sorting algorithm.

In general, when possible, the programs written for the C++ library derive directly from the BSPedupack for Java versions. This is to show that is easy to port existing programs to our library, requiring only slight adjustements besides having to change language-specific constructs and data structures. The sorting program has been instead written originally for our library and later ported to Java.

All the programs are self-contained and require no input other than the problem size and the number of desired parallel workers. Each program will (often randomly) populate the data structures on which the computation will be performed.

For the purpose of performance metrics calculations, for each parallel program a corresponding sequential version has been implemented. When possible, the sequential algorithm used is the closest version to the parallel one, using the same data structures.

All the tests have been executed using the same methodology. A suitable problem size is chosen in advance, large enough for the computation time to be much higher than the time needed for maintaining parallel structures. The program is then executed six times with this problem size, starting with parallelism degree 1 and doubling it each time up to a ceiling of $p = 64$. The time spent in the parallel portion of every execution is logged. The corresponding sequential program is also executed with the same problem size, logging the actual computation time (i.e. not counting data generation).

We call the set of these seven executions a *run*. Ten runs for each programs are executed; for each run and each parallism degree we discard the maximum and minimum times and calculate the average of the remaining ones. This average time spent in the computation forms the basis for the calculations of the abovementioned performance metrics. The tests aim to draw comparisons between three kinds of BSP libraries:

- the one presented in this work that uses the FastFlow slab allocator (Section 2.2.1)
- a variant of the same one that uses the standard library (STL) allocator;
- the MulticoreBSP for Java library.

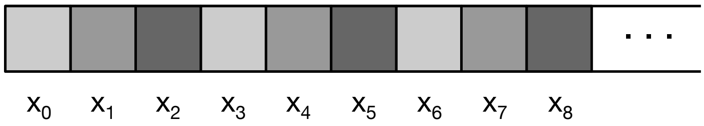


Figure 5.1: An example of cyclic distribution of an array over 3 processors.

5.3.1 Vector inner product

The first program is meant to be a simple introduction to BSP programming. Given two vectors of n integer numbers

$$\vec{x} = [x_0, x_1, \ldots, x_{n-1}] \qquad \vec{y} = [y_0, y_1, \ldots, y_{n-1}]$$

the **inner product** (or *dot product*) of the two vectors is defined as follows:

$$\vec{x} \cdot \vec{y} = \vec{x}\vec{y}^T = \sum_{i=0}^{n-1} x_i y_i \tag{5.4}$$

Given the input size n, the program creates the vector $\vec{x} = [1, 2, \ldots, n]$ (i.e. $x_i = i + 1$) and calculates the inner product of $\vec{x}$ with itself:

$$\vec{x} \cdot \vec{x} = \sum_{i=0}^{n-1} x_i x_i = \sum_{i=1}^{n} i^2.$$

The sequential algorithm simply scans the vector once, accumulating the squares of the x_i elements in a variable. To check for correctness, the known formula

$$\sum_{i=1}^{n} i^2 = \frac{1}{6} n(n+1)(2n+1) \tag{5.5}$$

is used for rapidly calculating the correct value.

The BSP algorithm distributes the input array over p processors according to a *cyclic distribution*: processor k is assigned all the elements x_i such that $i \bmod p = k$. In Figure 5.1 the array cells with the same color are assigned to the same processor. Each processor then computes (sequentially) the inner product of its portion of vector and broadcasts the value to all other processors. Finally, all the processors (reduntantly, but in parallel) sum the received data to obtain the final result.

The performance metrics for this program were evaluated by running it with a problem size of $n = 2 \cdot 10^9$. Figure 5.2 shows a comparison of average latencies (completion time) for the three libraries. (The series shown as "0" on the plot refers to the sequential program execution.) The MulticoreBSP for Java implementation performs better (by less than 10%) than either C++ ones when using a single thread. As the parallelism degree increases, the two C++ implementations prove to be faster, albeit by a narrow margin. As this is the simplest BSP program, with very few communications and a sequential portion that can be easily vectorized by compiler optimizations, these kinds of results are to be expected. Note that, between the two C++ implementations, the one that uses the standard allocator achieves lower latencies at any parallelism degree.

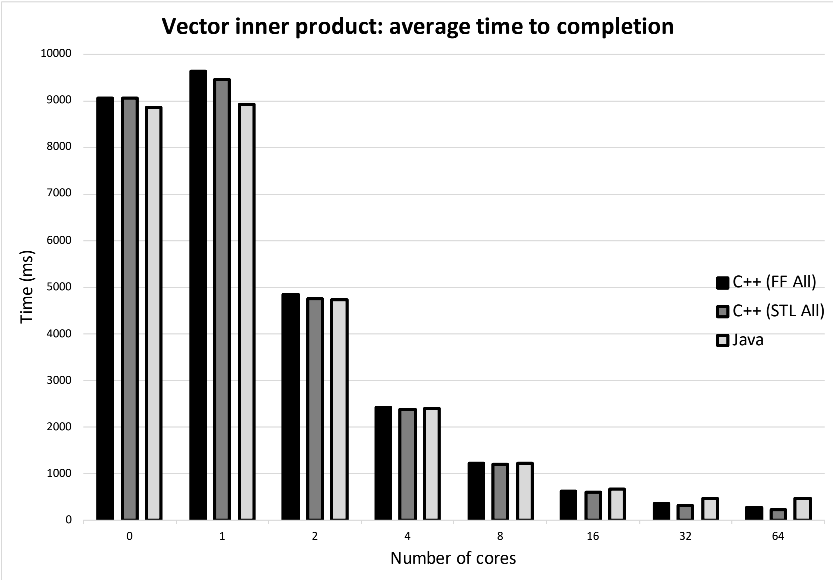


Figure 5.2: Average completion time for the vector inner product benchmark on 2B elements.

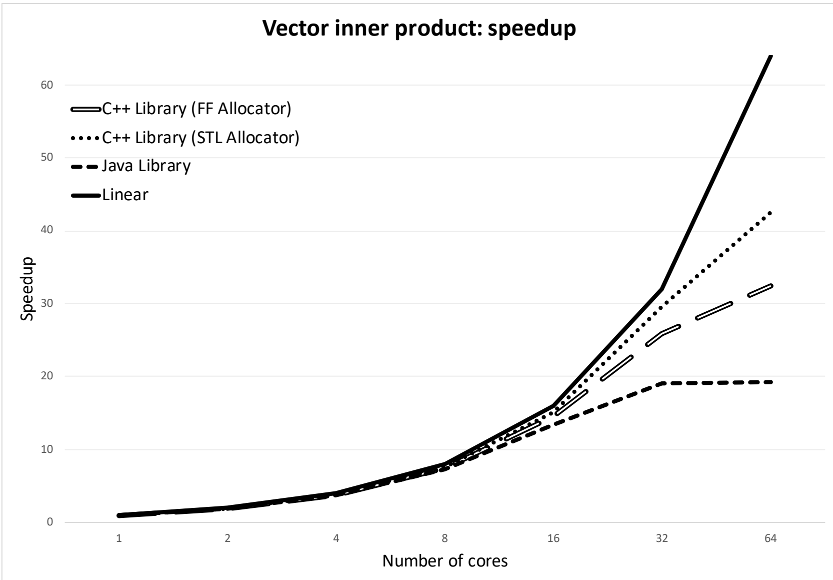


Figure 5.3: Speedup for the vector inner product benchmark.

Figure 5.3 shows the speedups achieved by the three libraries. The Java implementation cannot maintain a near-linear speedup for parallelism degrees higher than 8, and after $p = 16$ the gains are minimal (quadrupling the cores only increases the speedup of a factor ~ 0.6). Both C++ implementations exhibit a decent speedup up to 32 workers, and even after that the speedup increase is still acceptable, especially for the implementation with the STL allocator. The slab allocator version reaches a sort of middle ground between the two others.

The trends shown in Figure 5.4 about scalability are similar to the ones of Figure 5.3. In particular, the Java implementation scalability exhibits almost the same behavior as its speedup, meaning that it holds $T_{seq} \approx T_{par}(1)$ (i.e., the creation and maintenance of structures for supporting parallelism has a negligible effect on the latency). Conversely, both C++ implementations exhibit consistently higher scalability than speedup.

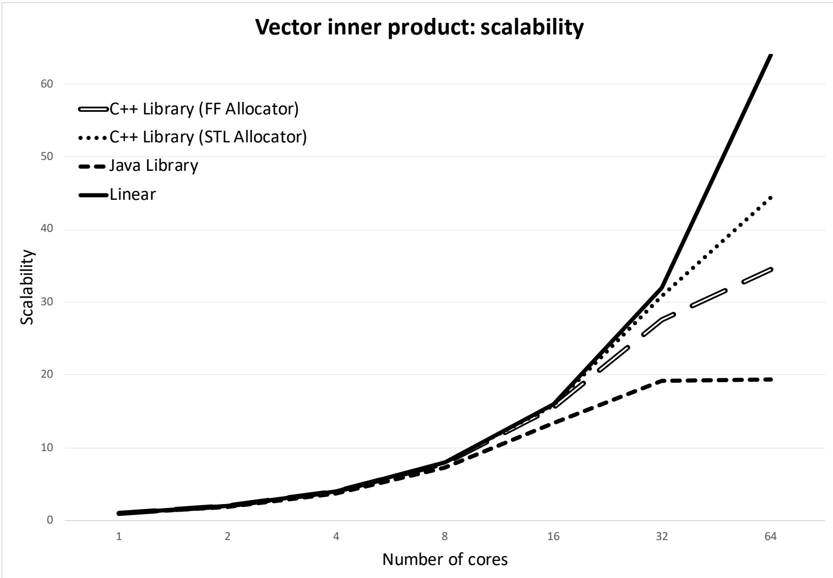


Figure 5.4: Scalability for the vector inner product benchmark.

Lastly, Figure 5.5 sums up the behavior of the three implementations with regards to this program by showing their efficiency parameter. As said before, the Java library offers good efficiency at low parallelism degrees, but it is outclassed by the other implementations after $p \geq 4$ (and starts to decline rapidly for $p \geq 8$). The C++ library on the other hand exhibits a good level of efficiency, between 0.9 and 0.95 for both variants up to $p = 16$. At $p = 32$ the implementation that uses the slab allocator begins to fall behind, while the one with the STL allocator maintains an efficiency of more than 0.9. Finally, at $p = 64$ both implementations show a definite decline in efficiency.

5.3.2 Parallel sorting by regular sampling

Sorting is one of the most important and well-studied operations in computing, as it is used as a subroutine of a large number of programs. Many parallel algorithms have been designed for this purpose [30, 1, 10], and with the advent of both new technologies (e.g. GPGPUs) and new use cases (e.g. big data analysis) sorting continues to be a relevant field of research [48, 43].

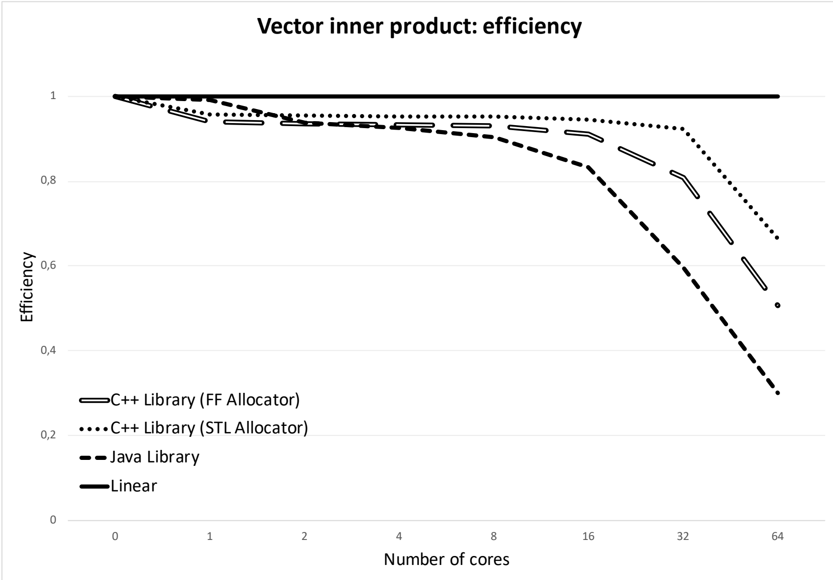


Figure 5.5: Efficiency for the vector inner product benchmark.

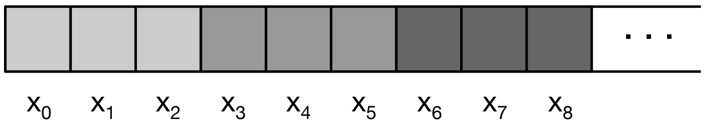


Figure 5.6: An example of block distribution of an array over 3 processors.

There exists a number of algorithms for parallel sorting on the BSP model [23, 24]. Between them, we chose Tiskin's version of the sorting by regular sampling algorithm [55, 46, 32], which is easy to understand and allows us to illustrate another form of distribution of the input, the *block distribution* (Figure 5.6). In this kind of distribution, each processor is assigned a contiguous block of the input: with a problem of size n and p processors, each one receives n/p elements. The block distribution can lead to poor load balancing if n/p is not an integer number.

The algorithm, as its name implies, is guided by *regularly-spaced samples* of the data elements. First of all, each processor sorts its own subarray using a sequential algorithm. After that, it selects $p + 1$ regularly-spaced elements from the sorted subarray (the first, the $(p + 1)$-th, the $(2p + 1)$-th, ..., the last). We call these elements the *primary samples* for that subarray. The p sets of $p + 1$ primary samples are then collected and sorted sequentially. The sampling process is applied again, this time to the sequence of sorted primary samples, to obtain a set of $p + 1$ *secondary samples* which are broadcast to all the processors. The secondary samples partition the original array into p blocks. Each processor sends all elements of the i-th block to the i-th processor. After all the blocks are distributed, the processors sort (sequentially) the received elements. The input array is now sorted in the sense that the first processor holds the first (ordered) block, the second processor holds the second block, and so on. Optionally, all blocks can be collected in a single processor afterwards.

The test creates an array $\vec{x}$ containing the first $n = 2^{24}$ natural numbers and shuffles it randomly. In this way, the algorithm correctness check is simply $\forall i. 0 \leq i < n : x_i = i$.

The sequential algorithms simply use the respective language library's *sort* function to sort the input. The input array is represented in the same way in both parallel and sequential applications: in Java it is stored as an `ArrayList<Integer>`, in C++ as a `std::vector<int>`. This is significant in the Java case, since sorting an `ArrayList<Integer>` (using the `Collections.sort()` function) can be up to *ten times* slower with respect to sorting an `int[]` using `Arrays.sort()` [1]. The performance metrics in the Java case are therefore to be interpreted having in mind the fact that the sequential program can be vastly improved by changing the input representation. Conversely, in C++ the difference between sorting a `std::vector<int>` and a `int[]` is negligible.

We could not obtain data for $p = 64$ for the Java parallel program. The execution with this parallelism degree on the machine detailed in Section 5.2 constantly failed with a `NullPointerException` error encountered when performing communication at the end of a superstep. We did not manage to reproduce the error on the machine used in the development, which used a more recent version of the Java Runtime Environment. As the MulticoreBSP for Java library has not been updated in years, it is unknown what causes this problem. As a consequence, in the following performance plots, data for $p = 64$ is only available for the C++ library.

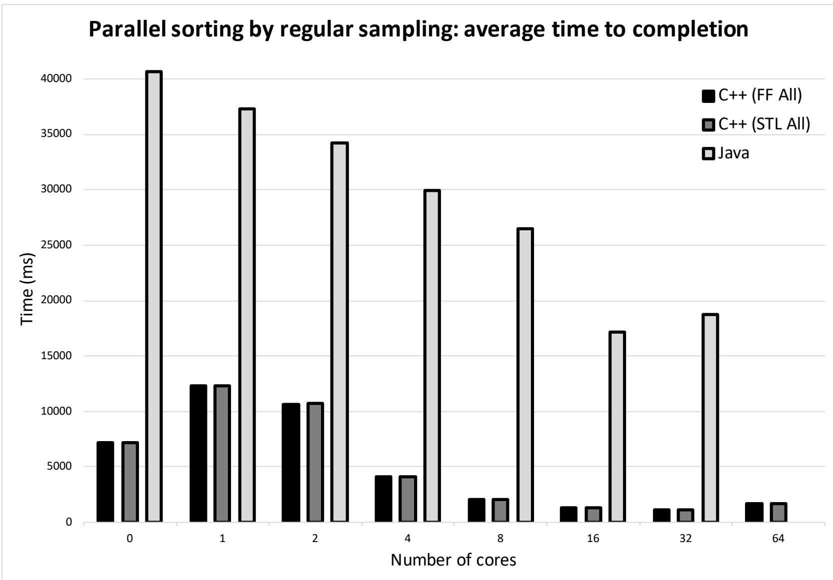


Figure 5.7: Average completion time for the PSRS benchmark on 2^{24} elements.

[1] The bottleneck is not in the sorting function, as `Collections.sort()` actually uses `Arrays.sort()` as the sorting subroutine. To the best of our knowledge, the problem lies in the additional book-keeping needed for maintaining `Integer` wrappers and `ArrayList` objects.

Figure 5.7 shows the average completion time for the sorting benchmark. The Java implementation actually achieves superlinear speedup for $p = 1$: in line with what we said before, this is due to the fact that in the first phase of the algorithm, the input array is distributed in a structure which relies on an `int[]` as the underlying data organization (the `BSP_INT_ARRAY` class of the MulticoreBSP for Java library). The other phases of the algorithm use `ArrayList<Integer>`, but at that point the array is at least partially ordered. This may explain the slight advantage in using the parallel program with a single thread over the sequential one. The C++ implementations do not benefit from different data organization, so at $p = 1$ they show the increased latency due to multiple sorting passes. As the parallelism degree increases, the Java library shows a modest decrement of latency up to $p = 16$. The C++ implementations follow a similar trend, instead managing to decrease latency up to $p = 32$. Overall, the C++ implementations are remarkably faster than the Java one, with the slab allocator version gaining a very slight edge over the STL allocator one.

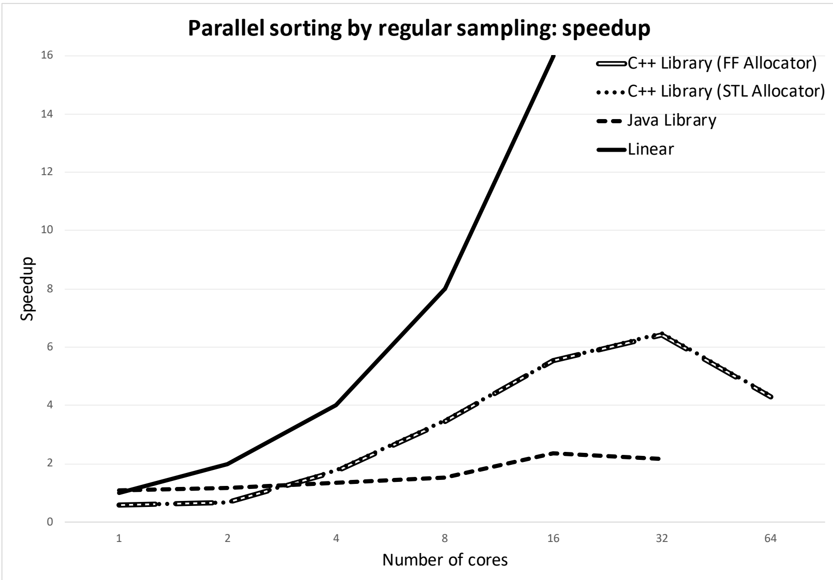


Figure 5.8: Speedup for the PSRS benchmark.

The speedup plot of Figure 5.8 offers an overview of the implementations' behavior for this benchmark. None of them manage to reach linear speedup, except for the Java one at $p = 1$ that we have discussed before. The two C++ implementations do not show noticeable differences between them; they both achieve better results with regards to the Java implementation for $p \geq 4$ and perform worse for $p = 64$ than for $p = 32$. The Java implementation shows almost constant speedup throughout the test.

Figure 5.9 refers to the scalability metrics of the benchmark. It is similar to the speedup plot of Figure 5.8 in that no implementation manages to get closer to linear scalability. We notice, although, that unlike in the other metrics

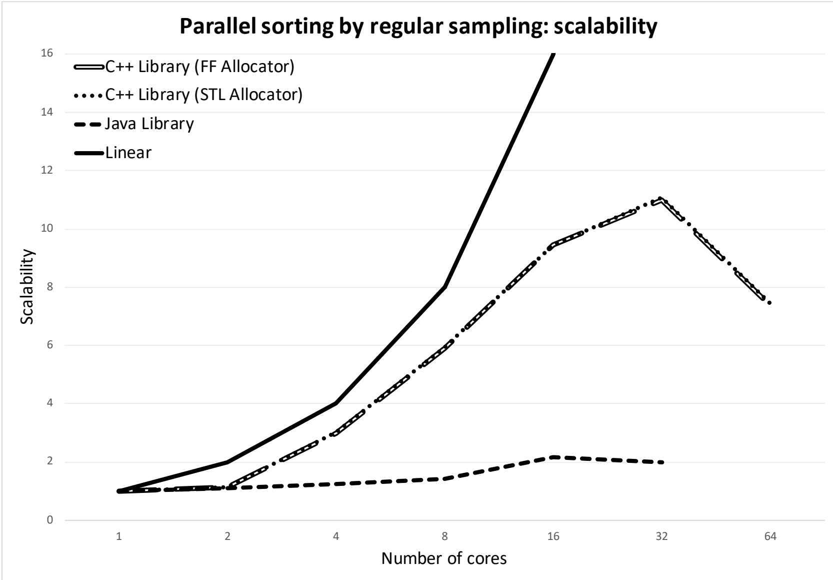


Figure 5.9: Scalability for the PSRS benchmark.

the Java implementation does not perform better than the C++ ones for lower values of p.

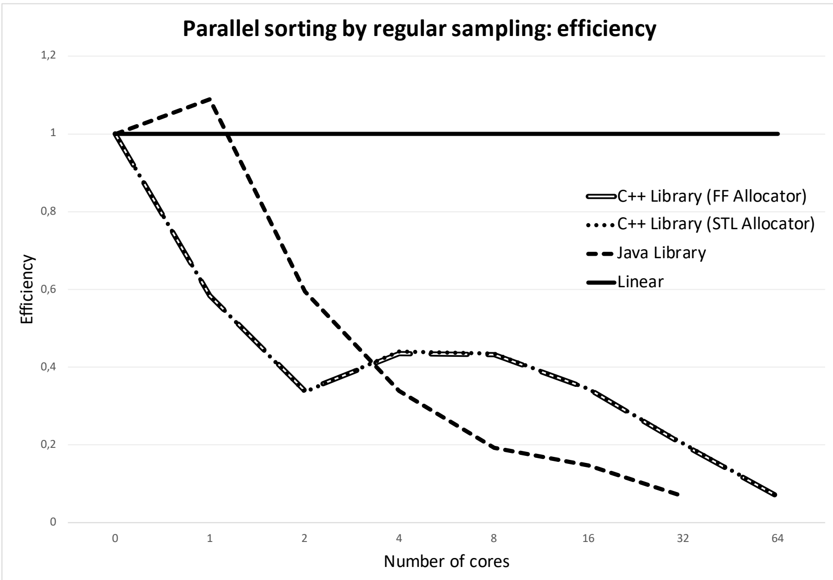


Figure 5.10: Efficiency for the PSRS benchmark.

Lastly, Figure 5.10 show the efficiency plot for this benchmark. As we anticipated when talking about speedup, the Java implementation with one worker performs slightly better than the sequential program, thus achieving an efficiency greater than 1 (about 1.08). The efficiency rapidly declines as the parallelism degree increases: at $p = 8$ it reaches 20% of the maximum, and the trend continues only to worsen as p increases. The C++ implementations immediately sink under 0.5 efficiency for $p \geq 2$, but manage to outperform the Java implementation for $p \geq 4$. Overall, none of the three libraries provide satisfying performances for this program.

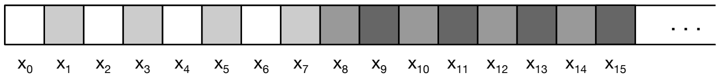


Figure 5.11: The group-cyclic distribution with $n = 16$, $p = 4$ and $c = 2$.

5.3.3 Fast Fourier Transform

Periodic functions are encountered in basically every field, from music to medical imaging to telecommunications and more. On computers, these functions can only be represented by the value they take at sample points: a song on an audio CD is sampled 44100 times per second, an MRI image is typically composed by 512×512 pixels, and so on. The **Discrete Fourier Transform** (DFT) of a sequence of points sampled from a continuous function converts it into a sequence of points in the frequency domain, which are themselves sample points of the *discrete-time Fourier transform* (DTFT). If the DTFT of a continuous function is known, then – under certain conditions – the original continuous function can be perfectly recovered from it. The DFT is therefore one of the fundamental component in digital signal processing and as such has been extensively studied.

Let $\vec{x}$ be a vector of n complex numbers $\vec{x} = (x_0, \ldots, x_{n-1}) \in \mathbb{C}^n$. The DFT of $\vec{x}$ is the vector $\vec{y} = (y_0, \ldots, y_{n-1}) \in \mathbb{C}^n$ such that

$$y_k = \sum_{j=0}^{n-1} x_j e^{-2\pi i jk/n} \qquad 0 \leq k < n \tag{5.6}$$

The straightforward application of the mathematical definition of DFT over a sequence of n samples has complexity $O(n^2)$. The **Fast Fourier Transform** is a class of algorithms that compute the DFT of a sequence of length n in time $O(n \log n)$ instead. The BSPedupack (see Section 5.3) contains an adaptation of the Cooley-Tukey algorithm [18] for calculating the FFT, which has therefore been ported to our C++ library. A detailed explanation of the BSP algorithm is outside the scope of this work, and the interested reader can find it in [9]. Here we limit ourselves to note that the `bspfft` program employs yet another kind of distribution for the input, which in a sense is an intermediate step between the cyclic distribution of Figure 5.1 and the block distribution of Figure 5.6. The *group-cyclic* distribution of n elements over p processors with cycle c is defined as follows: the j-th element of the input is assigned to the k-th processor, where

$$k = \left(\left(j \text{ div } \left\lceil \frac{cn}{p} \right\rceil \right) c + \left(\left(j \bmod \left\lceil \frac{cn}{p} \right\rceil \right) \bmod c \right) \right) \qquad 0 \leq j < n. \tag{5.7}$$

The distribution is defined for every c such that $1 \leq c \leq p$ and $p \bmod c = 0$. Figure 5.11 shows an example with $n = 16$, $p = 4$, $c = 2$. This type of distribution partitions the input into blocks of size $\lceil cn/p \rceil$; each block is then assigned to a group of c processors according to the cyclic distribution. For $c = 1$, the group-cyclic distribution becomes the "normal" block distribution, while for $c = p$ this reduces to the normal cyclic distribution. The `bspfft` program computes the FFT of a vector by redistributing the input at each step, according to group-cyclic distributions with different parameter c, and then computing a sequential FFT over the elements in the same processor. In fact, the redistribution subroutine is the only one that performs communication in this program.

The sequential implementation of the FFT follows the same general behavior of the parallel version, for consistency. Both sequential and parallel version actually execute the algorithm twice (once for the direct FFT, once for the inverse), so the latency for a single pass can be roughly deducted by halving the time spent executing the whole program.

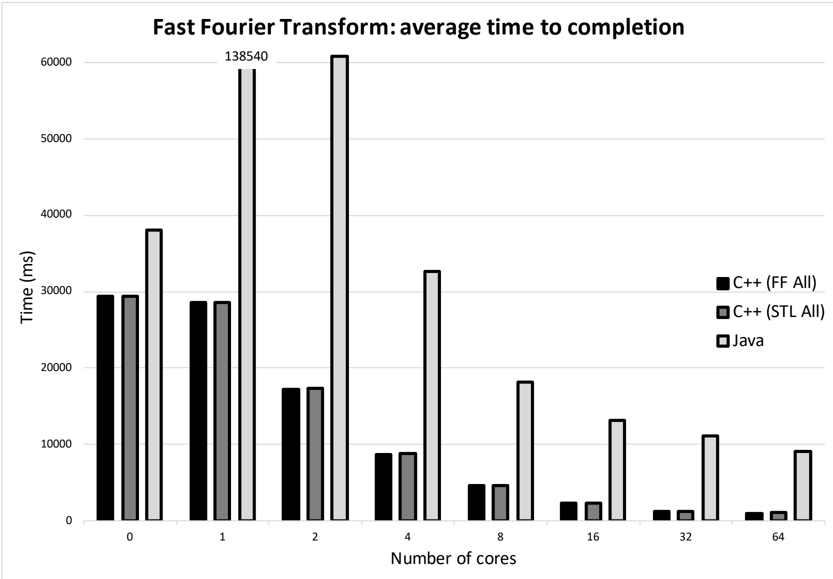


Figure 5.12: Average completion time for the FFT benchmark on 2^{25} elements.

Figure 5.12 shows the completion time chart relative to this benchmark. For $p = 1$ we notice that the C++ implementations perform faster than the sequential algorithm, albeit only slightly so. As there's no communication between processors even in the parallel version with one worker (since no redistribution ever happens), similar results with regards to the sequential version were expected. Both C++ implementations show a good decrease in latency when the parallelism degree increases. The version that uses the slab allocator generally performs a little better than its STL allocator counterpart. The Java implementation, on the other hand, performs poorly for low values of p, requiring at least 4 workers to marginally reduce the latency with respect to the sequential program. For $p \geq 8$ the decrease in latency is acceptable, though the performance of the Java library is constantly inferior to both C++ implementations.

The speedup plot of Figure 5.13 confirms the analysis of the previous paragraph. The Java library never reaches even radical ($\sqrt{p}$) speedup, while the C++ implementations achieve near-linear speedup for $p \leq 8$ and good results up to the ceiling of $p = 64$, with the slab allocator version performing slightly better.

Figure 5.14 offers a surprisingly different scenario for low parallelism degrees: the Java library manages to attain a slightly superlinear scalability up to

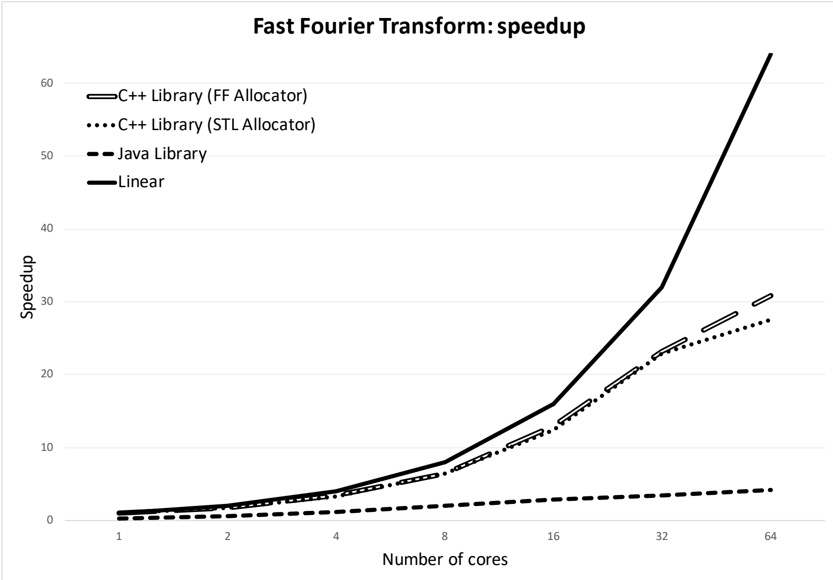


Figure 5.13: Speedup for the FFT benchmark.

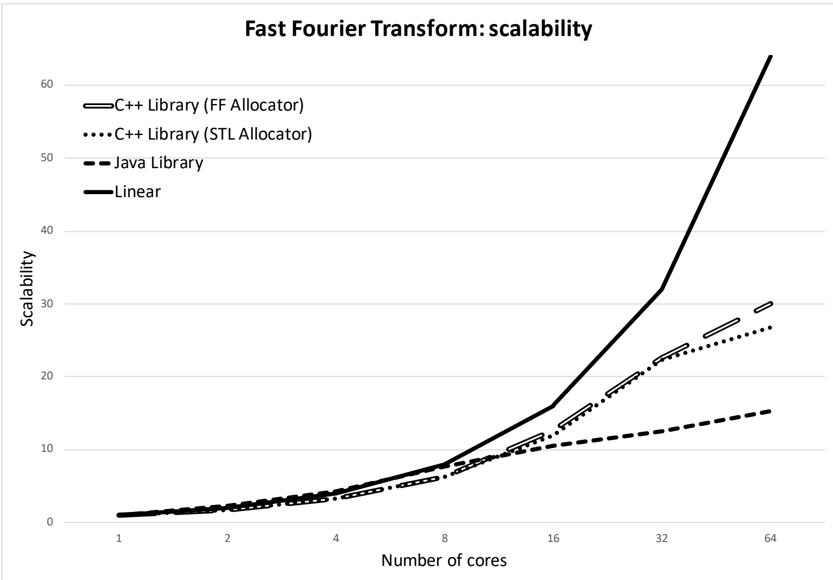


Figure 5.14: Scalability for the FFT benchmark.

$p \leq 8$. This suggests that the poor absolute performance of this library is due to setup costs that get amortized at higher parallelism degrees. Once again, the C++ implementations show good performances for any value of p, although the scalability increases slowly for $p \geq 32$.

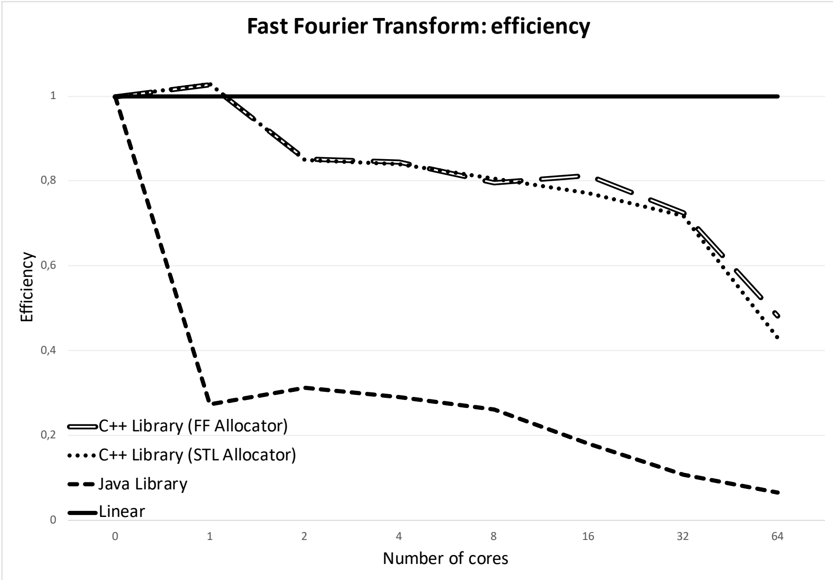


Figure 5.15: Efficiency for the FFT benchmark.

The efficiency plot (Figure 5.15) sums up the observations made in the last few paragraphs. Both C++ versions of the library manage to never fall lower than 0.8 efficiency until $p = 8$, with the slab allocator implementation maintaining the advantage over its counterpart throughout all parallelism degrees. The Java library, on the other hand, falls early and – after a modest increase for $p = 2$ – manages only to maintain a slowly descending rate for the efficiency metrics.

5.3.4 Matrix LU decomposition

The last algorithm we consider is another numerical one that is widely used in a broad series of contexts. Let A be a $n \times n$ nonsingular matrix and $\mathbf{b}$, $\mathbf{x}$ be two vectors of length n, of which the second is not known. $\mathbf{x}$ is the solution of the linear system

$$A\mathbf{x} = \mathbf{b}. \tag{5.8}$$

One of the possible methods for solving this system is by employing the **LU decomposition** of the matrix A, i.e. finding two $n \times n$ matrices L and U such that L is unit lower triangular and U is upper triangular and

$$A = LU.$$

A $n \times n$ matrix M is *unit lower triangular* if $m_{ii} = 1$ and $m_{ij} = 0$ for $i < j$, and is *upper triangular* instead if $m_{ij} = 0$ for $i > j$.

The matrices L and U can be used to find solutions of the system $A\mathbf{x} = \mathbf{b}$ by solving the triangular systems

$$L\mathbf{y} = \mathbf{b} \quad \text{and} \quad U\mathbf{x} = \mathbf{y} \tag{5.9}$$

since solving triangular systems is easy (in a mathematical sense). The advantage of LU decomposition over other approaches (e.g. Gaussian elimination) is that the L and U matrices can be reused when solving a different system such as

$$A\mathbf{x} = \mathbf{b}'.$$

LU decomposition is also used as a step of the solution of various other problems such as inverting a matrix or calculating its determinant.

	0	1	2	0	1
0	00	01	02	00	01
1	10	11	12	10	11
0	00	01	02	00	01
1	10	11	12	10	11
0	00	01	02	00	01

Figure 5.16: Cyclic distribution of a 5×5 matrix over 2×3 processors.

As with the FFT algorithm, the BSPedupack version of the algorithm was ported to the C++ library implemented in this work. We again point the reader to refer to [9] for a detailed explanation of the algorithm, and focus only on the input distribution and communication patterns. The algorithm actually solves the related problem

$$PA = LU$$

where P is a permutation matrix that reorders the rows of A. This family of LU decomposition algorithms are called *with partial pivoting*, since at each step they find *row pivots* that guide the computation by indicating which rows to swap.

Like the vector inner product program, this algorithm makes use of a cyclic distribution. In order to evenly distribute a matrix, though, processors need to be (at least logically) organized in a 2D grid. Each processor is assigned a row index s and a column index t and will be therefore referred as p_{st}. The BSP LU decomposition algorithm distributes the input matrix according to a cyclic distribution over *both* rows and columns. This means that processors p_{s*} will each hold an element of row s and processors p_{*t} will each hold an element of column t. Figure 5.16 shows how a 5×5 matrix is distributed over 6 processors in a 2×3 grid.

Both the matrix distribution and the processor organization help in reducing the number of communications at each step. The algorithm requires that every update of an element in the matrix is broadcast to all the other processors, so that they can update their own elements. Since this operation is costly, a *two-phase broadcast* strategy is employed: in the first phase, the node sends each element of the vector it must broadcast to a different intermediate peer. In the second phase, each intermediate node send copies of the received ele-

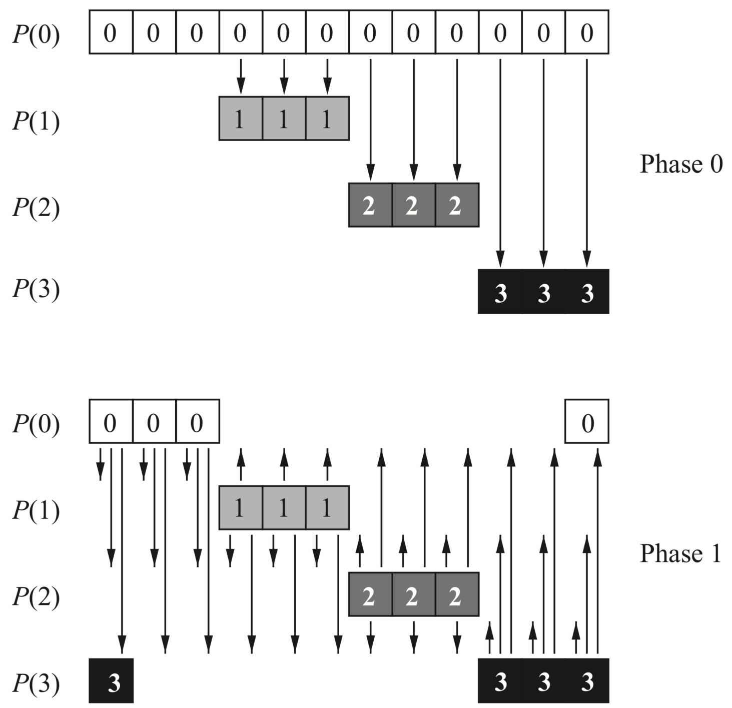


Figure 5.17: A two-phase broadcast of a vector of twelve elements over four processors (taken from [9]).

ment to the final destination. An example of two-phase broadcast is shown in Figure 5.17.

Initially, the sequential version of both the C++ and Java programs employed the sequential Doolittle LU decomposition algorithm [54, 11]. Their performance, however, was much lower than the parallel program using one thread. We therefore decided to use external libraries for this purpose: ALGLIB for the C++ version [34] and Apache Commons Math for the Java version [35].

All tests are run on a 5000×5000 input matrix. The parallel versions use the following processor grid configurations:

- 1×1 ($p = 1$);
- 1×2 ($p = 2$);
- 2×2 ($p = 4$);
- 2×4 ($p = 8$);
- 4×4 ($p = 16$);
- 4×8 ($p = 32$);
- 8×8 ($p = 64$).

The average completion time shown in the chart of Figure 5.18 immediately reveals that this benchmark is by far the heaviest for all the implementations,

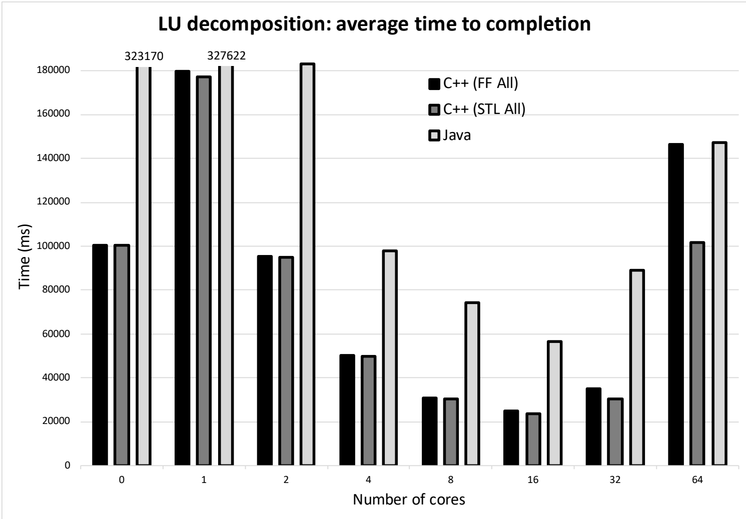


Figure 5.18: Average completion time for the LU decomposition of a 5000×5000 matrix.

with latencies that in some cases reached over 350 seconds. The sequential implementation in Java (Apache Commons library) is sensibly less efficient than the C++ one (ALGLIB), with a latency more than three times higher. Both C++ implementations of the parallel library exhibit a higher latency with one worker with respect to the sequential application. The situation improves for higher parallelism degrees up to $p = 16$, after which the computation starts to take longer again. The STL allocator version of the C++ library performes better than the slab allocator version, especially for $p \geq 16$. The Java parallel implementation with a single worker is nearly on par with the sequential version. After that it behaves like the other two implementations, peaking at $p = 16$ and degrading for higher parallelism degrees.

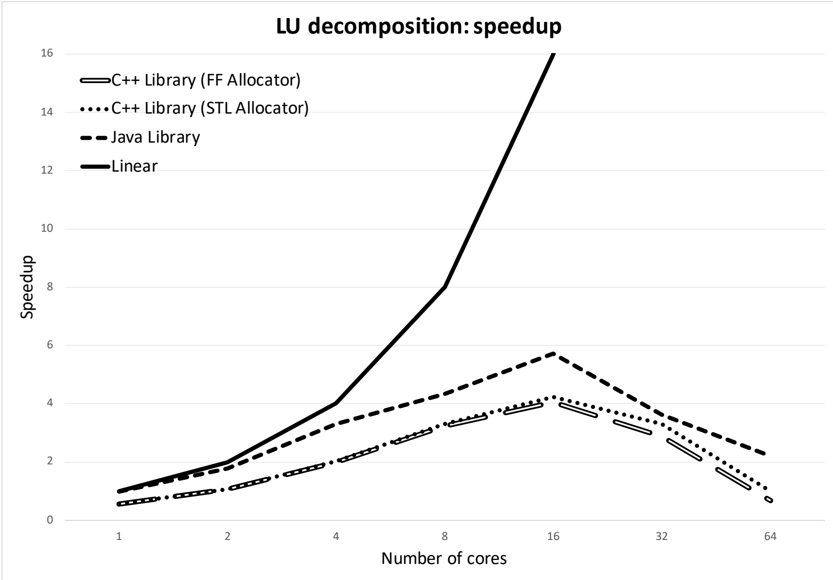


Figure 5.19: Speedup for the matrix LU decomposition benchmark.

The speedup metrics, by definition, strongly depends on the performance of the sequential program. Since the C++ one took less than a third of the time needed for its Java counterpart, it should come to no surprise that the plot of Figure 5.19 shows the Java parallel library on top. All three implementations follow a similar trend, slowly increasing for $p \leq 16$ and then rapidly decreasing thereafter. The two C++ implementations are basically on par with each other until the peak at $p = 16$, after which the slab allocator version falls behind.

Figure 5.20 shows the scalability plot. Since the Java library performs nearly the same as the sequential version for $p = 1$, its scalability is practically on par with its speedup. Conversely, the C++ implementations fare better here, as they are not compared with the highly efficient ALGLIB sequential program anymore.

Finally, the efficiency for this benchmark can be found in Figure 5.21. The same considerations we made for the speedup apply here: the Java implementation "looks better" due to the relatively poor performance of the sequential version. The C++ library variants behave the same, with the STL allocator having a slight edge. All three implementations reach less than 0.1 efficiency for $p \geq 32$. This poor performance may be explained by the exponentially higher amount of communication needed for the algorithm as p increases, especially due to the two-phase broadcast detailed before.

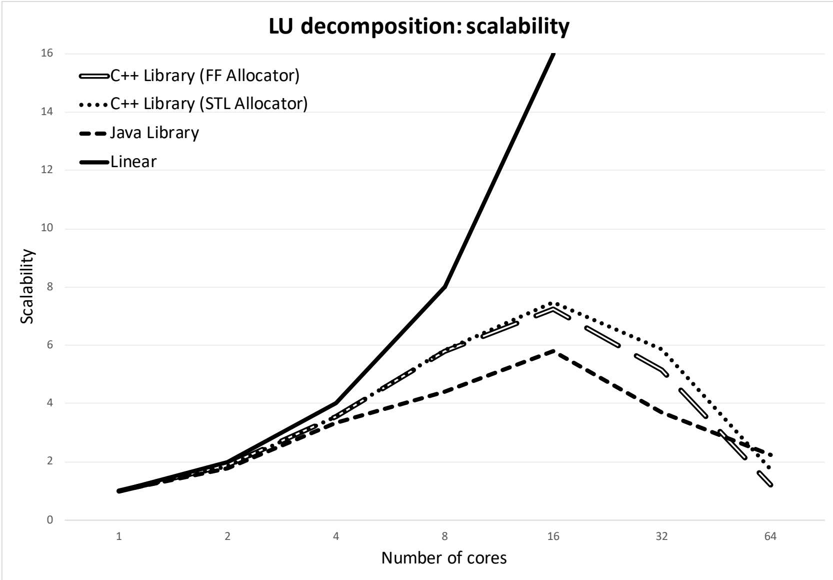


Figure 5.20: Scalability for the matrix LU decomposition benchmark.

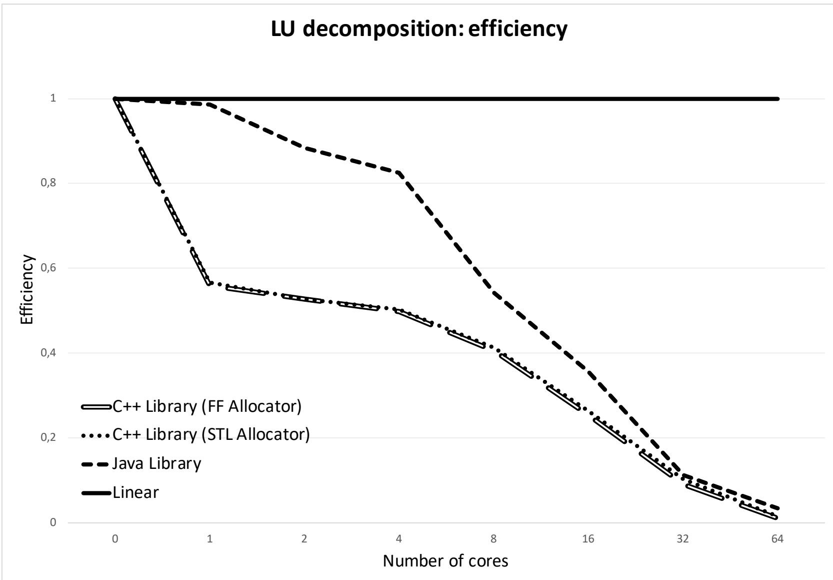


Figure 5.21: Efficiency for the matrix LU decomposition benchmark.

5.3.5 Other programs

Some other programs were implemented for this work. The BSPedupack contains a "benchmark" of sorts, called `bspbench`, that stresses both communication and computation aspects of the machine on which it runs. It can also be used to derive the parameters g, h, L of the BSPC implemented in the library. We provided this program (`BSPbench.cpp`), along with a custom-designed communication stress test (`commstress.cpp`) and a battery of unit tests (`unit_tests.cpp`), the latter two for checking the correctness of the implementation.

The `bspbench` implementation reported the following values on the machine used for the performance evaluation.

p	r[A]	g	l
1	1170.29	597.108	3666.18
2	1170.29	868.933	35265.2
4	1170.29	3591.81	91652.2
8	1170.29	6183.59	215230
16	1170.29	12593.1	465453
32	1170.29	51731.2	1916450

[A] Mflops/s

Figure 5.22: BSP parameters for the Phi machine as reported by `bspbench`.

5.4 Summary

In this Chapter we measured the performance of the library implemented in this work on a variety of commonly-used applications, both numerical and non-numerical. The experimental results are very promising: while the absolute performance depends on the algorithm, the BSP implementations that used our library showed a decent-to-good increase in performance at any parallelism degree. The other main "competitor" in the field of object-oriented modern BSP implementations, the MulticoreBSP for Java library, is consistently outranked by our solution. Although the native difference in performance between the Java and C++ languages helped achieving good results, we consider our implementation choices to have played a bigger role in determining the final difference in performance.

Chapter 6

Conclusion

In this thesis we introduced the design and implementation of a C++ library for the Bulk Synchronous Parallel paradigm, targeting shared memory multicore machines. The architectural design draws its inspiration from Tiskin's BSPRAM model [55], but maintains the communication and the shared memory as separate concepts. The implementation uses the compositional capabilities of the FastFlow framework to build its internal structure, which happens to be a plain farm template, actually. The library can also be considered an extension of FastFlow, as the whole BSP computation ultimately takes place in a FastFlow node which can be freely used inside any other parallel pattern.

Both the design and implementation aspects of the library varied a lot during the development of this work. At first, we thought about providing an all-new programming style for BSP computations: each superstep would be represented as a C++ object `superstep<Tin, Tout>`, with clearly defined input and output types. These objects were to be given, in the desired order, to a BSP "executor" that would regulate the superstep flow and perform the requested communications. We soon realized that this programming style was not necessarily easier to use, instead proving to be hugely limiting on the communication capabilities (each node could only send either a single object or a collection of same-type objects per superstep).

After this "superstep-centric" draft, we briefly reasoned upon an architecture where nodes were arranged in a fully or partially connected mesh, in order to remove the need for an entity that manages communications. The channels between nodes were to be implemented using FastFlow SPSC queues (see Section 2.2). This approach was quickly dismissed, as a fully connected mesh would require a huge number of queues even for modest parallelism degrees (with 64 nodes there need to be 4096 channels), and a partially connected mesh would not improve communication times too much. Moreover, this approach didn't fully utilize FastFlow's parallel patterns, instead relying on its low-level mechanisms. While this is certainly allowed by the framework, we wanted to exploit the easiness of use provided by its algorithmic skeletons.

Finally, we converged to a satisfying architectural model and implementation scheme, the ones we discussed in Chapters 3 and 4. Of course there was still a lot to improve: particular focus was given on finding the most efficient representation of shared objects in the memory, due to the difficulties of having to work with heterogeneous data to be stored inside a single container.

Another aspect that required particular attention was the requests mechanism, as it put a lot of stress on the number of end-of-superstep operations to be executed, leading to a lower performance. A common pattern we noticed in many BSP computations was that nodes often performed a high number of communication requests per superstep, but each communication only regarded one or few data elements (which, we remind, must be copied into the request object). The communicator therefore had to continuously allocate and deallocate many small request objects. We reasoned that the standard C++ allocator could be less efficient in this scenario, and sought after different allocation schemes which better suited our needs. Fortunately, FastFlow provided a custom solution based on the slab allocator (Section 2.2.1), which was therefore used in the library.

At the end of this implementation phase we obtained a final product that matched the goals we set for ourselves. The library provides support for BSP programs with a simple and clean API. The run-time support relieves users from having to manage memory operations and shared variable registration and deregistration, unlike many other BSP libraries. Programmers can code in a lean and modern way, relying on the object-oriented paradigm and supporting all C++11 (and above) features. The BSP computation can be performed over any type of data, including user-defined classes and objects. As we will see in a moment, the library's performance is sound, as it shows good scalability for basically any parallelism degree. The library is also integrated in the FastFlow framework, as the BSP computation can use input data received from other nodes and can send output to them.

We extensively tested our solution, both during and after the final stages of development. For conducting performance analysis, we chose four BSP algorithms to be used as benchmark and implemented them. Each one of those programs feature a different communication scheme and a different partitioning of the input between nodes. Three of the four programs are taken from Bisseling's BSPedupack [9], a collection of BSP algorithms for solving common numerical problems. The last one is a sorting algorithm, a crucial operation in basically any field of computer science. Each program was run multiple times using a different number of worker units, to check how the library performed at different parallelism degrees. In addition to the "normal" library, we also ran tests over a variant of the final implementation that used the standard C++ STL allocator, to test if the slab allocator effectively improved performance. We compared our results with identical tests for the MulticoreBSP for Java library. We used the BSPedupack for Java implementation of the three edupack programs and wrote our own BSP sorting program for Java.

The experimental results proved that our implementation is sound. We consistently achieved lower latencies than the Java library, with a large enough margin that allows us to ignore the inherent gap between the two languages. In some cases, our implementation was over ten times faster than the equivalent Java version. With few exceptions, our implementation also presented better scalability for any parallelism degree up to the machine's number of cores (in this case, 64). Lastly, we obtained good results also for the speedup and efficiency parameters, after factoring that C++ sequential versions of the programs were faster than Java ones, therefore "polluting" these two last performance indices. To our surprise, both the STL allocator version and the slab (FastFlow) allocator one produced similar results. We expected the slab alloca-

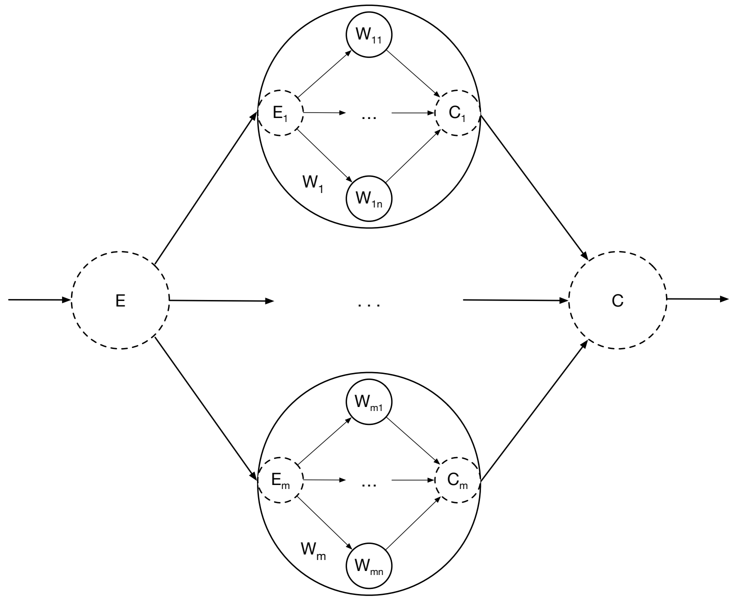


Figure 6.1: FastFlow graph of a possible implementation of the MultiBSP model. The solid line nodes compose the MultiBSP tree.

tor version to gain a more consistent edge in the abovementioned scenario of many requests with small payloads.

6.1 Future work

As mentioned above, the fact that the STL allocator provide similar results to the slab allocator (even outperforming it in some scenarios) can be studied more in-depth. A reasonable approach would be to allow the programmer to choose the desired allocation strategy, potentially even letting him/her specify a custom allocator.

An interesting direction for future work is the implementation of the Multi-BSP model (introduced in [60]) via exploitation of FastFlow's compositional capabilities. Since the whole BSP computation is already a FastFlow node, a nested BSP run can be achieved simply by allowing the computation node to act as a BSP node, therefore becoming part of the broader composition. The MultiBSP tree in this case corresponds to the FastFlow graph, minus the farm emitter and collector entities (Figure 6.1).

We could, lastly, exploit FastFlow's support for different architectures in order to provide other BSP variants. For example, we can use FastFlow `ff_dnodes` [4] to run BSP programs on a distributed computer, therefore "closing the circle" by providing support for the same architecture described in Valiant's seminal work. Another interesting topic, in the same vein, could be to provide support for offloading work to GPU accelerators. Both these directions also work well within the abovementioned MultiBSP paradigm.

While these features weren't implemented due to time constraints, we ultimately feel that they concern a slightly broader scope that the one we considered in this work, namely to realize an efficient and simple-to-use BSP library for shared-memory multicores integrated in the FastFlow framework. We think that the implementation we provided in this thesis matched our goals.

Appendix A

Library quick start

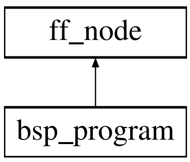
This Appendix details the API documentation for the library, i.e. public methods and fields that the programmer can use to build a BSP program. A complete documentation, which includes private and internally-used methods and fields, is available as Doxygen comments in the source code. The closing Sections provide additional clarifications about some mechanisms of the `bsp_variable` and `bsp_array` classes.

A.1 API documentation

A.1.1 `bsp_program` Class Reference

```
#include <bsp_program.hpp>
```

Inheritance diagram for `bsp_program`:



Public Member Functions

- ```
  bsp_program(std::vector<std::unique_ptr<bsp_node>>&& _processors,
  std::function<void(void)> _pre = nullptr,
  std::function<void(void)> _post = nullptr)
  ```
- `void start(void* in = nullptr)`
- `void* svc(void* in) override`

Class Description

Implements the Bulk Synchronous Parallel pattern as a FastFlow node.
  ```

Constructor Documentation

bsp_program()

```
explicit bsp_program(std::vector<std::unique_ptr<bsp_node>>&& _processors,
                     std::function<void(void)> _pre = nullptr,
                     std::function<void(void)> _post = nullptr)
```

Constructor for the `bsp_program` object.

Parameters

`_processors`	vector of BSP nodes for the computation
`_pre`	optional function to be executed before the BSP computation
`_post`	optional function to be executed after the BSP computation

Member Function Documentation

start(void*)

```
void start(void* in = nullptr)
```

Creates the FastFlow inner graph and executes the BSP computation.

Parameters

`in`	optional, the input token received from the preceding FastFlow node

---

svc(void*)

```
void* svc(void* in) override
```

Service function for the BSP program node. Starts the BSP computation.

Parameters

`in`	input token received from the preceding FastFlow node

Returns

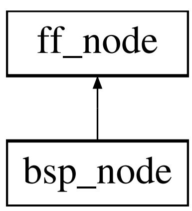
`GO_ON`, a special FastFlow token to continue the broader computation

---

A.1.2 bsp_node Class Reference

```
#include <bsp_node.hpp>
```

Inheritance diagram for `bsp_node`:



Public Member Functions

- `void* svc(void* in) final`

Protected Member Functions

- `void emit_output(void* payload)`
- `template <typename T>`
  `bsp_variable<T> get_variable(const T& initial_value)`
- `template <typename T>`
  `bsp_array<T> get_array(const std::vector<T>& initial_value)`
- `template <typename T>`
  `bsp_array<T> get_array(std::vector<T>* handle)`
- `template <typename T>`
  `bsp_array<T> get_empty_array(int size)`
- `int bsp_pid()`
- `int bsp_nprocs()`
- `void bsp_sync()`
- `virtual void parallel_function() = 0`

Protected Attributes

- `const void* fastflow_input`

  Pointer to the FastFlow input token

Class Description

Specialization of a `ff_node` to work as the unit of computation in the BSP model.

Member Function Documentation

`bsp_nprocs()`

```
int bsp_nprocs()
```

Returns the number of nodes in the current BSP computation.

Returns

the number of nodes in the current BSP computation.

`bsp_pid()`

```
int bsp_pid()
```

Returns the ID for this node.

Returns
this node's ID.

---

bsp_sync()

```
void bsp_sync()
```

Terminates the current superstep and waits for the other nodes to sync.

---

emit_output(void*)

```
void emit_output(void* payload)
```

Forwards an output token to the next stage in the FastFlow graph.

Parameters

payload	the token to forward

---

get_array(const std::vector<T>&)

```
template <typename T>
bsp_array<T> get_array(const std::vector<T>& initial_value)
```

Requests a new array with elements of type T from the communicator, initializing it with the copy a given vector.

Template Parameters

T	the type of elements of the requested array

Parameters

initial_value	value to copy inside this node's private copy of the array

Returns
a handle to this node's private copy of the shared array

---

get_array(std::vector<T>*)

```
template <typename T>
bsp_array<T> get_array(std::vector<T>* handle)
```

Requests a new array with elements of type T from the communicator, initializing it with the pointer to a vector. Any modifications done to the initializing vector after the call to this method are inherently BSP unsafe.

Template Parameters

T	the type of elements of the requested array

Parameters

`handle`	a pointer to the vector that will become this node's private copy of the shared array

Returns

a handle to this node's private copy of the shared array

`get_empty_array(int)`

```
template <typename T>
bsp_array<T> get_empty_array(int size)
```

Requests a new array with elements of type `T` from the communicator, initializing it with an empty vector of given size.

Template Parameters

`T`	the type of elements of the requested array

Parameters

`size`	the size of the empty vector that will become this node's private copy of the shared array

Returns

a handle to this node's private copy of the shared array

`get_variable(const T&)`

```
template <typename T>
bsp_variable<T> get_variable(const T& initial_value)
```

Requests a new variable of type `T` from the communicator.

Template Parameters

`T`	the type of the requested variable

Parameters

`initial_value`	value to copy inside this node's private copy of the variable

Returns

a handle to this node's private copy of the shared variable

`parallel_function()`

```
virtual void parallel_function() = 0;
```

Function to be overwritten as the main parallel execution.

---

svc(void*)

```
void* svc(void* in) final
```

The FastFlow node service method. Implementations of this class cannot redefine it.

Parameters

in	the input token (in this case, a special value ENDCOMP)

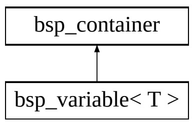
Returns

the same token as the input

A.1.3 bsp_variable<T> Template Class Reference

```
#include <bsp_variable.hpp>
```

Inheritance diagram for bsp_variable<T>:



Public Member Functions

```
• T get()
• void bsp_put(const T& elem, int destination)
• void bsp_put(const bsp_variable<T>& other)
• void bsp_put(const bsp_variable<T>& other, int destination)
• void bsp_put(int destination)
• void bsp_get(int source) override
• T bsp_direct_get(int source)
• T& BSPunsafe_access()
```

Class Description

A variable data structure, private to a single worker but with support for communication with other similar entities.

Template Parameters

T	type of the variable

Member Function Documentation

bsp_direct_get(int source)

```
T bsp_direct_get(int source)
```

Returns a copy of a node's private element of this shared variable. This method will return immediately, without waiting for a superstep sync.

Parameters

`source`	ID of the source node

Returns

a copy of the desired element

`bsp_get(int source)`

```
void bsp_get(int source) override
```

Replaces this node's private element with another node's private element of this shared variable.

Parameters

`source`	ID of the source node

`bsp_put(const bsp_variable<T>&)`

```
void bsp_put(const bsp_variable<T>& other)
```

Replaces this node's private element with this node's private element of another shared variable.

Parameters

`other`	the handle to the other shared variable

`bsp_put(const bsp_variable<T>&, int)`

```
void bsp_put(const bsp_variable<T>& other, int destination)
```

Replaces another node's private element with this node's private element of another shared variable.

Parameters

`other`	the handle to the other shared variable
`destination`	ID of the destination node

`bsp_put(const T&, int)`

```
void bsp_put(const T& elem, int destination)
```

Replaces another node's private element of the shared variable.

Parameters

`elem`	element to be copied
`destination`	ID of the destination node

---

bsp_put(int)

```
void bsp_put(int destination)
```

Replaces another node's private element with this node's private element of the shared variable.

Parameters

destination	ID of the destination node

BSPunsafe_access()

```
T& BSPunsafe_access()
```

Returns a handle to this node's private element of the shared variable. The returned object can be modified at will, without waiting for superstep syncs. This function is **BSP unsafe**.

Returns
a reference to the node's private element of the shared variable

---

get()

```
T get()
```

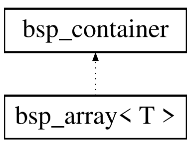
Returns a copy of the node's private element of the shared variable.

Returns
a copy of the desired element

A.1.4 bsp_array<T> Template Class Reference

```
#include <bsp_array.hpp>
```

Inheritance diagram for bsp_array<T>:



Public Member Functions

- T get()
- std::vector<T> get()
- int size()
- void BSPunsafe_put(const T& elem, int pos)
- void bsp_put(const T& elem, int pos)
- void bsp_put(const bsp_variable<T>& elem, int pos)
- void bsp_put(const T& elem, int destination, int pos)
- void bsp_put(const bsp_variable<T>& elem, int destination, int pos)
- void bsp_put(const std::vector<T>& array)
- void bsp_put(const bsp_array<T>& other)
- void bsp_put(const std::vector<T>& array, int destination)
- void bsp_put(const bsp_array<T>& other, int destination)
- void bsp_put(const std::vector<T>& array, int src_offset, int dest_offset, int length)
- void bsp_put(const bsp_array<T>& other, int src_offset, int dest_offset, int length)
- void bsp_put(const std::vector<T>& array, int destination, int src_offset, int dest_offset, int length)
- void bsp_put(const bsp_array<T>& other, int destination, int src_offset, int dest_offset, int length)
- void bsp_get(int from) override
- void bsp_get(int from, int from_offset, int to_offset, int length)
- T bsp_direct_get(int source, int pos)
- std::vector<T> bsp_direct_get(int source)
- std::vector<T>& BSPunsafe_access()

Class Description

An array structure, private to a single worker but with support for communication with other similar entities.

Template Parameters

T	type of the elements contained in the array

Member Function Documentation

bsp_direct_get(int)

```
std::vector<T> bsp_direct_get(int source)
```

Returns a copy of another node's private array. This method will return immediately, without waiting for a superstep sync.

Parameters

source	ID of the source node

Returns
the source's private copy of the shared array

`bsp_direct_get(int, int)`

```
T bsp_direct_get(int source, int pos)
```

Returns a copy of the element at the desired position from another node's copy of the array. This method will return immediately, without waiting for a super-step sync.

Parameters

`source`	ID of the source node
`pos`	position of the desired element

Returns
the element of the source's private copy at the desired position

---

`bsp_get(int)`

```
void bsp_get(int from) override
```

Replaces the node's private copy of the array with another node's private copy.

Parameters

`from`	ID of the source node

`bsp_get(int, int, int, int)`

```
void bsp_get(int from, int from_offset, int to_offset, int length)
```

Replaces a portion of the node's private copy of the array with a portion of another node's private copy.

Parameters

`from`	ID of the source node
`from_offset`	starting position of the replacing portion
`to_offset`	starting position of the portion to be replaced
`length`	length of the replacing portion

`bsp_put(const bsp_array<T>&)`

```
void bsp_put(const bsp_array<T>& other)
```

Replaces the node's private copy of the array with the vector contained inside another `bsp_array`.

Parameters

`other`	other `bsp_array` containing the vector that will replace the current one

`bsp_put(const bsp_array<T>&, int)`

```
void bsp_put(const bsp_array<T>& other, int destination)
```

Replaces another node's private copy of the array with this node's private copy of the vector contained inside another `bsp_array`.

Parameters

`other`	other `bsp_array` containing the vector that will replace the current one
`destination`	ID of the destination node

`bsp_put(const bsp_array<T>&, int, int, int, int)`

```
void bsp_put(const bsp_array<T>& other, int destination,
    int src_offset, int dest_offset, int length)
```

Replaces a portion of another node's private copy of the array with a portion of this node's private copy of the vector contained inside another `bsp_array`.

Parameters

`other`	other `bsp_array` containing the vector with the desired portion
`destination`	ID of the destination node
`src_offset`	starting position of the replacing portion
`dest_offset`	starting position of the portion to be replaced
`length`	length of the replacing portion

`bsp_put(const bsp_array<T>&, int, int, int)`

```
void bsp_put(const bsp_array<T>& other, int src_offset,
    int dest_offset, int length)
```

Replaces a portion of this node's private copy of the array with a portion of this node's private copy of the vector contained inside another `bsp_array`.

Parameters

`other`	other `bsp_array` containing the vector with the desired portion
`src_offset`	starting position of the replacing portion
`dest_offset`	starting position of the portion to be replaced
`length`	length of the replacing portion

bsp_put(const bsp_variable<T>&, int, int)

```
void bsp_put(const bsp_variable<T>& elem, int destination, int pos)
```

Puts an element (contained in a `bsp_variable`) into the given position in another node's private copy of the array.

Parameters

`elem`	`bsp_variable` that contains the element to be inserted
`destination`	ID of the destination node
`pos`	position in the array

---

bsp_put(const bsp_variable<T>&, int)

```
void bsp_put(const bsp_variable<T>& elem, int pos)
```

Puts an element (contained in a `bsp_variable`) into the given position in this node's private copy of the array.

Parameters

`elem`	`bsp_variable` that contains the element to be inserted
`pos`	position in the array

---

bsp_put(const std::vector<T>&)

```
void bsp_put(const std::vector<T>& array)
```

Replaces the node's private copy of the array with the given vector.

Parameters

`array`	vector that will replace the current private copy of the shared array

---

bsp_put(const std::vector<T>&, int)

```
void bsp_put(const std::vector<T>& array, int destination)
```

Replaces another node's private copy of the array with the given vector.

Parameters

`array`	vector that will replace the current private copy of the shared array
`destination`	ID of the destination node

---

bsp_put(const std::vector<T>&, int, int, int, int)

```
void bsp_put(const std::vector<T>& array, int destination,
    int src_offset, int dest_offset, int length)
```

Replaces a portion of another node's private copy of the array with a portion of the given vector.

Parameters

`array`	vector that will replace the current private copy of the shared array
`destination`	ID of the destination node
`src_offset`	starting position of the replacing portion
`dest_offset`	starting position of the portion to be replaced
`length`	length of the replacing portion

bsp_put(const std::vector<T>&, int, int, int)

```
void bsp_put(const std::vector<T>& array, int src_offset,
    int dest_offset, int length)
```

Replaces a portion of this node's private copy of the array with a portion of the given vector.

Parameters

`array`	vector that will replace the current private copy of the shared array
`src_offset`	starting position of the replacing portion
`dest_offset`	starting position of the portion to be replaced
`length`	length of the replacing portion

bsp_put(const T&, int, int)

```
void bsp_put(const T& elem, int destination, int pos)
```

Puts an element into the given position in another node's private copy of the array.

Parameters

`elem`	element to be inserted
`destination`	ID of the destination node
`pos`	position in the array

bsp_put(const T&, int)

```
void bsp_put(const T& elem, int pos)
```

Puts an element into the given position in this node's private copy of the array.

Parameters

`elem`	element to be inserted
`pos`	position in the array

`BSPunsafe_access()`

```
std::vector<T>& BSPunsafe_access()
```

Returns a handle to this node's private element of the shared array. The returned object can be modified at will, without waiting for superstep syncs. This function is **BSP unsafe**.

Returns

a reference to the node's private copy of the array

`BSPunsafe_put(const T&, int)`

```
void BSPunsafe_put(const T& elem, int pos)
```

Puts an element into the contained array in the desired position. This function is **BSP unsafe**.

Parameters

`elem`	element to be inserted
`pos`	position in the array

`get()`

```
std::vector<T> get()
```

Returns a copy of the node's private element of the shared array.

Returns

a copy of the array

`get(int)`

```
T get(int position)
```

Returns a copy of the element in a given position of the node's private copy of the shared array.

Parameters

`position`	position in the array

Returns
a copy the array element in the desired position

---

size()

```
int size()
```

Returns the size of the array.

Returns
the size of the array

A.2 Variable/array request mechanism

The main way to perform communication between BSP processors is using `bsp_variables` (or their specialization `bsp_arrays`). `bsp_variable<T>`s are special containers that hold an element of type `T` for each BSP node, i.e. each node will refer to a different - private - element using the same `bsp_variable` handle.

`bsp_arrays` are specializations of `bsp_variables` for variable-sized array types. A `bsp_array<T>` is basically equivalent to a `bsp_variable<std::vector<T>>`, except it also provides functions to efficiently get and put single elements or portions of the array instead of having to work with the whole object. In the following, any mention of `bsp_variable` mechanisms is also valid for `bsp_arrays`, unless where noted.

`bsp_variable` objects cannot be built using the standard initialization techniques; instead, they are created by the appropriate `get_variable<T>` method of the `bsp_node` class. In reality, `bsp_variable` objects are mere handles to the actual containers, which have different lifecycles. A new underlying container for variables of type `T` is created by the system when a BSP node requests one more `bsp_variable<T>` than the other BSP nodes; when the other BSP nodes "catch up" and make one more request for another `bsp_variable<T>`, the system gives them a handle to this freshly-created container. In short, for all nodes it holds the following:

Proposition 2. The `bsp_variable<T>` object obtained by the n-th consecutive call to `get_variable<T>` in a given superstep is a handle for the n-th container of objects of type `T`, regardless of everything else such as variable names.

The following example (listing A.1) helps understand these mechanisms. Suppose we have a BSP program with two nodes that execute this program:

```
void parallel_function() override {
    int id = bsp_pid();
    if (id == 0) {
        auto v1 = get_variable<int>(0);
        sleep(1000);
        auto v2 = get_variable<int>(0);
        auto v3 = get_variable<double>(0.0);
    }
    if (id == 1) {
        sleep(100);
        auto v1 = get_variable<double>(1.1);
        auto v2 = get_variable<int>(1);
        auto v3 = get_variable<int>(1);
    }
[...]
```

Listing A.1: The code for the example.

The nodes request the same number of variables for each type, but in a different order. (The `sleep` functions are to emphasize the order of execution.) What happens is that the two nodes will call the same shared object with different names.

1. Node 0 requests a variable of type `int`. This is request number 1 for this node and this type, but globally the system has reserved 0 containers for type `int`. The system thus creates a container for an `int` variable: this container will reserve space for two `int`s. The system then sets the first `int` to 0 (as requested by Node 0) and returns a `bsp_variable<int>`, which is an handle to the aforementioned container and in particular to the element assigned to Node 0. This `bsp_variable<int>` has the name `v1` in the environment of Node 0.

2. Node 1 requests a variable of type `double`. The system creates a new container of `double` elements, sets the element corresponding to Node 1 to 1.1, and returns the handle to this element to Node 1 as a `bsp_variable<double>` named `v1`. Note that Node 0's `v1` and Node 1's `v1` point to two different containers of different types.

3. Node 1 requests a variable of type `int`. This is the first request done by Node 1 for such a variable, and globally the system has already reserved a container of type `int`. No new containers are created: the second `int` of the existing container is set to 1, as requested by Node 1, and a `bsp_variable<int>` which refers to this element is returned to Node 1. This `bsp_variable<int>` has the name `v2` in the environment of Node 1, but it actually refers to the same container as Node 0's `v1`! If Node 0 were to call `v1.bsp_put(5, 1)` (see next subsection), this operation would actually change the element referred by Node 1 with the name `v2` after a superstep sync (i.e. `v2.get()` would return 5 after a superstep sync).

4. Node 1 requests a variable of type int. This is the second request for such a variable, but globally there's only one int container, so the system creates a new one. Node 1's handle for this container is a bsp_variable<int> named v3.

5. Node 0 requests a variable of type int. It's the second request for such a variable and there are already two containers of type int, so the system returns a handle to the second container, which is a bsp_variable<int> named v2.

6. Node 0 requests a variable of type double. It's the first request for such a variable and there's already one container of type double, so the system returns a handle to this container, which is a bsp_variable<double> named v3. Node 0's v3 and Node 1's v1 refer to the same container.

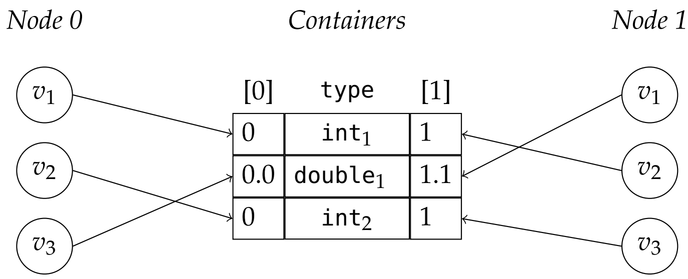


Figure A.1: An example of the difference between bsp_variables and their underlying containers.

A.3 A note on BSPunsafe methods

Both bsp_variable and bsp_array classes feature a few methods, identified by the prefix BSPunsafe, that allows the user to directly access the node's private copy of the shared object. Another way to gaining "direct access" to a bsp_array is by requesting it via the get_array<T>(pointer) method; the vector pointed by pointer is the same entity as the element in the container. This type of access to data is "BSP-unsafe" in the sense that it doesn't fit well with BSP directives (get and put). For example, if node n modifies its element with BSPunsafe_access() during superstep k and another node m performs a bsp_put() on the same element in the same superstep, at the beginning of superstep $k+1$ the value held in n's element will be the one written by m (as bsp_put() writes are performed at the end of the superstep, overwriting any modification previously made). If the data will be used in a read-only way, it's advisable to get a copy using the bsp_direct_get() methods.

The BSP-unsafe methods can nevertheless be useful when properly managed; some of the test programs (i.e. the Fast Fourier Transform and LU decomposition) use them for easier access to private copies, saving some superstep synchronizations. Other BSP implementations (e.g. MulticoreBSP for Java) also provide similar methods.

Appendix B

Source code

B.1 BSP Library for FastFlow

bsp_array.hpp

```
#ifndef FF_BSP_BSP_ARRAY_HPP
#define FF_BSP_BSP_ARRAY_HPP

#include <type_traits>
#include <vector>
#include "bsp_variable.hpp"

/**
 * An array structure, private to a single worker but with support
 * for communication with other similar entities.
 *
 * @tparam T type of the elements contained in the array.
 */
template<typename T>
class bsp_array : bsp_container {

    // The template type must be copy-constructible and copy-assignable
    static_assert(std::is_copy_constructible<T>::value &&
                  std::is_copy_assignable<T>::value,
                  "Type of elements of bsp_array must be copy-constructible "
                  " and copy-assignable");

private:

    // bsp_communicator can access private fields of this class
    friend class bsp_communicator;

    /**
     * Pointer to the actual vector data.
     */
    std::vector<T>* arr;

    /**
     * Default constructor.
     */
    bsp_array() = default;

    /**
```

```
     * Builds a bsp_array object that holds a node's private copy of a shared
     * array.
     *
     * @param _ref ID of the shared array
     * @param _hold ID of the requesting node
     * @param comm pointer to the communicator component
     * @param ptr pointer to the node's private copy
     */
    bsp_array(int _ref, int _hold, bsp_communicator* comm,
              std::vector<T>* ptr) :
            bsp_container(_ref, _hold, comm), arr{ptr} {
    }

    /**
     * Returns the shared object type (in this case, an array).
     * @return the \c vartype value for arrays
     */
    vartype var_type() final {
        return vartype::array;
    }

public:

    /**
     * Returns a copy of the element in position <tt>position</tt> of the
     * node's private copy of the shared array.
     * @param position position of the desired element
     * @return a copy the array element in the desired position
     */
    T get(int position) {
        return arr->at(position);
    }

    /**
     * Returns a copy of the node's private copy of the shared array.
     * @return a copy of the array
     */
    std::vector<T> get() {
        return *arr;
    }

    /**
     * Returns the size of the array.
     * @return the size of the array
     */
    int size() {
        return arr->size();
    }

    /**
     * Puts an element into the contained array in the desired
     * position. This function is <b>BSP unsafe</b>.
     * @param elem element to be copied
     * @param pos position in the array
     */
    void BSPunsafe_put(const T& elem, int pos) {
        arr->at(pos) = elem;
    }
```

```

    /**
     * Puts an element into the given position in this node's private copy of
     * the array.
     * @param elem element to be inserted
     * @param pos position in the array
     */
    void bsp_put(const T& elem, int pos) {
        bsp_put(elem, holder, pos);
    }

    /**
     * Puts an element (contained in a <tt>>bsp_variable</tt>) into the given
     * position in this node's private copy of the array.
     * @param elem \c bsp_variable that contains the element to be inserted
     * @param pos position in the array
     */
    void bsp_put(const bsp_variable<T>& elem, int pos) {
        const auto& t = *(elem.element);
        bsp_put(t, pos);
    }

    /**
     * Puts an element into the given position in another node's private copy of
     * the array.
     * @param elem element to be inserted
     * @param destination ID of the destination node
     * @param pos position in the array
     */
    void bsp_put(const T& elem, int destination, int pos) {
        comm->array_put(reference, holder, destination, pos, elem);
    }

    /**
     * Puts an element (contained in a <tt>>bsp_variable</tt>) into the given
     * position in another node's private copy of the array.
     * @param elem \c bsp_variable that contains the element to be inserted
     * @param destination ID of the destination node
     * @param pos position in the array
     */
    void bsp_put(const bsp_variable<T>& elem, int destination, int pos) {
        const auto& t = *(elem.element);
        bsp_put(t, destination, pos);
    }

    /**
     * Replaces the node's private copy of the array with the given vector.
     * @param array vector that will replace the current private copy of the
     * array
     */
    void bsp_put(const std::vector<T>& array) {
        comm->array_put(reference, holder, 0, holder, 0, 0, array);
    }

    /**
     * Replaces the node's private copy of the array with the vector
     * contained inside another <tt>bsp_array</tt>.
     * @param other <tt>bsp_array</tt> containing the vector that will
```

```
     * replace the current one
     */
    void bsp_put(const bsp_array<T>& other) {
        const auto& t = *(other.arr);
        bsp_put(t);
    }

    /**
     * Replaces another node's private copy of the array with the given vector.
     * @param array vector that will replace the current private copy of the
     * array
     * @param destination ID of the destination node
     */
    void bsp_put(const std::vector<T>& array, int destination) {
        comm->array_put(reference, holder, 0, destination, 0, 0, array);
    }

    /**
     * Replaces another node's private copy of the array with the vector
     * contained inside the given <tt>bsp_array</tt>.
     * @param other <tt>bsp_array</tt> containing the vector that will
     * replace the current private copy of the array
     * @param destination ID of the destination node
     */
    void bsp_put(const bsp_array<T>& other, int destination) {
        const auto& t = *(other.arr);
        bsp_put(t, destination);
    }

    /**
     * Replaces a portion of this node's private copy of the array with a
     * portion of the given vector.
     * @param array vector that contains the desired portion
     * @param src_offset starting position of the replacing portion
     * @param dest_offset starting position of the portion to be replaced
     * @param length length of the replacing portion
     */
    void bsp_put(const std::vector<T>& array, int src_offset, int dest_offset,
                 int length) {
        comm->array_put(reference, holder, src_offset, holder, dest_offset,
                        length, array);
    }

    /**
     * Replaces a portion of this node's private copy of the array with a
     * portion of the vector contained inside the given <tt>bsp_array</tt>.
     * @param other <tt>bsp_array</tt> containing the vector that contains
     * the desired portion
     * @param src_offset starting position of the replacing portion
     * @param dest_offset starting position of the portion to be replaced
     * @param length length of the replacing portion
     */
    void bsp_put(const bsp_array<T>& other, int src_offset, int dest_offset,
                 int length) {
        const auto& t = *(other.arr);
        bsp_put(t, src_offset, dest_offset, length);
    }
```

```
    /**
     * Replaces a portion of another node's private copy of the array with a
     * portion of the given vector.
     * @param array vector that contains the desired portion
     * @param destination ID of the destination node
     * @param src_offset starting position of the replacing portion
     * @param dest_offset starting position of the portion to be replaced
     * @param length length of the replacing portion
     */
    void bsp_put(const std::vector<T>& array, int destination, int src_offset,
                 int dest_offset, int length) {
        comm->array_put(reference, holder, src_offset, destination, dest_offset,
                        length, array);
    }

    /**
     * Replaces a portion of another node's private copy of the array with a
     * portion of the vector contained inside the given <tt>bsp_array</tt>.
     * @param other <tt>bsp_array</tt> containing the vector that contains
     * the desired portion
     * @param destination ID of the destination node
     * @param src_offset starting position of the replacing portion
     * @param dest_offset starting position of the portion to be replaced
     * @param length length of the replacing portion
     */
    void bsp_put(const bsp_array<T>& other, int destination, int src_offset,
                 int dest_offset, int length) {
        const auto& t = *(other.arr);
        bsp_put(t, destination, src_offset, dest_offset, length);
    }

    /**
     * Replaces the node's private copy of the array with another node's
     * private copy.
     * @param from ID of the source node
     */
    void bsp_get(int from) override {
        comm->array_get<T>(reference, from, 0, holder, 0, 0);
    }

    /**
     * Replaces a portion of the node's private copy of the array with a
     * portion of another node's private copy.
     * @param from ID of the source node
     * @param from_offset starting position of the replacing portion
     * @param to_offset starting position of the portion to be replaced
     * @param length length of the replacing portion
     */
    void bsp_get(int from, int from_offset, int to_offset, int length) {
        comm->array_get<T>(reference, from, from_offset, holder, to_offset,
                           length);
    }

    /**
     * Returns a copy of the element at the desired position from another node's
     * copy of the array. This method will return immediately, without
     * waiting for a superstep sync.
     * @param source ID of the source node
```

```
     * @param pos position of the desired element
     * @return the element of the source's private copy at the desired position
     */
    T bsp_direct_get(int source, int pos) {
        return comm->array_direct_get<T>(reference, source, pos);
    }

    /**
     * Returns a copy of another node's private array. This method will return
     * immediately, without waiting for a superstep sync.
     * @param source ID of the source node
     * @return the source's private copy of the shared array
     */
    std::vector<T> bsp_direct_get(int source) {
        return comm->array_direct_get<T>(reference, source);
    }

    /**
     * Returns a handle to this node's private copy of the shared array. The
     * returned object can be modified at will, without waiting for superstep
     * syncs.
     * This function is <b>BSP unsafe</b>.
     * @return a reference to the node's private copy of the array
     */
    std::vector<T>& BSPunsafe_access() {
        return *arr;
    }

};


#endif //FF_BSP_BSP_ARRAY_HPP
```

bsp_barrier.hpp

```
#ifndef FF_BSP_BSP_BARRIER_HPP
#define FF_BSP_BSP_BARRIER_HPP

#include <mutex>

/**
 * A simple reusable barrier.
 */
class bsp_barrier {
private:

    //! Mutex for access to the barrier
    std::mutex mutex;
    //! Condition variable for the wait/notify mechanism
    std::condition_variable cond_var;
    //! The number of workers that need to access the barrier before
    //! unlocking it.
    int threshold = 0;
    //! Current count of workers at the barrier
    int count = 0;
    //! Number of times the barrier has been used
    int generation = 0;
```

```
public:

    /**
     * Deleted default constructor.
     */
    bsp_barrier() = delete;

    /**
     * Creates a barrier for a certain number of threads.
     * @param size the number of threads that will access the barrier
     */
    explicit bsp_barrier(int size) :
            threshold{size},
            count{size},
            generation{0} {
    }

    /**
     * Copy constructor.
     * @param other the other <tt>bsp_barrier</tt> object to copy.
     */
    bsp_barrier(const bsp_barrier& other) :
            threshold{other.threshold},
            count{other.count},
            generation{other.generation} {
    }

    /**
     * Waits on the barrier until all peers do the same.
     */
    void wait() {
        std::unique_lock<std::mutex> lock{mutex};
        auto lastgen = generation;
        if (!(--count)) {
            generation++;
            count = threshold;
            cond_var.notify_all();
        } else {
            cond_var.wait(lock,
                          [this, lastgen]() { return lastgen != generation; });
        }
    }
};

#endif //FF_BSP_BSP_BARRIER_HPP
```

bsp_communicator.hpp

```
#ifndef FF_BSP_BSP_COMMUNICATOR_HPP
#define FF_BSP_BSP_COMMUNICATOR_HPP

#include <set>
#include <iostream>
#include "bsp_array.hpp"

#ifdef STL_ALLOC
```

```cpp
#define m_shared_ptr(U, what) \
    std::allocate_shared<U, std::llocator<U>> (std::allocator<U>(), (what))
#define alloc(U) std::allocator<U>
#else
#define m_shared_ptr(U, what) \
    std::allocate_shared<U, ff::FF_Allocator<U>> (ff::FF_Allocator<U>(), (what))
#define alloc(U) ff::FF_Allocator<U>
#endif

/**
 * Implementation of the communicator entity.
 */

// Requests management

/**
 * Inserts a request for replacing a node's private copy of a shared variable
 * with the provided value.
 * @tparam T the type of the variable
 * @param what ID of the variable to be modified
 * @param source ID of the node that performs the request
 * @param destination ID of the node that holds the private copy of the
 * variable that will be modified
 * @param element the value to be copied inside the destination
 */
template<typename T>
void bsp_communicator::variable_put(int what, int source, int destination,
                                    const T& element) {
    if (what == 0) return;
    mutexes[destination].lock();
    requests[destination].emplace_back(request_type::var_put, what, source,
                                       destination, 0, 0, 0,
                                       m_shared_ptr(T, element));
    mutexes[destination].unlock();
}

/**
 * Inserts a request for replacing a node's private copy of a shared variable
 * with the requestor's private value of the same shared variable.
 * @tparam T the type of the variable
 * @param what ID of the variable to be modified
 * @param source ID of the node that performs the request
 * @param dest ID of the node that holds the private copy of the
 * variable that will be modified
 */
template<typename T>
void bsp_communicator::variable_put(int what, int source, int dest) {
    if (what == 0) return;
    mutexes[dest].lock();
    auto elem = (static_cast<T*>(variables_storage.at(what).element[source]));
    requests[dest].emplace_back(request_type::var_put, what, source, dest, 0, 0,
                                0, m_shared_ptr(T, *elem));
    mutexes[dest].unlock();
}

/**
 * Inserts a request for replacing an element of a node's private copy of a
 * shared array with the provided value.
```

```
 * @tparam T the type of the elements of the array
 * @param what ID of the array to be modified
 * @param source ID of the node that performs the request
 * @param dest ID of the node that holds the private copy of the
 * array that will be modified
 * @param pos position of the element to be modified
 * @param elem the value to be copied inside the destination
 */
template<typename T>
void bsp_communicator::array_put(int what, int source, int dest, int pos,
                                 const T& elem) {
    mutexes[dest].lock();
    requests[dest].emplace_back(request_type::arr_put_el, what, source, dest, 0,
                                pos, 0, m_shared_ptr(T, elem));
    mutexes[dest].unlock();
}

/**
 * Inserts a request for replacing a portion of a node's private copy of a
 * shared array with a portion of the provided array.
 * @tparam T the type of the elements of the arrays
 * @param what ID of the array to be modified
 * @param src ID of the node that performs the request
 * @param src_off starting position of the portion of the replacing array
 * @param dest ID of the node that holds the private copy of the
 * array that will be modified
 * @param dest_off starting position of the portion to be replaced
 * @param len length of the portion of the replacing array
 * @param v the replacing array
 */
template<typename T>
void bsp_communicator::array_put(int what, int src, int src_off, int dest,
                                 int dest_off, int len,
                                 const std::vector<T>& v) {
    mutexes[dest].lock();
    requests[dest].emplace_back(request_type::arr_put, what, src, dest,
                                src_off, dest_off, len,
                                m_shared_ptr(std::vector<T>, v));
    mutexes[dest].unlock();
}

/**
 * Inserts a request for replacing a portion of a node's private copy of a
 * shared array with a portion of the private copy of the requesting node
 * @tparam T the type of the elements of the arrays
 * @param what ID of the array to be modified
 * @param src ID of the node that performs the request
 * @param src_off starting position of the portion of the requesting node's
 * private copy of the array
 * @param dest ID of the node that holds the private copy of the
 * array that will be modified
 * @param dest_off starting position of the portion to be replaced
 * @param len length of the portion of the replacing array
 */
template<typename T>
void bsp_communicator::array_get(int what, int from, int from_off, int to,
                                 int to_off, int len) {
    mutexes[to].lock();
```

```
    auto arr = static_cast<std::vector<T>*>(arrays_storage.at(
            what).element[from]);
    requests[to].emplace_back(request_type::arr_get, what, from, to, from_off,
                              to_off, len, m_shared_ptr(std::vector<T>, *arr));
    mutexes[to].unlock();
}

/**
 * Returns the value of a node's private copy of a shared variable,
 * without waiting for a superstep sync.
 * @tparam T the type of the variable
 * @param what ID of the variable to be queried
 * @param source ID of the node that holds the requested private copy
 * @return a copy of the desired value
 */
template<typename T>
T bsp_communicator::variable_direct_get(int what, int source) {
    if (what == 0) return fastflow_input;
    return *(static_cast<T*>(variables_storage.at(what).element.at(source)));
}

/**
 * Returns the value of an element of a node's private copy of a shared array,
 * without waiting for a superstep sync.
 * @tparam T the type of the elements of the array
 * @param what ID of the variable to be queried
 * @param src ID of the node that holds the requested private copy
 * @param pos position of the desired element
 * @return a copy of the desired value
 */
template<typename T>
T bsp_communicator::array_direct_get(int what, int src, int pos) {
    auto arptr = static_cast<std::vector<T>*>(arrays_storage.at(
            what).element.at(src));
    return arptr->at(pos);
}

/**
 * Returns a node's private copy of a shared array, without waiting for a
 * superstep sync.
 * @tparam T the type of the elements of the array
 * @param what ID of the variable to be queried
 * @param src ID of the node that holds the requested private copy
 * @return a copy of the desired array
 */
template<typename T>
std::vector<T> bsp_communicator::array_direct_get(int what, int src) {
    return *(static_cast<std::vector<T>*>(arrays_storage.at(what).element.at(
            src)));
}

/**
 * Requests the handle to a shared variable of type \c T. Will create a new
 * shared variable if the requesting node already has the handle for all
 * already-present shared variables of type T. Sets the private copy of the
 * returned shared variable to an initial value.
 * @tparam T the type of the requested shared variable
 * @param holder ID of the node that requests the handle
```

```
 * @param initial_val pointer to the value to be copied inside the requesting
 * node's private copy
 * @return a handle of the shared variable, wrapped in a \c bsp_variable.
 */
template<typename T>
bsp_variable<T> bsp_communicator::get_variable(int holder, T* initial_val) {
    auto tname = typeid(T).name();
    // no. of variables of type T requested by the current worker
    int get_count = 0;
    try {
        get_count = var_count[holder].at(std::string(tname));
    } catch (const std::out_of_range&) {
        var_count[holder].insert({std::string(tname), 0});
    }
    // hash on type, superstep and number of vars
    int hash = get_hash(typeid(T).name(), (generation * 5000000) + get_count);
    int ref;
    // try to find a variable w/ the same hash
    //(i.e. another worker has already requested the creation of the var)
    try {
        // multiple readers-single writer pattern
        std::shared_lock lock(var_mutex);
        ref = variable_dict.at(hash);
    } catch (const std::out_of_range&) {
        std::unique_lock lock(var_mutex);
        // try again, in case another thread created the variable
        // while this one waited
        auto iter = variable_dict.find(hash);
        // if the variable is still not present...
        if (iter == variable_dict.end()) {
            // create the variable...
            inner_var var{nprocs, [](void* el, void* other) {
                auto tptr = static_cast<T*>(el);
                *tptr = *(static_cast<T*>(other));
            }, [](void* el) {
                auto tptr = static_cast<T*>(el);
                delete tptr;
            }};
            ref = variables_storage.size();
            // ..and store it
            variables_storage.push_back(var);
            // together with its hash
            variable_dict.insert({hash, ref});
        } else {
            // the variable is present
            ref = iter->second;
        }
    }
    // initialize with the value requested by the worker
    variables_storage[ref].element.at(holder) = initial_val;
    var_count[holder].at(std::string(tname))++;
    return bsp_variable<T>{ref, holder, this, initial_val};
}

/**
 * Requests the handle to a shared array with elements of type \c T. Will
 * create a new shared array if the requesting node already has the handle for
 * all already-present shared arrayss of type T. Sets the private copy of the
```

```cpp
 * returned shared array to an initial value.
 * @tparam T the type of the requested shared variable
 * @param holder ID of the node that requests the handle
 * @param initial_arr pointer to the value to be copied inside the requesting
 * node's private copy
 * @param to_delete flag that indicates whether the array can be deleted
 * safely when perfoming cleanup at the end of the computation
 * @return a handle of the shared array, wrapped in a \c bsp_array.
 */
template<typename T>
bsp_array<T>
bsp_communicator::get_array(int holder, std::vector<T>* initial_arr,
                            bool to_delete) {
    auto tname = typeid(T).name();
    int get_count = 0;
    try {
        get_count = arr_count[holder].at(std::string(tname));
    } catch (const std::out_of_range&) {
        arr_count[holder].insert({std::string(tname), 0});
    }
    int hash = get_hash(typeid(T).name(), (generation * 5000000) + get_count);
    int ref;
    try {
        std::shared_lock lock(arr_mutex);
        ref = array_dict.at(hash);
    } catch (const std::out_of_range&) {
        std::unique_lock lock(arr_mutex);
        auto iter = array_dict.find(hash);
        if (iter == array_dict.end()) {
            inner_array arr{nprocs, [](void* el, void* other, int pos) {
                auto arrptr = static_cast<std::vector<T>*>(el);
                arrptr->at(pos) = *(static_cast<T*>(other));
            }, [](void* el, int srcof, int dstof, int size, void* toput) {
                auto arrptr = static_cast<std::vector<T>*>(el);
                auto otherptr = static_cast<std::vector<T, alloc(T)>*>(toput);
                if (size == 0) {
                    arrptr->resize(otherptr->size());
                    for (size_t idx{0}; idx < otherptr->size(); ++idx) {
                        arrptr->at(idx) = otherptr->at(idx);
                    }
                } else {
                    std::copy_n(otherptr->begin() + srcof, size,
                                arrptr->begin() + dstof);
                }
            }, to_delete ? [](void* el) {
                auto tptr = static_cast<std::vector<T>*>(el);
                delete tptr;
            } : [](void*) {}};
            ref = arrays_storage.size();
            arrays_storage.push_back(arr);
            array_dict.insert({hash, ref});
        } else {
            ref = iter->second;
        }
    }
    arrays_storage[ref].element.at(holder) = initial_arr;
    arr_count[holder].at(std::string(tname))++;
    return bsp_array<T>{ref, holder, this, initial_arr};
```

```
}

/**
 * During the superstep synchronization, processes all requests with a
 * certain node as destination. The last node to finish this operation will also
 * advance the superstep count.
 * @param id ID of the node that will process the requests.
 */
void bsp_communicator::process_requests(int id) {
    // Request filtering by worker id
    for (const auto& req: requests[id]) {
        switch (req.t) {
            case request_type::var_put: {
                auto ptr = variables_storage.at(
                        req.reference).element[req.destination];
                variables_storage.at(req.reference).swap(ptr,
                                                         req.element.get());
                break;
            }
            case request_type::arr_put_el: {
                auto ptr = arrays_storage.at(
                        req.reference).element[req.destination];
                arrays_storage.at(req.reference).put(ptr, req.element.get(),
                                                     req.dest_offset);
                break;
            }
            case request_type::arr_put: {
                auto ptr = arrays_storage.at(
                        req.reference).element[req.destination];
                arrays_storage.at(req.reference).replace(ptr, req.src_offset,
                                                         req.dest_offset,
                                                         req.length,
                                                         req.element.get());
                break;
            }
            case request_type::arr_get: {
                auto ptr = arrays_storage.at(
                        req.reference).element[req.destination];
                arrays_storage.at(req.reference).replace(ptr, req.src_offset,
                                                         req.dest_offset,
                                                         req.length,
                                                         req.element.get());
                break;
            }
        }
    }
    // The last worker to finish managing its requests will advance the
    // computation by increasing the superstep count and resetting all the
    // superstep-specific data structures used by the communicator
    if (++process_count == nprocs) {
        generation++;
        delete[] arr_count;
        arr_count = new std::map<std::string, int>[nprocs]();
        delete[] var_count;
        var_count = new std::map<std::string, int>[nprocs]();
        delete[] requests;
        requests = new std::vector<request,alloc(request)>[nprocs]();
        process_count = 0;
```

```
    }
}

/**
 * Makes the FastFlow input token available to all BSP nodes.
 * @param in pointer to the input token
 */
void bsp_communicator::set_fastflow_input(const void* in) {
    fastflow_input = in;
}

/**
 * Returns the pointer to the FastFlow input token.
 * @return a \c const pointer to the FastFlow input token.
 */
const void* bsp_communicator::get_fastflow_input() {
    return fastflow_input;
}

/**
 * Deallocates all the data structures used in the communicator and all shared
 * arrays and variables.
 */
void bsp_communicator::end() {

    for (auto& var: variables_storage) {
        auto del = var.free_el;
        for (auto& el: var.element) {
            del(el);
        }
    }

    for (auto& arr: arrays_storage) {
        auto del = arr.free_el;
        for (auto& el: arr.element) {
            del(el);
        }
    }

    delete[] arr_count;
    arr_count = nullptr;
    delete[] var_count;
    var_count = nullptr;
    delete[] requests;
    requests = nullptr;
    delete[] mutexes;
    mutexes = nullptr;
}

#endif //FF_BSP_BSP_COMMUNICATOR_HPP
```

bsp_internals.hpp

```
#ifndef FF_BSP_INTERNALS_HPP
#define FF_BSP_INTERNALS_HPP

#include <utility>
```

```cpp
#include <vector>
#include <memory>
#include <shared_mutex>
#include <map>
#include <algorithm>
#include <atomic>
#include "stl_allocator.hpp"

#ifdef STL_ALLOC
    #define alloc(T) std::allocator<T>
#else
    #define alloc(T) ff::FF_Allocator<T>
#endif

/**
 * Contains definitions and class prototypes used by the communicator and
 * variable/array classes
 */

/**
 * Represents the type of a shared object (variable or array)
 */
typedef enum {
    array,
    variable
} vartype;

// Forward declaration
class bsp_communicator;

/**
 * Base class for BSP variables and arrays, i.e. private copies of shared
 * objects.
 */
class bsp_container {
protected:
    //! ID of the shared object
    int reference;
    //! ID of the node that owns this private copy
    int holder;
    //! Pointer to the communicator object
    bsp_communicator* comm;

    //! Method that will return the object's type (variable or array)
    virtual vartype var_type() = 0;

    /**
     * Base constructor for the class.
     * @param _reference ID of the shared object
     * @param holder_pid ID of the node that owns this private copy
     * @param _comm Pointer to the communicator object
     */
    bsp_container(int _reference, int holder_pid, bsp_communicator* _comm) :
            reference{_reference}, holder{holder_pid}, comm{_comm} {};

public:

    virtual void bsp_get(int) = 0;
```

```cpp

};

// Forward declaration
template<typename T>
class bsp_variable;
// Forward declaration
template<typename T>
class bsp_array;

// Class declaration for the communicator
class bsp_communicator {
private:
    /**
     * Represent the possible types for a request object.
     */
    enum request_type {
        var_put,
        arr_put_el,
        arr_put,
        arr_get
    };

    /**
     * Representation of a put/get request in the communicator.
     */
    struct request {
        request_type t;
        int reference;
        int source;
        int destination;
        int src_offset;
        int dest_offset;
        int length;
        std::shared_ptr<void> element;

        request(request_type _t, int _ref, int _src, int _dest,
                int _srcof, int _dstof, int _len, std::shared_ptr<void> _el) :
                t{_t}, reference{_ref}, source{_src}, destination{_dest},
                src_offset{_srcof}, dest_offset{_dstof}, length{_len},
                element{std::move(_el)} {};
    };

    /**
     * Shared variable object in the shared memory.
     */
    struct inner_var {
        //! Vector of private copies of the variable
        std::vector<void*, alloc(void*)> element;
        //! Bookkeeping function needed to work with <tt>void*</tt>
        //! Replaces a variable with another value
        void (* swap)(void* el,
                      void* other);
        //! Bookkeeping function needed to work with <tt>void*</tt>
        //! Safely frees memory occupied by an element
        void (* free_el)(
                void* el);

```

```
    /**
     * Constructor for the shared variable object.
     * @param nprocs number of BSP nodes in the computation
     * @param swapfun pointer to function for element swapping
     * @param free_elfun pointer to function for safe delete
     */
    inner_var(int nprocs, void (* swapfun)(void*, void*),
              void (* free_elfun)(void*)) : swap{swapfun},
                                            free_el{free_elfun} {
        element.resize(nprocs);
    }
};

/**
 * Shared array object in the shared memory.
 */
struct inner_array {
    //! Vector of private copies of the array
    std::vector<void*, alloc(void*)> element;
    //! Bookkeeping function needed to work with <tt>void*</tt>
    //! Replaces an element of the a private copy of the array with another
    //! value
    void (* put)(void* el, void* toput,
                 int pos);
    //! Bookkeeping function needed to work with <tt>void*</tt>
    //! Replaces a portion of a private copy of the array
    void (* replace)(void* el, int srcof, int dstof, int len,
                     void* toput);
    //! Bookkeeping function needed to work with <tt>void*</tt>
    //! Safely frees memory occupied by an element
    void (* free_el)(
            void* el);

    /**
     * Constructor for the shared variable object.
     * @param nprocs number of BSP nodes in the computation
     * @param putfun pointer to function for element swapping
     * @param replacefun pointer to function for replacing a portion
     * @param free_elfun pointer to function for safe delete
     */
    inner_array(int nprocs, void (* putfun)(void*, void*, int),
                void (* replacefun)(void*, int, int, int, void*),
                void (* free_elfun)(void*)) :
            put{putfun}, replace{replacefun}, free_el{free_elfun} {
        element.resize(nprocs);
    }
};

/**
 * Hashes a string and a number according to the dbj2 hash.
 * @param s the string to be hashed
 * @param seed an initial value for the hashing function
 * @return a (hopefully) unique number that represents the input pair
 */
static int get_hash(const char* s, int seed) {
    unsigned int hash = seed + 5381;
    while (*s) {
        hash = hash * 33 ^ (*s++);
```

```
        }
        return hash;
    }

    //! Number of BSP nodes
    int nprocs;
    //! Number of current superstep
    int generation = 1;

    //! Counts the variables requested by each node in this superstep
    std::map<std::string, int>* var_count;
    //! Counts the arrays requested by each node in this superstep
    std::map<std::string, int>* arr_count;

    //! Mutex for multiple readers-single writer access to the variables
    mutable std::shared_mutex var_mutex;
    //! Mutex for multiple readers-single writer access to the arrays
    mutable std::shared_mutex arr_mutex;

    //! Dictionary for quick retrieving of variables based on their hash
    std::map<int, int> variable_dict;
    //! Dictionary for quick retrieving of arrays based on their hash
    std::map<int, int> array_dict;

    //! Container that stores variables
    std::vector<inner_var> variables_storage;
    //! Container that stores arrays
    std::vector<inner_array> arrays_storage;

    //! Array of mutexes to access request queues
    std::mutex* mutexes;
    //! Requests of variable/array modifications submitted during this superstep
    std::vector<request, alloc(request)>* requests;

    //! Counts the nodes that requested a sync during this superstep
    std::atomic_int process_count{0};

    //! Pointer to the input token received by the FastFlow node
    const void* fastflow_input = nullptr;

public:

    /**
     * Constructor for the communicator.
     * @param _nprocs number of BSP nodes in this computation.
     */
    explicit bsp_communicator(int _nprocs) : nprocs{_nprocs} {
        var_count = new std::map<std::string, int>[_nprocs]();
        arr_count = new std::map<std::string, int>[_nprocs]();
        mutexes = new std::mutex[_nprocs]();
        requests = new std::vector<request, alloc(request)>[_nprocs]();
        inner_var invar{nprocs, [](void*, void*) {}, [](void*) {}};
        variables_storage.push_back(invar);
    };

    // Refer to bsp_communicator.hpp

    void set_fastflow_input(const void*);
```

```
    const void* get_fastflow_input();

    template<typename T>
    void variable_put(int, int, int, const T&);

    template<typename T>
    void variable_put(int, int, int);

    template<typename T>
    T variable_direct_get(int, int);

    template<typename T>
    void array_put(int, int, int, int, const T&);

    template<typename T>
    void array_put(int, int, int, int, int, int, const std::vector<T>&);

    template<typename T>
    void array_get(int, int, int, int, int, int);

    template<typename T>
    T array_direct_get(int, int, int);

    template<typename T>
    std::vector<T> array_direct_get(int, int);

    template<typename T>
    bsp_variable<T> get_variable(int holder, T* initial_val);

    template<typename T>
    bsp_array<T>
    get_array(int holder, std::vector<T>* initial_arr, bool to_delete = true);

    void process_requests(int id);

    void end();

};

#endif //FF_BSP_BSP_VARIABLE_INTERNAL_HPP
```

bsp_node.hpp

```
#ifndef FF_BSP_BSP_NODE_HPP
#define FF_BSP_BSP_NODE_HPP

#include <ff/ff.hpp>
#include "bsp_communicator.hpp"
#include "bsp_barrier.hpp"


/**
 * Specialization of a ff_node to work as the unit of computation in the BSP
 * model.
 */
class bsp_node: public ff::ff_node {
```

```cpp
private:

    //! Number of BSP nodes in this computation
    int nprocs;
    //! ID for this node
    int id = -1;
    //! Pointer to the communicator; implementation of the Communication Channel
    bsp_communicator* comm;
    //! Pointer to the barrier; implementation of the Synchronization Channel
    bsp_barrier* barrier;

    // bsp_program can access fields of this class
    friend class bsp_program;

protected:

    //! Pointer to the FastFlow input token
    const void* fastflow_input;

    /**
     * Forwards an output token to the next stage in the FastFlow graph.
     * @param payload the token to forward
     */
    void emit_output(void* payload) {
        ff_send_out(payload);
    }

    /**
     * Requests a new variable of type T from the communicator.
     * @tparam T the type of the requested variable
     * @param initial_value value to copy inside this node's private copy of
     * the variable
     * @return a handle to this node's private copy of the shared variable
     */
    template <typename T>
    bsp_variable<T> get_variable(const T& initial_value) {
        T* val = new T(initial_value);
        return comm->get_variable<T>(id, val);
    }

    /**
     * Requests a new array with elements of type T from the communicator,
     * initializing it with the copy a given vector.
     * @tparam T the type of elements of the requested array
     * @param initial_value value to copy inside this node's private copy of
     * the array
     * @return a handle to this node's private copy of the shared variable
     */
    template <typename T>
    bsp_array<T> get_array(const std::vector<T>& initial_value) {
        auto val = new std::vector<T>(initial_value);
        return comm->get_array(id, val);
    }

    /**
     * Requests a new array with elements of type T from the communicator,
     * initializing it with the pointer of a vector. Any modifications done
     * to the initializing vector after the call to this method is inherently
```

```
     * BSP unsafe.
     * @tparam T the type of elements of the requested array
     * @param handle a pointer to the vector that will become this node's
     * private copy of the shared array
     * @return a handle to this node's private copy of the shared variable
     */
    template <typename T>
    bsp_array<T> get_array(std::vector<T>* handle) {
        return comm->get_array(id, handle, false);
    }

    /**
     * Requests a new array with elements of type T from the communicator,
     * initializing it with an empty vector of given size.
     * @tparam T the type of elements of the requested array
     * @param size the size of the empty vector that will become this node's
     * private copy of the shared array
     * @return a handle to this node's private copy of the shared variable
     */
    template <typename T>
    bsp_array<T> get_empty_array(int size) {
        auto val = new std::vector<T>(size);
        return comm->get_array(id, val);
    }

    /**
     * Returns the ID for this node.
     * @return this node's ID.
     */
    int bsp_pid() {
        return id;
    }

    /**
     * Returns the number of nodes in the current BSP computation.
     * @return the number of nodes in the current BSP computation.
     */
    int bsp_nprocs() {
        return nprocs;
    }

    /**
     * Terminates the current superstep and waits for the other nodes to sync.
     */
    void bsp_sync() {
        barrier->wait();
        comm->process_requests(id);
        barrier->wait();
    }

    /**
     * Function to be overwritten as the main parallel execution.
     */
    virtual void parallel_function() = 0;

public:

    /**
```

```
     * The FastFlow node service method.
     *
     * Implementations of this class cannot redefine it.
     * @param in the input token (in this case, a special value \c ENDCOMP)
     * @return the same token as the input
     */
    void* svc (void* in) final {
        fastflow_input = comm->get_fastflow_input();
        parallel_function();
        return in;
    }
};

#endif //FF_BSP_BSP_NODE_HPP
```

bsp_program.hpp

```
#ifndef FF_BSP_BSP_PROGRAM_HPP
#define FF_BSP_BSP_PROGRAM_HPP

#include <ff/ff.hpp>
#include <memory>
#include <iostream>
#include "bsp_node.hpp"

/**
 *  Implements the Bulk Synchronous Parallel pattern as a FastFlow node.
 */
class bsp_program : public ff::ff_node {

private:

    /**
     * The emitter of the master-worker scheme; it allows the user to perform a
     * (sequential) function before the execution of the main parallel part.
     */
    struct emitter : ff::ff_node {
        //! Value to end the computation after every node finishes its job
        void* ENDCOMP = (void*) ((unsigned long long) ff::FF_TAG_MIN - 1);

        //! Optional function to be executed before the BSP computation
        std::function<void(void)> preprocessing;
        //! Number of BSP nodes
        int emitter_nprocs;

        /**
         * Constructor for the farm emitter.
         * @param pre optional function to be executed before the BSP
         * computation
         * @param size number of BSP nodes
         */
        emitter(std::function<void(void)> pre, int size) :
                preprocessing{std::move(pre)},
                emitter_nprocs{size} {
        }

        /**
```

```
     * Service function of the farm emitter.
     * @param in input FastFlow token
     * @return the End-of-Stream token
     */
    void* svc(void* in) override {
        if (preprocessing != nullptr) preprocessing();
        for (int i = 0; i < emitter_nprocs; i++) {
            ff_send_out(ENDCOMP);
        }
        return EOS;
    }
};

/**
 * The collectot of the master-worker scheme; it allows the user to perform
 * a (sequential) function after the execution of the main parallel part.
 */
struct collector : ff::ff_node {
    //! Value to end the computation after every node finishes its job
    void* ENDCOMP = (void*) ((unsigned long long) ff::FF_TAG_MIN - 1);
    //! Optional function to be executed after the BSP computation
    std::function<void(void)> postprocessing;
    //! Number of nodes in the BSP computation
    int threshold;
    //! Nodes that have finished their local computation
    int count;
    //! Pointer to the the outer class
    bsp_program* master;

    /**
     * Constructor for the farm collector.
     * @param post optional function to be executed after the BSP
     * computation
     * @param size number of BSP nodes
     */
    collector(std::function<void(void)> post, int size) :
            postprocessing{std::move(post)},
            threshold{size},
            count{0} {
    };

    /**
     * Service function of the farm emitter.
     * @param in input FastFlow token
     * @return the input token, or GO_ON
     */
    void* svc(void* in) override {
        if (in == ENDCOMP) {
            if (++count == threshold) {
                if (postprocessing != nullptr) postprocessing();
            }
            return GO_ON;
        } else {
            master->forward(in);
        }
        return in;
    }
};
```

```cpp

    /**
     * Forwards the received token to the succeeding FastFlow node.
     * @param token the token to be forwarded.
     */
    void forward(void* token) {
        ff_send_out(token);
    }

    //! Number of BSP nodes (workers of the FastFlow farm pattern)
    int nprocs;
    //! Vector of pointers to BSP nodes
    std::vector<std::unique_ptr<bsp_node>> processors;
    //! The entity responsible for communication between nodes
    bsp_communicator comm;
    //! A barrier for synchronization between supersteps
    bsp_barrier barr;
    //! The farm emitter
    emitter E;
    //! The farm collector
    collector C;

public:

    /**
     * Constructor for the bsp_program object. Pushes necessary information
     * into the relevant entities (emitter, workers, collector)
     * @param _processors vector of BSP nodes for the computation
     * @param _pre optional function to be executed before the BSP computation
     * @param _post optional function to be executed after the BSP computation
     */
    explicit bsp_program(std::vector<std::unique_ptr<bsp_node>>&& _processors,
                         std::function<void(void)> _pre = nullptr,
                         std::function<void(void)> _post = nullptr) :
            nprocs{static_cast<int>(_processors.size())},
            comm{nprocs},
            barr{nprocs},
            E{std::move(_pre), nprocs},
            processors{std::move(_processors)},
            C{std::move(_post), nprocs} {
        for (size_t i{0}; i < nprocs; ++i) {
            processors[i]->nprocs = nprocs;
            processors[i]->id = i;
            processors[i]->barrier = &barr;
            processors[i]->comm = &comm;
            C.master = this;
        }
    };

    /**
     * Creates the FastFlow inner graph and executes the BSP computation.
     * @param in optional, the input token received from the preceding FastFlow
     * node
     */
    void start(void* in = nullptr) {
        std::vector<std::unique_ptr<ff::ff_node>> workers;
        for (size_t i{0}; i < nprocs; ++i) {
            auto d = static_cast<ff_node*>(processors[i].release());
```

```
            workers.emplace_back(std::unique_ptr<ff_node>(d));
        }

        comm.set_fastflow_input(in);

        ff::ff_Farm<> farm(std::move(workers), E, C);

        if (farm.run_and_wait_end() < 0)
            std::cout << "error in running farm" << std::endl;

        comm.end();
    }

    /**
     * Service function for the BSP program node. Starts the BSP computation.
     * @param in input token received from the preceding FastFlow node
     * @return GO_ON
     */
    void* svc(void* in) override {
        start(in);
        return GO_ON;
    }
};

#endif //FF_BSP_BSP_PROGRAM_HPP
```

bsp_variable.hpp

```
#ifndef FF_BSP_BSP_VARIABLE_HPP
#define FF_BSP_BSP_VARIABLE_HPP

#include <type_traits>
#include "bsp_internals.hpp"

/**
 * A variable data structure, private to a single worker but with support for
 * communication with other similar entities.
 *
 * @tparam T type of the variable
 */
template<typename T>
class bsp_variable : public bsp_container {

    // The template type must be copy-constructible and copy-assignable
    static_assert(std::is_copy_constructible<T>::value &&
                  std::is_copy_assignable<T>::value,
                  "Type of bsp_variable must be copy-constructible "
                  "and copy-assignable");

private:

    // bsp_array can access private fields of this class
    template<typename E>
    friend class bsp_array;

    // bsp_communicator can access private fields of this class
    friend class bsp_communicator;
```

```
    /**
     * Pointer to the actual data element.
     */
    T* element;

    /**
     * Default constructor.
     */
    bsp_variable() = default;

    /**
     * Builds a bsp_variable object that holds a node's private copy of a shared
     * variable.
     * @param _ref ID of the shared variable
     * @param _hold ID of the requesting node
     * @param comm pointer to the communicator component
     * @param ptr pointer to the node's private copy
     */
    explicit bsp_variable(int _ref, int _hold, bsp_communicator* comm, T* ptr) :
            bsp_container(_ref, _hold, comm), element{ptr} {
    }

    /**
     * Returns the shared object type (in this case, a variable).
     * @return the \c vartype value for variables
     */
    vartype var_type() final {
        return vartype::variable;
    }

public:

    /**
     * Returns a copy of the node's private element of the shared variable.
     * @return a copy of the desired element
     */
    T get() {
        return *element;
    }

    /**
     * Replaces another node's private element of the shared variable.
     * @param elem element to be copied
     * @param destination ID of the destination node
     */
    void bsp_put(const T& elem, int destination) {
        comm->variable_put<T>(reference, holder, destination, elem);
    }

    /**
     * Replaces this node's private element with this node's private element of
     * another shared variable.
     * @param other the handle to the other shared variable
     */
    void bsp_put(const bsp_variable<T>& other) {
        const auto& t = *(other.element);
        bsp_put(t, holder);
```

```
    }

    /**
     * Replaces another node's private element with this node's private
     * element of another shared variable.
     * @param other the handle to the other shared variable
     * @param destination ID of the destination node
     */
    void bsp_put(const bsp_variable<T>& other, int destination) {
        const auto& t = *(other.element);
        bsp_put(t, destination);
    }

    /**
     * Replaces another node's private element with this node's private
     * element of this shared variable.
     * @param destination ID of the destination node
     */
    void bsp_put(int destination) {
        comm->variable_put<T>(reference, holder, destination);
    }

    /**
     * Replaces this node's private element with another node's private
     * element of this shared variable.
     * @param source ID of the source node
     */
    void bsp_get(int source) override {
        comm->variable_put<T>(reference, source, holder);
    }

    /**
     * Returns a copy of a node's private element of this shared variable.
     * This method will return immediately, without waiting for a superstep
     * sync.
     * @param source ID of the source node
     * @return a copy of the desired element
     */
    T bsp_direct_get(int source) {
        return comm->variable_direct_get<T>(reference, source);
    }

    /**
     * Returns a handle to this node's private element of the shared variable.
     * The returned object can be modified at will, without waiting for
     * superstep syncs.
     * This function is <b>BSP unsafe</b>.
     * @return a reference to the node's private element of the shared variable
     */
    T& BSPunsafe_access() {
        return *element;
    }
};

#endif //FF_BSP_BSP_VARIABLE_HPP
```

stl_allocator.hpp

```cpp
#ifndef FF_BSP_STL_ALLOCATOR_HPP
#define FF_BSP_STL_ALLOCATOR_HPP

#include <ff/allocator.hpp>

namespace ff {

    /**
     * STL-compliant wrapper for FastFlow's custom allocator.
     * @tparam T the type of objects to be allocated or deallocated.
     */
    template<typename T>
    class FF_Allocator {
    public:
        using value_type = T;
        using propagate_on_container_move_assignment = std::true_type;
        using is_always_equal = std::true_type;

        /**
         * Default constructor.
         */
        FF_Allocator() noexcept = default;

        /**
         * Empty copy constructor.
         * @tparam U the type of objects of the other allocator object
         * @param other the other allocator object
         */
        template<class U>
        explicit FF_Allocator(const FF_Allocator<U>& other) noexcept {};

        /**
         * Allocates n * sizeof(T) bytes of uninitialized storage by calling
         * the FastFlow allocator instance's \c malloc function.
         * @param n the number of objects to allocate storage for
         * @return the pointer to the first newly-allocated object
         */
        value_type* allocate(std::size_t n) {
            return static_cast<value_type*>(FFAllocator::instance()
                    ->malloc(n * sizeof(value_type)));
        }

        /**
         * Deallocates the storage referenced by the pointer p, which must be
         * a pointer obtained by an earlier call to allocate().
         * @param ptr pointer to the object to deallocate
         */
        void deallocate(value_type* ptr, std::size_t) noexcept {
            FFAllocator::instance()->free(ptr);
        }
    };

    /**
     * Checks that two allocator objects can be considered the same.
     * @tparam T type of the first allocator
     * @tparam U type of the second allocator
```

```
     * @return true
     */
    template<class T, class U>
    bool
    operator==(const FF_Allocator<T>&, const FF_Allocator<U>&) { return true; }

    /**
     * Checks that two allocator objects can be considered different.
     * @tparam T type of the first allocator
     * @tparam U type of the second allocator
     * @return false
     */
    template<class T, class U>
    bool
    operator!=(const FF_Allocator<T>&, const FF_Allocator<U>&) { return false; }
}

#endif //FF_BSP_STL_ALLOCATOR_HPP
```

B.2 Parallel test programs

BSPinprod.cpp

```
#include <bsp_program.hpp>

struct BSPinprod : public bsp_node {

    int problem_size;

    explicit BSPinprod(int n) : problem_size{n} {};

    inline int nloc(int p, int s, int n) const {
        return (n + p - s - 1) / p;
    }

    double bspip(int p, int s, int n,
                 const std::vector<double>& x,
                 const std::vector<double>& y) {

        auto Inprod = get_empty_array<double>(p);

        double inprod = 0.0;

        for (int i{0}; i < nloc(p, s, n); ++i) {
            inprod += x.at(i) * y.at(i);
        }
        for (int t{0}; t < p; ++t) {
            Inprod.bsp_put(inprod, t, s);
        }
        bsp_sync();

        const auto& Inprod_arr = Inprod.BSPunsafe_access();
        double alpha = 0.0;
        for (int t{0}; t < p; ++t) {
            alpha += Inprod_arr.at(t);
        }
        return alpha;
```

```
    }

    void parallel_function() override {
        int n = problem_size;
        int p = bsp_nprocs();
        int s = bsp_pid();
        int nl = nloc(p, s, n);

        std::vector<double> x(nl);

        for (int i{0}; i < nl; ++i) {
            x[i] = i * p + s + 1;
        }

        bsp_sync();
        std::cout << "taking time 1" << std::endl;
        auto t1 = std::chrono::high_resolution_clock::now();
        double alpha = bspip(p, s, n, x, x);
        auto t2 = std::chrono::high_resolution_clock::now();
        auto time = std::chrono::duration_cast<std::chrono::milliseconds>(
                t2 - t1);
        if (s == 0) {
            std::cout << "Processor " << s << ": sum of squares up to "
                      << n << "*" << n << " is " << alpha << std::endl;
            std::cout << "Processor " << s << ": local time taken is "
                      << time.count()
                      << std::endl;
        }

    }
};

int main(int argc, char* argv[]) {
    if (argc == 1) {
        std::cerr << "Usage: " << argv[0] << " <P> (n)" << std::endl;
        std::cerr << "       where <P> is the number of processors used"
                  << std::endl;
        std::cerr << "       and (n) is the size of input vector, "
                  << "optional (default is 500000000)" << std::endl;
        return 1;
    }
    int p = std::stoi(argv[1]);
    int n = (argc >= 3) ? std::stoi(argv[2]) : 500000000;
    std::vector<std::unique_ptr<bsp_node>> nodes;
    for (int i{0}; i < p; ++i) {
        nodes.emplace_back(std::make_unique<BSPinprod>(n));
    }
    auto checksum = [n]() {
        auto dn = static_cast<double>(n);
        dn *= n + 1.0;
        dn *= 2.0 * n + 1.0;
        dn /= 6.0;
        std::cout << "Checksum: " << dn << std::endl;
    };
    bsp_program computation(std::move(nodes), nullptr, checksum);
    computation.start();
    return 0;
}
```

BSPpsrs.cpp

```cpp
#include <random>
#include <chrono>
#include <bsp_program.hpp>

struct BSPpsrs : public bsp_node {

    int n;
    int seed;

    explicit BSPpsrs(int problem_size, int random_seed = -1) :
            n{problem_size},
            seed{random_seed} {
    };

    void parallel_function() override {

        bsp_array<int> to_sort = get_empty_array<int>(1);
        int p = bsp_nprocs();
        int s = bsp_pid();
        auto t = std::chrono::high_resolution_clock::now();

        if (s == 0) {
            if (seed == -1) {
                std::random_device x;
                seed = x();
            }
            std::mt19937 mtw(seed);
            std::vector<int> data(n);
            std::iota(data.begin(), data.end(), 0);
            std::shuffle(data.begin(), data.end(), mtw);

#ifdef DEBUG
            std::ofstream print1{"generated_array.log"};
            for (const auto& el: data) print1 << el << std::endl;
            print1.close();
#endif
            auto t2 = std::chrono::duration_cast<std::chrono::milliseconds>(
                    std::chrono::high_resolution_clock::now() - t).count();
            std::cout << "Ended vector generation and shuffling (spent " << t2
                      << " ms)" <<
                      "\nstarting parallel part" << std::endl;

            t = std::chrono::high_resolution_clock::now();
            int numels = n / p;
            int count = 0;
            for (int i{0}; i < n; i += numels) {
                auto last = std::min(n, i + numels);
                to_sort.bsp_put(
                        std::vector<int>(data.begin() + i, data.begin() + last),
                        count++);
            }
        }

        bsp_sync();

        bsp_array<std::vector<int>> ps_array = get_empty_array<std::vector<int>>(
```

```
                p); // array of vectors!

        auto& vec = to_sort.BSPunsafe_access();

        std::sort(vec.begin(), vec.end());

#ifdef DEBUG
        std::ofstream print1a{"distributed_array-" + std::to_string(s) + ".log"};
        for (const auto& el: vec) print1a << el << std::endl;
        print1a.close();
#endif

        std::vector<int> primary_samples;

        size_t samplesize = vec.size() / p;
        size_t i;

        for (i = 0; i < vec.size(); i += samplesize) {
            primary_samples.emplace_back(vec.at(i));
        }

        if (i != vec.size() - 1) primary_samples.emplace_back(vec.at(i - 1));

#ifdef DEBUG
        std::ofstream print2{"primary_samples-" + std::to_string(s) + ".log"};
        for (const auto& el: primary_samples) print2 << el << std::endl;
        print2.close();
#endif

        for (i = 0; i < p; ++i) {
            ps_array.bsp_put(primary_samples, i, s);
        }

        bsp_sync();

        bsp_array<std::vector<int>> portion = get_empty_array<std::vector<int>>(
                p);

        std::vector<int> ps_all;
        std::vector<int> secondary_samples;
        const auto& psref = ps_array.BSPunsafe_access();

        for (const auto& vecs: psref) {
            ps_all.insert(ps_all.end(), vecs.begin(), vecs.end());
        }

        std::sort(ps_all.begin(), ps_all.end());

        samplesize = ps_all.size() / p;

        for (i = 0; i < ps_all.size(); i += samplesize) {
            secondary_samples.emplace_back(ps_all.at(i));
        }

        if (i == ps_all.size())
            secondary_samples.emplace_back(ps_all.at(i - 1));

#ifdef DEBUG
```

```
        std::ofstream print3{"secondary_samples-" + std::to_string(s) + ".log"};
        for (const auto& el: secondary_samples) print3 << el << std::endl;
        print3.close();
#endif

        int upperbound;
        int count = 1;

        do {
            upperbound = secondary_samples.at(count++);
        } while (vec.at(0) > upperbound);

        count--;
        std::vector<int> temp;
        for (i = 0; i < vec.size(); ++i) {
            if (vec.at(i) > upperbound) {
                portion.bsp_put(temp, count - 1, s);
                temp.clear();
                upperbound = secondary_samples.at(++count);
            }
            temp.emplace_back(vec.at(i));
        }

        portion.bsp_put(temp, count - 1, s);
        bsp_sync();

        bsp_array<std::vector<int>> final_arr =
                get_empty_array<std::vector<int>>(p);

        std::vector<int> secondary_block;

        auto& pbls_ref = portion.BSPunsafe_access();

        size_t totalsz = 0;
        for (const auto& pbl: pbls_ref) totalsz += pbl.size();

        secondary_block.reserve(totalsz);
        for (auto& pbl: pbls_ref) {
            secondary_block.insert(
                    secondary_block.end(),
                    std::make_move_iterator(pbl.begin()),
                    std::make_move_iterator(pbl.end()));
        }

        std::sort(secondary_block.begin(), secondary_block.end());

#ifdef DEBUG
        std::ofstream print4{"secondary_block-" + std::to_string(s) + ".log"};
        for (const auto& el: secondary_block) print4 << el << std::endl;
        print4.close();
#endif

        final_arr.bsp_put(secondary_block, 0, s);
        bsp_sync();

        if (s == 0) {
            auto& sbls_ref = final_arr.BSPunsafe_access();

```

```
            std::vector<int> final;
            for (auto& sbl: sbls_ref) {
                final.insert(final.end(), std::make_move_iterator(sbl.begin()),
                             std::make_move_iterator(sbl.end()));
            }

#ifdef DEBUG
            std::ofstream print5{"final.log"};
            for (const auto& el: final) print5 << el << std::endl;
            print5.close();
#endif

            bool passed = true;
            for (int j{0}; j < n; j++) {
                if (final[j] != j) {
                    passed = false;
                    break;
                }
            }

            std::cout << "Check " << (passed ? "passed" : "failed")
                      << std::endl;
            auto t1 = std::chrono::high_resolution_clock::now();
            std::cout << "Parallel part took "
                      << std::chrono::duration_cast<std::chrono::milliseconds>(
                              t1 - t).count() << " ms." << std::endl;
        }

        bsp_sync(); // Needed to keep all threads alive!
    }

};

int main(int argc, char* argv[]) {
    if (argc < 3) {
        std::cerr << "Usage: " << argv[0] << " <N> <P> (seed)" << std::endl;
        std::cerr << " where N is the problem size (power of 2)" << std::endl;
        std::cerr << "       P is the number of threads used (power of 2)"
                  << std::endl;
        std::cerr << "       N must be >= P^3" << std::endl;
        std::cerr
                << "       seed is an optional seed for permutations "
                << "(leave blank to randomize it)"
                << std::endl;
        return -1;
    }
    int n, p, s;
    n = std::stoi(argv[1]);
    p = std::stoi(argv[2]);
    if (n < p * p * p) {
        std::cerr << "N must be >= P^3" << std::endl;
        return -1;
    }
    s = (argc >= 4 ? std::stoi(argv[3]) : -1);
    std::vector<std::unique_ptr<bsp_node>> nodes;
    for (size_t i{0}; i < p; ++i) {
        nodes.push_back(std::make_unique<BSPpsrs>(n, s));
    }
```

```
    bsp_program mbsp(std::move(nodes));
    mbsp.start();
    return 0;
}
```

BSPfft.cpp

```
#include <cmath>
#include <fstream>
#include <bsp_program.hpp>

struct BSPfft : public bsp_node {

    std::vector<double> global_x;
    int global_n;

    explicit BSPfft(int _n, std::vector<double> glob) : global_n{_n}, global_x{
            std::move(glob)} {
    }

    constexpr static double PI = 3.141592653589793;

    static void ufft(std::vector<double>& x, int offset, int n, bool sign,
                     const std::vector<double>& w) {

        for (int k = 2; k <= n; k *= 2) {
            int nk = n / k;
            for (int r = 0; r < nk; ++r) {
                int rk = 2 * r * k;
                for (int j = 0; j < k; j += 2) {
                    double wr = w[j * nk];
                    double wi;
                    if (sign) {
                        wi = w[j * nk + 1];
                    } else {
                        wi = -w[j * nk + 1];
                    }

                    int j0 = rk + j + offset;
                    int j1 = j0 + 1;
                    int j2 = j0 + k;
                    int j3 = j2 + 1;

                    double taur = wr * x[j2] - wi * x[j3];
                    double taui = wi * x[j2] + wr * x[j3];

                    x[j2] = x[j0] - taur;
                    x[j3] = x[j1] - taui;
                    x[j0] += taur;
                    x[j1] += taui;
                }
            }
        }
    }

    static void ufft_init(int n, std::vector<double>& w) {
```

```
        assert(w.size() == n);

        if (n == 1) return;

        w[0] = 1.0;
        w[1] = 0.0;

        if (n == 4) {
            w[2] = 0.0;
            w[3] = -1.0;
        } else if (n >= 8) {
            double theta = -2.0 * PI / static_cast<double>(n);
            for (int j = 1; j <= n / 8; j++) {
                w[2 * j] = std::cos(j * theta);
                w[2 * j + 1] = std::sin(j * theta);
            }
            for (int j = 0; j < n / 8; j++) {
                int n4j = n / 4 - j;
                w[2 * n4j] = -w[2 * j + 1];
                w[2 * n4j + 1] = -w[2 * j];
            }
            for (int j = 1; j < n / 4; j++) {
                int n2j = n / 2 - j;
                w[2 * n2j] = -w[2 * j];
                w[2 * n2j + 1] = w[2 * j + 1];
            }
        }
    }

    static void twiddle(std::vector<double>& x, int length, bool sign,
                        const std::vector<double>& w, int offset) {
        for (int jo = 0; jo < 2 * length; jo += 2) {
            int j = jo;
            int j1 = j + 1;
            double wr = w[offset + j];
            double wi;

            if (sign) {
                wi = w[offset + j1];
            } else {
                wi = -w[offset + j1];
            }

            double xr = x[j];
            double xi = x[j1];
            x[j] = wr * xr - wi * xi;
            x[j1] = wi * xr + wr * xi;
        }
    }

    static void twiddle_init(int n, double alpha, const std::vector<int>& rho,
                             std::vector<double>& w, int offset) {
        double theta = -2.0 * PI * alpha / static_cast<double>(n);
        for (int j = 0; j < n; ++j) {
            double rt = static_cast<double>(rho[j]) * theta;
            w[offset + 2 * j] = std::cos(rt);
            w[offset + 2 * j + 1] = std::sin(rt);
        }
```

```
}

static void
permute(std::vector<double>& x, int n, const std::vector<int>& sigma,
        int line) {
    assert(x.size() / 2 == sigma.size());

    for (int j = 0; j < n; ++j) {
        if (j < sigma[j]) {
            int j0 = 2 * j;
            int j1 = j0 + 1;
            int j2 = 2 * sigma[j];
            int j3 = j2 + 1;
            double tmpr = x[j0];
            double tmpi = x[j1];
            x[j0] = x[j2];
            x[j1] = x[j3];
            x[j2] = tmpr;
            x[j3] = tmpi;
        }
    }
}

static void bitrev_init(std::vector<int>& rho) {
    int n = rho.size();

    auto binary_len = static_cast<int>(std::ceil(
            std::log(static_cast<double>(n)) / std::log(2.0)));
    std::vector<bool> bits(binary_len);
    std::vector<int> pwrs(binary_len);
    pwrs[0] = 1;
    for (int j = 1; j < binary_len; ++j) {
        pwrs[j] = pwrs[j - 1] * 2;
    }
    int j = 0;
    while (j < n - 1) {
        j++;
        int lastbit = 0;
        while (bits[lastbit]) {
            bits[lastbit] = false;
            lastbit++;
        }
        bits[lastbit] = true;
        int val = 0;
        for (int k = 0; k < binary_len; ++k) {
            if (bits[k]) {
                val += pwrs[binary_len - k - 1];
            }
        }
        rho[j] = val;
    }
}

static int k1_init(int n, int p) {
    assert(p < n);

    int np = n / p;
    int c;
```

```
    for (c = 1; c < p; c *= np);
    return n / c;
}

static void bspfft_init(int n, int p, int s, std::vector<double>& w0,
                        std::vector<double>& w,
                        std::vector<double>& tw, std::vector<int>& rho_np,
                        std::vector<int>& rho_p) {
    int np = n / p;
    bitrev_init(rho_np);

    if (p > 1) {
        bitrev_init(rho_p);
    }
    int k1 = k1_init(n, p);
    ufft_init(k1, w0);
    ufft_init(np, w);

    int ntw = 0;
    for (int c = k1; c <= p; c *= np) {
        double alpha = static_cast<double>(s % c) / static_cast<double>(c);
        twiddle_init(np, alpha, rho_np, tw, 2 * ntw * np);
        ntw++;
    }
}

static void calcError(const std::vector<double>& xlocal,
                      const std::vector<double>& xarr, int n, int p,
                      int s) {
    double error = 0.0;
    int c = 0;
    for (; c <= n / p; c++) {
        double lerror = std::abs(xlocal[c] - xarr[c]);
        error += lerror;
    }
    std::cout << s << ": local error is "
              << (error / static_cast<double>(n)) << std::endl;
}

void bspredistr(bsp_array<double>& x, int n, int p, int s, int c0, int c1,
                bool rev, const std::vector<int>& rho_p) {
    assert(1 <= c0);
    assert(c0 <= c1);
    assert(c1 <= p);

    auto xarr = x.BSPunsafe_access();
    int np = (int) xarr.size() / 2;
    int ratio = c1 / c0;
    int size = std::max(np / ratio, 1);
    int npackets = np / size;
    std::vector<double> tmp(2 * size);

    assert (p <= n);

    int j0, j2;
    if (rev) {
        j0 = rho_p[s] % c0;
        j2 = rho_p[s] / c0;
```

```
    } else {
        j0 = s % c0;
        j2 = s / c0;
    }

    for (int j = 0; j < npackets; j++) {
        int jglob = j2 * c0 * np + j * c0 + j0;
        int destproc = (jglob / (c1 * np)) * c1 + jglob % c1;
        int destindex = (jglob % (c1 * np)) / c1;
        for (int r = 0; r < size; ++r) {
            int tr = 2 * r;
            int tjrr = 2 * (j + r * ratio);
            tmp[tr] = xarr[tjrr];
            tmp[tr + 1] = xarr[tjrr + 1];
        }
        assert(destproc <= p);
        assert(destindex < xarr.size() / 2);
        x.bsp_put(tmp, destproc, 0, 2 * destindex, 2 * size);
    }
    bsp_sync();
}

void bspfft(bsp_array<double>& x, int n, int p, int s, bool sign,
            const std::vector<double>& w0, const std::vector<double>& w,
            const std::vector<double>& tw,
            const std::vector<int>& rho_np, const std::vector<int>& rho_p) {
    int np = n / p;
    int k1 = k1_init(n, p);
    permute(x.BSPunsafe_access(), np, rho_np, __LINE__);
    bool rev = true;

    for (int r = 0; r < np / k1; r++) {
        ufft(x.BSPunsafe_access(), 2 * r * k1, k1, sign, w0);
    }

    int c0 = 1;
    int ntw = 0;

    for (int c = k1; c <= p; c *= np) {
        bspredistr(x, n, p, s, c0, c, rev, rho_p);
        rev = false;
        twiddle(x.BSPunsafe_access(), np, sign, tw, 2 * ntw * np);
        ufft(x.BSPunsafe_access(), 0, np, sign, w);
        c0 = c;
        ntw++;
    }

    if (!sign) {
        auto& xarr = x.BSPunsafe_access();
        double ninv = 1.0 / static_cast<double>(n);
        for (int j = 0; j < 2 * np; ++j) {
            xarr[j] *= ninv;
        }
    }
}

std::vector<double>& fft(const std::vector<double>& xlocal) {
    int s = bsp_pid();
```

```
        int n = global_n / 2;
        int p = bsp_nprocs();
        int k1 = k1_init(n, p);
        std::vector<double> w0(k1);
        std::vector<double> w(n / p);
        std::vector<double> tw(2 * n / p);
        std::vector<int> rho_np(n / p);
        std::vector<int> rho_p(p);

        bsp_array<double> x = get_array(xlocal);

        auto time = std::chrono::high_resolution_clock::now();
        if (p == 1) {
            bspfft_init(n, p, s, w0, w, tw, rho_np, rho_p);
            permute(x.BSPunsafe_access(), n, rho_np, __LINE__);
            ufft(x.BSPunsafe_access(), 0, n, true, w);
            permute(x.BSPunsafe_access(), n, rho_np, __LINE__);
            ufft(x.BSPunsafe_access(), 0, n, false, w);
            auto& xi = x.BSPunsafe_access();
            double ninv = 1.0 / static_cast<double>(n);
            for (int j = 0; j < 2 * n; ++j) {
                xi[j] *= ninv;
            }
            calcError(x.BSPunsafe_access(), global_x, n, p, s);
            auto t = std::chrono::high_resolution_clock::now();
            auto tc = std::chrono::duration_cast<std::chrono::milliseconds>(
                    t - time).count();
            std::cout << "Parallel part took " << tc << " ms." << std::endl;
            return x.BSPunsafe_access();
        }

        bsp_sync();

        bspfft_init(n, p, s, w0, w, tw, rho_np, rho_p);
        std::cout << s << ": calling bspfft with n=" << n << ", p=" << p
                  << ", s=" << s << std::endl;
        bspfft(x, n, p, s, true, w0, w, tw, rho_np, rho_p);
        bsp_sync();

        std::cout << s << ": calling bspfft (inv) with n=" << n << ", p=" << p
                  << ", s=" << s << std::endl;
        bspfft(x, n, p, s, false, w0, w, tw, rho_np, rho_p);

        if (s == 0) {
            auto t = std::chrono::high_resolution_clock::now();
            auto tc = std::chrono::duration_cast<std::chrono::milliseconds>(
                    t - time).count();
            std::cout << "Parallel part took " << tc << " ms." << std::endl;
        }

        calcError(x.BSPunsafe_access(), xlocal, n, p, s);
        return x.BSPunsafe_access();
    }

    void parallel_function() override {

        bsp_array<double> local_x = get_empty_array<double>(1);
```

```
        if (bsp_pid() == 0) {
            auto p = bsp_nprocs();
            for (int s = 0; s < p; ++s) {
                std::vector<double> xlocal(global_n / p);
                int c = 0;
                int n1 = global_n / 2;
                for (int i = 0; i < 2 * n1; ++i) {
                    if ((i / 2) % p == s) {
                        xlocal[c++] = global_x[i];
                    }
                }
                local_x.bsp_put(xlocal, s);
            }
        }

        bsp_sync();
        fft(local_x.BSPunsafe_access());

    }
};

int main(int argc, char* argv[]) {
    if (argc < 3) {
        std::cerr << "Usage: " << argv[0] << " <N> <P>" << std::endl;
        std::cerr << " where N is the problem size (power of 2)" << std::endl;
        std::cerr << "       P is the number of threads used (power of 2)"
                  << std::endl;
        return -1;
    }
    int n, p;
    n = std::stoi(argv[1]);
    p = std::stoi(argv[2]);
    std::vector<double> global_x(n);
    for (int i = 0; i < n; i += 2) {
        global_x[i] = static_cast<double>(i) / 2.0;
    }

    std::vector<std::unique_ptr<bsp_node>> nodes;
    for (size_t i{0}; i < p; ++i) {
        nodes.push_back(std::make_unique<BSPfft>(n, global_x));
    }

    bsp_program mbsp(std::move(nodes));
    mbsp.start();
}
```

BSPlu.cpp

```
#include <bsp_program.hpp>
#include <cmath>
#include <limits>
#include <random>

class BSPlu : public bsp_node {

private:

```

```cpp
int param_n, param_M, param_N;
std::mt19937 rand;
std::uniform_real_distribution<double> dist{-32.768, 32.768};

template<typename T>
using matrix = std::vector<std::vector<T>>;

using clock = std::chrono::high_resolution_clock;
using milliseconds = std::chrono::milliseconds;

inline int nloc(int p, int s, int n) const {
    return (n + p - s - 1) / p;
}

void bsplu(int M, int N, int n, int s, int t,
           std::vector<int>& raw_pi, matrix<double>& a) {

    int p = bsp_pid();
    int P = bsp_nprocs();
    int p_i = p / N;
    int p_j = p % N;

    assert(p_i == s);
    assert(p_j == t);

    auto pi = get_array<int>(&raw_pi);

    std::vector<bsp_array<double>> A;
    for (int i{0}; i < a.size(); ++i) {
        A.emplace_back(get_array<double>(&a.at(i)));
    }
    std::vector<double> raw_pivots(M);
    auto pivots = get_array<double>(&raw_pivots);
    std::vector<int> raw_pivot_info(3);
    auto pivot_info = get_array<int>(&raw_pivot_info);
    std::vector<double> raw_pivot_row(n / N);
    auto pivot_row = get_array<double>(&raw_pivot_row);
    std::vector<double> raw_pivot_col(n / M);
    auto pivot_col = get_array<double>(&raw_pivot_col);

    bsp_sync();


    for (int i{0}; i < nloc(P, p, n); ++i) {
        raw_pi.at(i) = p + i * P;
    }

    for (int k{0}; k < n - 1; ++k) {
        int kdN = k / N;
        raw_pivot_info.at(0) = nloc(M, p_i, k);
        int kdM = raw_pivot_info.at(0);
        double temp_m;
        double absmax = 0.0;
        if (p_j == k % N) {
            for (int i{raw_pivot_info.at(0)}; i < a.size(); ++i) {
                if (std::abs(a.at(i).at(kdN)) > absmax) {
                    raw_pivot_info.at(0) = i;
                    absmax = std::abs(a.at(i).at(kdN));
```

```
                    }
                }
                temp_m = (raw_pivot_info.at(0) == a.size()) ?
                         -std::numeric_limits<double>::infinity() :
                         a.at(raw_pivot_info.at(0)).at(kdN);
                for (int i{0}; i < M; ++i) {
                    pivots.bsp_put(temp_m, i * N + p_j, p_i);
                }
            }

            bsp_sync();

            raw_pivot_info.at(1) = p_i;
            raw_pivot_info.at(2) = k % N;
            if (p_j == raw_pivot_info.at(2)) {
                temp_m = raw_pivots.at(0);
                raw_pivot_info.at(1) = 0;
                for (int i{1}; i < M; ++i) {
                    double temp = raw_pivots.at(i);
                    if (temp > temp_m) {
                        temp_m = temp;
                        raw_pivot_info.at(1) = i;
                    }
                }
                auto el = pivot_info.bsp_direct_get(
                        raw_pivot_info.at(1) * N + raw_pivot_info.at(2), 0);
                pivot_info.BSPunsafe_access().at(0) = el;
                for (int i{kdM}; i < a.size(); ++i) {
                    a.at(i).at(kdN) /= temp_m;
                }
                if (p_i == raw_pivot_info.at(1)) {
                    a.at(raw_pivot_info.at(0)).at(kdN) = temp_m;
                }

                for (int j{0}; j < N; j++) {
                    pivots.bsp_put(raw_pivots, p_i * N + j,
                                   raw_pivot_info.at(1), raw_pivot_info.at(1),
                                   1);
                    pivot_info.bsp_put(raw_pivot_info, p_i * N + j);
                }
            }

            bsp_sync();

            temp_m = raw_pivots.at(raw_pivot_info.at(1));

            assert(temp_m != 0.0);

            int temp_m_rho =
                    (M * raw_pivot_info.at(0) + raw_pivot_info.at(1)) % P;
            int temp_l_cyc =
                    (M * raw_pivot_info.at(0) + raw_pivot_info.at(1)) / P;

            if (p == temp_m_rho) {
                pi.bsp_get(k % P, k / P, temp_l_cyc, 1);
            }
            if (p == k % P) {
                pi.bsp_get(temp_m_rho, temp_l_cyc, k / P, 1);
```

```
        }
        if (p_i == raw_pivot_info.at(1)) {
            auto arr = A.at(k / M).bsp_direct_get((k % M) * N + p_j);
            if (arr.size() == 2)
                std::cout << "size 2" << std::endl;
            A.at(raw_pivot_info.at(0))
                    .bsp_put(arr);
        }
        if (p_i == k % M) {
            auto arr = A.at(raw_pivot_info.at(0)).bsp_direct_get(
                    raw_pivot_info.at(1) * N + p_j);
            if (arr.size() == 2)
                std::cout << "size 2" << std::endl;
            A.at(k / M)
                    .bsp_put(arr);
        }

        if (p_i == raw_pivot_info.at(1)) {
            for (int j{0}; j < a.at(0).size(); ++j) {
                raw_pivot_row.at(j) = a.at(raw_pivot_info.at(0)).at(j);
            }
            for (int i{0}; i < M; ++i) {
                if (i == p_i) continue;
                pivot_row.bsp_put(raw_pivot_row, i * N + p_j);
            }
        }

        if (p_j == raw_pivot_info.at(2)) {
            for (int i{0}; i < a.size(); ++i) {
                raw_pivot_col.at(i) = a.at(i).at(k / N);
            }
            for (int j{0}; j < N; ++j) {
                if (j == p_j) continue;
                pivot_col.bsp_put(raw_pivot_col, p_i * N + j);
            }
        }

        bsp_sync();

        if (p_i == raw_pivot_info.at(1)) {
            auto el = A.at(raw_pivot_info.at(0)).bsp_direct_get(
                    p_i * N + k % N);
            std::copy_n(el.begin() + k / N, 1,
                        raw_pivot_col.begin() + raw_pivot_info.at(0));
            //pivot_col.bsp_put(el, k/N, raw_pivot_info.at(0), 1);
        }

        bsp_sync();

        int istart = k / M;
        int jstart = k / N;
        // a.at(istart) = A.at(istart).BSPunsafe_access();
        if (p_i <= k % M) istart++;
        if (p_j <= k % N) jstart++;
        auto& raw_pivot_row2 = pivot_row.BSPunsafe_access();
        auto& raw_pivot_col2 = pivot_col.BSPunsafe_access();
        for (int i{istart}; i < a.size(); i++) {
            //a.at(i) = A.at(i).BSPunsafe_access();
```

```
                for (int j{jstart}; j < a.at(i).size(); ++j) {
                    a.at(i).at(j) -=
                            raw_pivot_row2.at(j) * raw_pivot_col2.at(i);
                }
            }
            bsp_sync();
        }
    }

public:

    BSPlu(int n, int M, int N, unsigned int seed) : param_n{n}, param_M{M},
                                                    param_N{N}, rand{seed} {

    };

    void parallel_function() override {
        auto time = clock::now();

        int n = param_n;
        int M = param_M;
        int N = param_N;
        int s = bsp_pid() / N;
        int t = bsp_pid() % N;
        int nlr = nloc(M, s, n);
        int nlc = nloc(N, t, n);
        matrix<double> a(nlr);
        for (int i{0}; i < nlr; ++i) {
            a.at(i).resize(nlc);
            for (int j{0}; j < nlc; ++j) {
                a.at(i).at(j) = dist(rand);
            }
        }

        std::vector<int> pi(nloc(bsp_nprocs(), bsp_pid(), n));
        auto timec = std::chrono::duration_cast<milliseconds>(
                clock::now() - time).count();
        std::cout << "Processor " << bsp_pid() << ": init took " << timec
                  << " ms." << std::endl;

        time = clock::now();
        bsplu(M, N, n, s, t, pi, a);
        timec = std::chrono::duration_cast<milliseconds>(
                clock::now() - time).count();
        if (bsp_pid() == 0)
            std::cout << "Processor " << bsp_pid() << ": LU took " << timec
                      << " ms." << std::endl;
    }
};

int main(int argc, char* argv[]) {
    if (argc < 4) {
        std::cerr << "Usage: " << argv[0] << " <n> <M> <N>" << std::endl;
        std::cerr << "       will start LU decomposition on a n by n matrix"
                  << std::endl;
        std::cerr << "       using M times N threads." << std::endl;
        return 1;
    }
```

```
    int n = std::stoi(argv[1]);
    int M = std::stoi(argv[2]);
    int N = std::stoi(argv[3]);

    std::random_device rd;
    std::vector<std::unique_ptr<bsp_node>> nodes;
    for (int i{0}; i < M * N; ++i) {
        nodes.emplace_back(std::make_unique<BSPlu>(n, M, N, rd()));
    }
    auto t0 = std::chrono::high_resolution_clock::now();
    auto f0 = [&t0]() {
        t0 = std::chrono::high_resolution_clock::now();
    };

    auto f1 = [&t0]() {
        auto t1 = std::chrono::high_resolution_clock::now();
        auto diff = std::chrono::duration_cast<std::chrono::milliseconds>(
                t1 - t0);
        std::cout << "Main part took " << diff.count() << " ms." << std::endl;
    };

    bsp_program computation(std::move(nodes), f0, f1);
    computation.start();
    return 0;
}
```

B.3 Sequential test programs

sequential_inprod.cpp

```
#include <string>
#include <chrono>
#include <iostream>

int main(int argc, char* argv[]) {

    if (argc < 2) return 1;

    int n = std::stoi(argv[1]);

    auto x = new double[n]();

    for (int i{0}; i < n; ++i) {
        x[i] = i + 1;
    }

    double inprod = 0.0;
    auto t1 = std::chrono::high_resolution_clock::now();
    for (int i{0}; i < n; ++i) {
        inprod += x[i] * x[i];
    }
    auto t2 = std::chrono::high_resolution_clock::now() - t1;
    std::cout << "Processor 0: sum of squares up to " << n << "*" << n << " is "
              << inprod << std::endl;
    std::cout << "Processor 0: local time taken is "
              << std::chrono::duration_cast<std::chrono::milliseconds>(
```

```
                        t2).count() << std::endl;

    auto dn = static_cast<double>(n);
    dn *= n + 1.0;
    dn *= 2.0 * n + 1.0;
    dn /= 6.0;
    std::cout << "Checksum: " << dn << std::endl;

    delete[] x;

    return 0;
}
```

sequential_sorting.cpp

```
#include <random>
#include <vector>
#include <iostream>
#include <algorithm>
#include <numeric>
#include <chrono>

int main(int argc, char* argv[]) {
    if (argc < 2) {
        std::cerr << "Usage: " << argv[0] << " <N> (seed)" << std::endl;
        std::cerr << " where N is the problem size (power of 2)" << std::endl;
        std::cerr
                << "       seed is an optional seed for permutations "
                << "(leave blank to randomize it)"
                << std::endl;
        return -1;
    }
    int n, s;
    n = std::stoi(argv[1]);
    s = (argc >= 3 ? std::stoi(argv[2]) : -1);
    auto t = std::chrono::high_resolution_clock::now();
    if (s == -1) {
        std::random_device x;
        s = x();
    }
    std::mt19937 mtw(s);
    std::vector<int> data(n);
    std::iota(data.begin(), data.end(), 0);
    std::shuffle(data.begin(), data.end(), mtw);

    int* dat2 = new int[n]();

    for (int i = 0; i < n; ++i) dat2[i] = data.at(i);

    auto t1 = std::chrono::duration_cast<std::chrono::milliseconds>(
            std::chrono::high_resolution_clock::now() - t).count();
    std::cout << "Ended vector generation and shuffling (spent " << t1 << " ms)"
              << "\nstarting sequential part" << std::endl;

    auto t2 = std::chrono::high_resolution_clock::now();

    std::sort(data.begin(), data.end());
```

```

    auto t3p = std::chrono::high_resolution_clock::now();

    std::sort(dat2, dat2 + n);

    auto t4 = std::chrono::high_resolution_clock::now();

    bool passed = true;
    for (int j{0}; j < n; j++) {
        if (data[j] != j) passed = false;
    }

    auto t3 = std::chrono::duration_cast<std::chrono::milliseconds>(
            t3p - t2).count();
    auto t4c = std::chrono::duration_cast<std::chrono::milliseconds>(
            t4 - t3p).count();

    std::cout << "Check " << (passed ? "passed" : "failed") << std::endl;
    std::cout << "Parallel part took " << t3 << " ms." << std::endl;
    std::cout << "Plain array part took " << t4c << " ms." << std::endl;

    delete[] dat2;

}
```

sequential_fft.cpp

```
#include <complex>
#include <iostream>
#include <vector>
#include <chrono>
#include <cassert>

constexpr static double PI = 3.141592653589793;

void ufft(std::vector<double>& x, int offset, int n, bool sign,
          const std::vector<double>& w) {

    for (int k = 2; k <= n; k *= 2) {
        int nk = n / k;
        for (int r = 0; r < nk; ++r) {
            int rk = 2 * r * k;
            for (int j = 0; j < k; j += 2) {
                double wr = w[j * nk];
                double wi;
                if (sign) {
                    wi = w[j * nk + 1];
                } else {
                    wi = -w[j * nk + 1];
                }

                int j0 = rk + j + offset;
                int j1 = j0 + 1;
                int j2 = j0 + k;
                int j3 = j2 + 1;

                double taur = wr * x[j2] - wi * x[j3];
```

```
                double taui = wi * x[j2] + wr * x[j3];

                x[j2] = x[j0] - taur;
                x[j3] = x[j1] - taui;
                x[j0] += taur;
                x[j1] += taui;
            }
        }
    }
}

void ufft_init(int n, std::vector<double>& w) {
    assert(w.size() == n);

    if (n == 1) return;

    w[0] = 1.0;
    w[1] = 0.0;

    if (n == 4) {
        w[2] = 0.0;
        w[3] = -1.0;
    } else if (n >= 8) {
        double theta = -2.0 * PI / static_cast<double>(n);
        for (int j = 1; j <= n / 8; j++) {
            w[2 * j] = std::cos(j * theta);
            w[2 * j + 1] = std::sin(j * theta);
        }
        for (int j = 0; j < n / 8; j++) {
            int n4j = n / 4 - j;
            w[2 * n4j] = -w[2 * j + 1];
            w[2 * n4j + 1] = -w[2 * j];
        }
        for (int j = 1; j < n / 4; j++) {
            int n2j = n / 2 - j;
            w[2 * n2j] = -w[2 * j];
            w[2 * n2j + 1] = w[2 * j + 1];
        }
    }
}


void twiddle_init(int n, double alpha, const std::vector<int>& rho,
                  std::vector<double>& w, int offset) {
    double theta = -2.0 * PI * alpha / static_cast<double>(n);
    for (int j = 0; j < n; ++j) {
        double rt = static_cast<double>(rho[j]) * theta;
        w[offset + 2 * j] = std::cos(rt);
        w[offset + 2 * j + 1] = std::sin(rt);
    }
}

void permute(std::vector<double>& x, int n, const std::vector<int>& sigma) {
    assert(x.size() / 2 == sigma.size());

    for (int j = 0; j < n; ++j) {
        if (j < sigma[j]) {
            int j0 = 2 * j;
```

```
            int j1 = j0 + 1;
            int j2 = 2 * sigma[j];
            int j3 = j2 + 1;
            double tmpr = x[j0];
            double tmpi = x[j1];
            x[j0] = x[j2];
            x[j1] = x[j3];
            x[j2] = tmpr;
            x[j3] = tmpi;
        }
    }
}

void bitrev_init(std::vector<int>& rho) {
    int n = rho.size();

    auto binary_len = static_cast<int>(std::ceil(
            std::log(static_cast<double>(n)) / std::log(2.0)));
    std::vector<bool> bits(binary_len);
    std::vector<int> pwrs(binary_len);
    pwrs[0] = 1;
    for (int j = 1; j < binary_len; ++j) {
        pwrs[j] = pwrs[j - 1] * 2;
    }
    int j = 0;
    while (j < n - 1) {
        j++;
        int lastbit = 0;
        while (bits[lastbit]) {
            bits[lastbit] = false;
            lastbit++;
        }
        bits[lastbit] = true;
        int val = 0;
        for (int k = 0; k < binary_len; ++k) {
            if (bits[k]) {
                val += pwrs[binary_len - k - 1];
            }
        }
        rho[j] = val;
    }
}


void fft_init(int n, std::vector<double>& w,
              std::vector<double>& tw, std::vector<int>& rho_np) {
    bitrev_init(rho_np);
    ufft_init(n, w);

    twiddle_init(n, 0, rho_np, tw, 0);
}

static void
calcError(const std::vector<double>& xlocal, const std::vector<double>& xarr) {
    double error = 0.0;
    for (int c{0}; c < xlocal.size(); c++) {
        double lerror = std::abs(xlocal[c] - xarr[c]);
        error += lerror;
```

```
    }
    std::cout << "local error is "
              << (error / static_cast<double>(xlocal.size())) << std::endl;
}

int main(int argc, char* argv[]) {
    if (argc < 2) {
        std::cerr << "Usage: " << argv[0] << " <N>" << std::endl;
        std::cerr << " where N is the problem size (power of 2)" << std::endl;
        return -1;
    }
    int n;
    n = std::stoi(argv[1]);
    std::vector<double> data(n);

    n /= 2;
    for (int i = 0; i < n; i += 2) {
        data[i] = static_cast<double>(i) / 2.0;
    }

    std::vector<double> old(data);

    std::vector<double> w(n);
    std::vector<double> tw(2 * n);
    std::vector<int> rho_np(n);

    auto t = std::chrono::high_resolution_clock::now();
    fft_init(n, w, tw, rho_np);
    // forward fft
    permute(data, n, rho_np);
    ufft(data, 0, n, true, w);

    // inverse fft
    permute(data, n, rho_np);
    ufft(data, 0, n, false, w);

    double ninv = 1.0 / static_cast<double>(n);
    for (int j = 0; j < 2 * n; ++j) {
        data[j] *= ninv;
    }

    calcError(data, old);
    auto t1 = std::chrono::high_resolution_clock::now();


    std::cout << "Sequential part took "
              << std::chrono::duration_cast<std::chrono::milliseconds>(
                     t1 - t).count() << " ms." << std::endl;

    calcError(data, old);


    return 0;
}
```

sequential_lu.cpp

```
#include <iostream>
#include <chrono>
#include "lib/src/linalg.h"

int main(int argc, char* argv[]) {
    if (argc < 2) {
        std::cerr << "Usage: " << argv[0] << " <n>" << std::endl;
        std::cerr << "       will start LU decomposition on a n by n matrix."
                  << std::endl;
        return 1;
    }

    int n = std::stoi(argv[1]);

    alglib::real_2d_array a;
    alglib::integer_1d_array piv;

    a.setlength(n, n);
    piv.setlength(n);

    for (int i{0}; i < n; ++i) {
        for (int j{0}; j < n; ++j) {
            a[i][j] = (alglib::randomreal() * 65.536) - 32.768;
        }
    }

    auto t1 = std::chrono::high_resolution_clock::now();
    alglib::rmatrixlu(a, n, n, piv);
    auto t2 = std::chrono::high_resolution_clock::now();

    auto time = std::chrono::duration_cast<std::chrono::milliseconds>(t2 - t1);
    std::cout << "Sequential computation took " << time.count() << " ms."
                                                                << std::endl;
}
```

B.4 Miscellaneous

commstress.cpp

```
#include <bsp_program.hpp>
#include <random>
#include <iostream>


/**
 * benchmarks the communication aspects of the bsp pattern
 */
struct communication_stress : public bsp_node {
    int NITERS = 10000;

    void parallel_function() override {
        using clock = std::chrono::high_resolution_clock;
        using duration = std::chrono::nanoseconds;

        std::random_device rd;
```

```cpp
        std::mt19937 mersenne(rd());
        std::uniform_int_distribution uid(0, NITERS - 1);

        long long int rng_duration = 0;
        if (bsp_pid() == 0) {
            // benchmarking the time spent on an RNG call
            int x = 0;
            auto rng_start = clock::now();
            for (int i{0}; i < NITERS; ++i) {
                if (i % 2) x += uid(mersenne);
                else x -= uid(mersenne);
            }
            auto rng_stop = clock::now();
            auto dur = std::chrono::duration_cast<duration>(
                    rng_stop - rng_start).count();
            rng_duration = dur / NITERS;
            // Use the value
            std::cout << "RNG benchmark: a single RNG call takes "
                      << rng_duration << " ns (value is " << x
                      << "), whole op takes "
                      << dur << "ns" << std::endl;
        }

        // Test #1: insertion of random values inside random workers' variables

        bsp_variable<int> var = get_variable<int>(0);

        auto start = clock::now();
        for (size_t i{0}; i < NITERS; ++i) {
            size_t next = uid(mersenne) % bsp_nprocs();
            var.bsp_put(uid(mersenne), next);
            bsp_sync();
        }

        if (bsp_pid() == 0) {
            auto time = std::chrono::duration_cast<duration>(
                    clock::now() - start).count() - (2 * NITERS * rng_duration);
            std::cout << "--- PHASE 1 - VARIABLES ---" << std::endl;
            std::cout << "Spent " << time << "ns" << std::endl;
            std::cout << "That is, " << time / NITERS << "ns per operation"
                      << std::endl;
        }

        // Test #2: replacement of arrays inside random worker's memory

        bsp_array<int> arr1 = get_empty_array<int>(NITERS);
        std::vector<int> swap(NITERS);

        auto gen = [&]() { return uid(mersenne); };
        std::generate(swap.begin(), swap.end(), gen);

        start = clock::now();
        for (size_t i{0}; i < NITERS; ++i) {
            size_t next = uid(mersenne) % bsp_nprocs();
            arr1.bsp_put(swap, next);
            bsp_sync();
        }

```

```
        if (bsp_pid() == 0) {
            auto time = std::chrono::duration_cast<duration>(
                    clock::now() - start).count() - (NITERS * rng_duration);
            std::cout << "--- PHASE 2 - ARR_SWAP ---" << std::endl;
            std::cout << "Spent " << time << "ns" << std::endl;
            std::cout << "That is, " << time / NITERS << "ns per operation"
                      << std::endl;
        }

        // Test #3: replacement of portion of arrays inside
        // random positions in a random worker's memory

        bsp_array<int> arr2 = get_empty_array<int>(NITERS);
        std::vector<int> empl(10);
        std::generate(empl.begin(), empl.end(), gen);

        start = clock::now();
        for (size_t i{0}; i < NITERS; ++i) {
            size_t next = uid(mersenne) % bsp_nprocs();
            size_t pos = uid(mersenne) % (NITERS - 10);
            arr2.bsp_put(empl, next, pos, 10);
            bsp_sync();
        }

        if (bsp_pid() == 0) {
            auto time = std::chrono::duration_cast<duration>(
                    clock::now() - start).count() - (2 * NITERS * rng_duration);
            std::cout << "--- PHASE 3 - ARR_EMPLACE ---" << std::endl;
            std::cout << "Spent " << time << "ns" << std::endl;
            std::cout << "That is, " << time / NITERS << "ns per operation"
                      << std::endl;
        }

        // Test #4: replacement of elements in random positions
        // in arrays inside random worker's memory

        bsp_array<int> arr3 = get_empty_array<int>(NITERS);

        start = clock::now();
        for (size_t i{0}; i < NITERS; ++i) {
            size_t next = uid(mersenne) % bsp_nprocs();
            for (size_t j{0}; j < 50; ++j) {
                size_t pos = uid(mersenne) % NITERS;
                arr3.bsp_put(bsp_pid(), next, pos);
            }
            bsp_sync();
        }

        if (bsp_pid() == 0) {
            auto time = std::chrono::duration_cast<duration>(
                    clock::now() - start).count() - (NITERS * rng_duration) -
                        (50 * NITERS * rng_duration);
            std::cout << "--- PHASE 4 - ARR_PUT ---" << std::endl;
            std::cout << "Spent " << time << "ns" << std::endl;
            std::cout << "That is, " << time / (NITERS * 50)
                      << "ns per operation" << std::endl;
        }
    }
```

```
};

int main() {
    std::vector<std::unique_ptr<bsp_node>> nodes;
    for (size_t i{0}; i < 4; ++i) {
        nodes.push_back(std::make_unique<communication_stress>());
    }
    bsp_program mbsp(std::move(nodes));
    mbsp.start();
}
```

BSPbench.cpp

```
#include <array>
#include <cmath>
#include <bsp_program.hpp>

struct bspbench : public bsp_node {

    int maxH, maxN, niters;

    const double MEGA = 1000000.0;

    bspbench(int _maxH, int _maxN, int _niters) :
            maxH{_maxH}, maxN{_maxN}, niters{_niters} {};

    std::array<double, 2>
    leastsquares(int h0, int h1, const std::vector<double>& t) const {
        std::array<double, 2> ret{};
        auto nh = static_cast<double>(h1 - h0 + 1);

        double sumt = 0.0;
        double sumth = 0.0;

        for (int h = h0; h <= h1; ++h) {
            sumt += t[h];
            sumth += t[h] * h;
        }
        double sumh = static_cast<double>(h1 * h1 - h0 * h0 + h1 + h0) / 2;
        double sumhh = static_cast<double>(h1 * (h1 + 1) * (2 * h1 + 1) -
                                           (h0 - 1) * h0 * (2 * h0 - 1)) / 6;

        if (std::abs(nh) > std::abs(sumh)) {
            double a = sumh / nh;
            ret[0] = (sumth - a * sumt) / (sumhh - a * sumh);
            ret[1] = (sumt - sumh * ret[0]) / nh;
        } else {
            double a = nh / sumh;
            ret[0] = (sumt - a * sumth) / (sumh - a * sumhh);
            ret[1] = (sumth - sumhh * ret[0]) / sumh;
        }

        return ret;
    }

    void parallel_function() override {

```

```
        double r = 0.0;

        std::vector<double> x(maxN);
        std::vector<double> y(maxN);
        std::vector<double> z(maxN);

        std::vector<int> destproc(maxH);
        std::vector<int> destindex(maxH);
        std::vector<double> src(maxH);

        std::vector<double> t(maxH + 1);

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

        bsp_array<double> Time = get_empty_array<double>(p);
        bsp_array<double> Dest = get_empty_array<double>(2 * maxH + p);

        for (int n = 1; n <= maxN; n *= 2) {

            double alpha = 1.0 / 3.0;
            double beta = 4.0 / 9.0;
            for (int i = 0; i < n; ++i) {
                z[i] = x[i] = y[i] = i;
            }

            auto time0 = std::chrono::high_resolution_clock::now();
            for (int iter = 0; iter < niters; ++iter) {
                for (int i = 0; i < n; ++i) {
                    y[i] += alpha * x[i];
                }
                for (int i = 0; i < n; ++i) {
                    z[i] -= beta * x[i];
                }
            }
            auto time1 = std::chrono::high_resolution_clock::now();
            auto time = std::chrono::duration_cast<std::chrono::milliseconds>(
                    time1 - time0).count() / 1000.0;
            Time.bsp_put(time, 0, s);
            bsp_sync();

            if (s == 0) {
                auto time_arr = Time.BSPunsafe_access();
                double mintime = time_arr[0];
                double maxtime = time_arr[0];
                for (int s1 = 1; s1 < p; ++s1) {
                    mintime = std::min(mintime, time_arr[s1]);
                    maxtime = std::max(maxtime, time_arr[s1]);
                }
                if (mintime > 0.0) {
                    int nflops = 4 * niters * n;
                    r = 0.0;
                    for (int s1 = 0; s1 < p; ++s1) {
                        r += static_cast<double>(nflops) / time_arr[s1];
                    }
                    r /= static_cast<double>(p);
                    std::cout << "n= "
                              << n << " min= "
```

```
                                << nflops / (maxtime * MEGA) << " max= "
                                << nflops / (mintime * MEGA) << " av= "
                                << r / MEGA << " Mflop/s" << std::endl;
                    std::cout << " fool=" << y[n - 1] + z[n - 1] << std::endl;
                } else {
                    std::cout << "minimum time is 0" << std::endl;
                }
            }
        }

        for (int h = 0; h <= maxH; ++h) {
            for (int i = 0; i < h; ++i) {
                src[i] = i;
                if (p == 1) {
                    destproc[i] = 0;
                    destindex[i] = i;
                } else {
                    destproc[i] = (s + 1 + i % (p - 1)) % p;
                    destindex[i] = s + (i / (p - 1)) * p;
                }
            }

            auto time0 = std::chrono::high_resolution_clock::now();
            for (int iter = 0; iter < niters; ++iter) {
                for (int i = 0; i < h; i++) {
                    Dest.bsp_put(src[i], destproc[i], destindex[i]);
                }
                bsp_sync();
            }
            auto time1 = std::chrono::high_resolution_clock::now();
            auto time = std::chrono::duration_cast<std::chrono::milliseconds>(
                    time1 - time0).count() / 1000.0;

            if (s == 0) {
                t[h] = (time * r) / static_cast<double>(niters);
                std::cout << "Time of " << h << "-relation= "
                          << time / niters << " sec= " << t[h] << " flops"
                          << std::endl;
            }
        }

        if (s == 0) {
            auto temp = leastsquares(0, p, t);
            std::cout << "Range h=0 to p: g= " << temp[0] << ", l= " << temp[1]
                      << std::endl;
            temp = leastsquares(p, maxH, t);
            std::cout << "Range h=p to HMAX: g= " << temp[0] << ", l= "
                      << temp[1] << std::endl;

            std::cout << "The bottom line for this BSP computer is:"
                      << std::endl;
            std::cout << "p= " << p << ", r= " << r / MEGA << " Mflop/s, g= "
                      << temp[0] << ", l= " << temp[1] << std::endl;

        }
    }

};
```

```

int main(int argc, char* argv[]) {

    int nprocs, maxh, maxn, niters;

    if (argc == 1) {
        std::cerr << "Usage: " << argv[0] << " <P> (NITERS) (MAXN) (MAXH)"
                  << std::endl;
        std::cerr << "<..> are obligatory parameters" << std::endl;
        std::cerr << "(..) are optional" << std::endl;
        return -1;
    }

    nprocs = std::stoi(argv[1]);

    if (argc >= 3) {
        niters = std::stoi(argv[2]);
    } else {
        niters = 10000;
    }

    if (argc >= 4) {
        maxn = std::stoi(argv[3]);
    } else {
        maxn = 1024;
    }

    if (argc >= 5) {
        maxh = std::stoi(argv[4]);
    } else {
        maxh = 128;
    }

    std::vector<std::unique_ptr<bsp_node>> nodes;
    for (size_t i{0}; i < nprocs; ++i) {
        nodes.push_back(std::make_unique<bspbench>(maxh, maxn, niters));
    }
    bsp_program mbsp(std::move(nodes));
    mbsp.start();
}
```

ff_example.cpp

```
#include <iostream>
#include <iterator>
#include <algorithm>
#include "bsp_program.hpp"

struct ff_example : public bsp_node {
    void parallel_function() override {
        int s = bsp_pid();
        auto v1 = (const std::vector<long>*) fastflow_input;
        long count = 0;
        int portion_size = v1->size() / bsp_nprocs();
        for (int i{portion_size * s}; i < portion_size * (s + 1); ++i)
            count += v1->at(i);
        bsp_array<long> counts = get_empty_array<long>(bsp_nprocs());
```

```
        counts.bsp_put(count, 0, s);
        bsp_sync();
        if (s == 0) {
            long total = 0;
            for (long l: counts.BSPunsafe_access()) {
                total += l;
            }
            emit_output((void*) total);
        }
    }
};

struct generator : public ff::ff_node {
    void* svc(void* in) override {
        auto source = new std::vector<long>;
        for (int i{0}; i < 10; ++i) source->push_back(i);
        ff_send_out(source);
        return EOS;
    }
};

struct checker : public ff::ff_node {
    void* svc(void* in) override {
        if (in != GO_ON && in != EOS) {
            auto val = (long) in;
            if (val == 45) std::cout << "OK" << std::endl;
            else std::cout << "KO" << std::endl;
        }
        return GO_ON;
    }
};

int main() {
    std::vector<std::unique_ptr<bsp_node>> nodes;
    for (int i{0}; i < 2; ++i) {
        nodes.push_back(std::make_unique<ff_example>());
    }
    auto pre_fun = []() {
        std::cout << "Before BSP computation" << std::endl;
    };
    auto post_fun = []() {
        std::cout << "After BSP computation" << std::endl;
    };
    bsp_program computation(std::move(nodes), pre_fun, post_fun);
    generator g;
    checker c;

    ff::ff_Pipe<> pipe(g, computation, c);

    if (pipe.run_and_wait_end() < 0)
        std::cout << "Error in running pipe" << std::endl;
    return 0;
}
```

mwe.cpp

```
#include <iostream>
#include <iterator>
#include <algorithm>
#include <thread>
#include "bsp_program.hpp"

struct my_bsp_comp : public bsp_node {
    void parallel_function() override {
        int s = bsp_pid();
        int n = bsp_nprocs();
        auto v1 = get_variable<int>(10);
        v1.bsp_put(s, (s + 1) % n);
        bsp_sync();
        std::vector<int> init{s, s, s, s};
        auto a1 = get_array<int>(init);
        a1.bsp_put(init, (s + 1) % n, 0, 2, 2);
        bsp_sync();
        auto a = a1.bsp_direct_get(s);
        std::this_thread::sleep_for(std::chrono::seconds(s));
        std::cout << s << ": v1 is " << v1.bsp_direct_get(s) << std::endl;
        std::cout << s << ": a1 is ";
        std::copy(a.begin(),
                  a.end(),
                  std::ostream_iterator<int>(std::cout, " "));
        std::cout << std::endl;
    }
};

int main() {
    std::vector<std::unique_ptr<bsp_node>> nodes;
    for (int i{0}; i < 2; ++i) {
        nodes.push_back(std::make_unique<my_bsp_comp>());
    }
    auto pre_fun = []() {
        std::cout << "Before BSP computation" << std::endl;
    };
    auto post_fun = []() {
        std::cout << "After BSP computation" << std::endl;
    };
    bsp_program computation(std::move(nodes), pre_fun, post_fun);
    computation.start();
    return 0;
}
```

unit_tests.cpp

```
#include <bsp_program.hpp>

static bool ok = true;
static int total_tests = 35;

static int mod(int k, int n) {
    return ((k %= n) < 0) ? k + n : k;
}

```

```cpp
static void assertint(int a, int b, int id, int line) {
    std::string s;
    if (a != b) {
        s += "Node " + std::to_string(id) + ": mismatch at line " +
             std::to_string(line)
             + " (expected " + std::to_string(b) + ", obtained " +
             std::to_string(a) + ")";
        std::cerr << s << std::endl;
        ok = false;
    }
}

static void assertint(int a, int b, int c, int id, int line) {
    std::string s;
    if (a != b && a != c) {
        s += "Node " + std::to_string(id) + ": mismatch at line " +
             std::to_string(line)
             + " (expected " + std::to_string(b) + " or " + std::to_string(c) +
             ", obtained " + std::to_string(a) + ")";
        std::cerr << s << std::endl;
        ok = false;
    }
}

static void
assertvec(const std::vector<int>& a, const std::vector<int>& b, int id,
          int line) {
    std::string s;
    auto printvect = [](const std::vector<int>& v) {
        std::string s{"["};
        for (const auto& el: v) {
            s += std::to_string(el) + " ";
        }
        s.pop_back();
        s += "]";
        return s;
    };
    if (a != b) {
        s += "Node " + std::to_string(id) + ": mismatch at line " +
             std::to_string(line)
             + " (expected " + printvect(b) + ", obtained " + printvect(a) +
             ")";
        std::cerr << s << std::endl;
        ok = false;
    }
}

static void assertvec(const std::vector<int>& a, const std::vector<int>& b,
                      const std::vector<int>& c, int id, int line) {
    std::string s;
    auto printvect = [](const std::vector<int>& v) {
        std::string s{"["};
        for (const auto& el: v) {
            s += std::to_string(el) + " ";
        }
        s.pop_back();
        s += "]";
        return s;
```

```
    };
    if (a != b && a != c) {
        s += "Node " + std::to_string(id) + ": mismatch at line " +
             std::to_string(line)
             + " (expected " + printvect(b) + " or " + printvect(c) +
             ", obtained " + printvect(a) + ")";
        std::cerr << s << std::endl;
        ok = false;
    }
}

struct unit_tests : public bsp_node {

    void parallel_function() override {

        auto endfun = [this](bool end = false) {
            static int passed_tests = 0;
            if (end) {
                std::cout << "Number of tests executed successfully: "
                          << passed_tests << "/" << total_tests << std::endl;
                if (passed_tests == total_tests)
                    std::cout << "ALL TESTS COMPLETED SUCCESSFULLY!"
                              << std::endl;
                else
                    std::cerr
                            << "Some tests failed, "
                            << "refer to the logs for more information"
                            << std::endl;
            } else {
                if (ok) {
                    passed_tests++;
                    std::cout << "Test passed";
                } else std::cout << "Test failed";
                std::cout << " (" << passed_tests << "/" << total_tests << ")"
                          << std::endl;
                ok = true;
            }
        };


        if (bsp_pid() == 0) {
            std::cout
                    << "PART 1 : VARIABLES (SEQ)"
                    << std::endl;
            std::cout
                    << "Test 1: bsp_put(const T& elem, int id)"
                    << std::endl;
        }

        bsp_variable<int> v1 = get_variable(-1);

        v1.bsp_put(bsp_pid(), bsp_pid());

        bsp_sync();

        assertint(v1.BSPunsafe_access(), bsp_pid(), bsp_pid(), __LINE__);

        if (bsp_pid() == 0) {
```

```
        endfun();
        std::cout
                << "Test 2 : bsp_put(const bsp_variable<T>& other)"
                << std::endl;
    }

    bsp_variable<int> v2 = get_variable(-1);

    v2.bsp_put(v1);

    bsp_sync();

    assertint(v2.BSPunsafe_access(), bsp_pid(), bsp_pid(), __LINE__);

    if (bsp_pid() == 0) {
        endfun();
        std::cout
                << "Test 3 : bsp_put(const bsp_variable<T>& other, int id)"
                << std::endl;
    }

    bsp_variable<int> v3 = get_variable(-1);

    v3.bsp_put(v2, mod((bsp_pid() + 1), bsp_nprocs()));
    bsp_sync();

    assertint(v3.BSPunsafe_access(), mod((bsp_pid() - 1), bsp_nprocs()),
              bsp_pid(), __LINE__);

    if (bsp_pid() == 0) {
        endfun();
        std::cout
                << "Test 4: bsp_put(int destination)"
                << std::endl;
    }

    bsp_variable<int> v4 = get_variable(bsp_pid());

    int nextnode = mod(bsp_pid() + 1, bsp_nprocs());
    v4.bsp_put(nextnode);
    bsp_sync();

    assertint(v4.BSPunsafe_access(), mod((bsp_pid() - 1), bsp_nprocs()),
              bsp_pid(), __LINE__);

    if (bsp_pid() == 0) {
        endfun();
        std::cout
                << "Test 5: bsp_get(int destination)"
                << std::endl;
    }

    bsp_variable<int> v5 = get_variable(bsp_pid());
    bsp_sync(); // needed to make sure all nodes have time to create the variable

    v5.bsp_get(nextnode);
    bsp_sync();

```

```
        assertint(v5.BSPunsafe_access(), nextnode, bsp_pid(), __LINE__);

        if (bsp_pid() == 0) {
            endfun();
            std::cout
                    << "PART 2 : VARIABLES (PAR)"
                    << std::endl;
            std::cout
                    << "Test 6: bsp_put(const T& elem, int id)"
                    << std::endl;
        }

        bsp_variable<int> v6 = get_variable(bsp_pid());

        if (bsp_pid() != 0) v6.bsp_put(bsp_pid(), 0);
        bsp_sync();

        if (bsp_pid() == 0) {
            assertint(v6.BSPunsafe_access(), 1, 2, bsp_pid(), __LINE__);
            endfun();
            std::cout
                    << "Test 7 : bsp_put(const bsp_variable<T>& other, int id)"
                    << std::endl;
        }

        bsp_variable<int> v7 = get_variable(-1);
        if (bsp_pid() != 0) v7.bsp_put(v6, 0);
        bsp_sync();

        if (bsp_pid() == 0) {
            assertint(v7.BSPunsafe_access(), 1, 2, bsp_pid(), __LINE__);
            endfun();
            std::cout
                    << "Test 8: bsp_put(int destination)"
                    << std::endl;
        }

        bsp_variable<int> v8 = get_variable(bsp_pid());
        if (bsp_pid() != 0) v8.bsp_put(0);
        bsp_sync();

        if (bsp_pid() == 0) {
            assertint(v8.BSPunsafe_access(), 1, 2, bsp_pid(), __LINE__);
            endfun();
            std::cout
                    << "PART 3: ARRAYS (SEQ)"
                    << std::endl;
            std::cout
                    << "Test 9 : bsp_put(const T& elem, int pos)"
                    << std::endl;
        }

        std::vector<int> base_array{9, 8, 7, 6, 5, 4, 3, 2, 1, 0};
        std::vector<int> compare_array1{bsp_pid(), 8, 7, 6, 5, 4, 3, 2, 1, 0};

        bsp_array<int> a1 = get_array(base_array);
        a1.bsp_put(bsp_pid(), 0);
        bsp_sync();
```

```
        assertvec(a1.BSPunsafe_access(), compare_array1, bsp_pid(), __LINE__);

        if (bsp_pid() == 0) {
            endfun();
            std::cout
                    << "Test 10: bsp_put(const bsp_variable<T>& elem, int pos)"
                    << std::endl;
        }

        bsp_array<int> a2 = get_array(base_array);
        bsp_variable<int> v9 = get_variable(bsp_pid());
        a2.bsp_put(v9, 0);
        bsp_sync();

        assertvec(a2.BSPunsafe_access(), compare_array1, bsp_pid(), __LINE__);

        if (bsp_pid() == 0) {
            endfun();
            std::cout
                    << "Test 11 : bsp_put(T elem, int dest, int pos)"
                    << std::endl;
        }

        int prevnode = mod(bsp_pid() - 1, bsp_nprocs());
        std::vector<int> compare_array2{prevnode, 8, 7, 6, 5, 4, 3, 2, 1, 0};
        bsp_array<int> a3 = get_array(base_array);

        a3.bsp_put(bsp_pid(), nextnode, 0);
        bsp_sync();

        assertvec(a3.BSPunsafe_access(), compare_array2, bsp_pid(), __LINE__);
        if (bsp_pid() == 0) {
            endfun();
            std::cout
                    << "Test 12 : bsp_put(bsp_variable<T>"
                    << "elem, int dest, int pos)"
                    << std::endl;
        }

        bsp_array<int> a4 = get_array(base_array);
        bsp_variable<int> v10 = get_variable(bsp_pid());
        a4.bsp_put(v10, nextnode, 0);
        bsp_sync();

        assertvec(a4.BSPunsafe_access(), compare_array2, bsp_pid(), __LINE__);

        if (bsp_pid() == 0) {
            endfun();
            std::cout
                    << "Test 13 : bsp_put(std::vector<T> other)"
                    << std::endl;

        }

        std::vector<int> replace_array{bsp_pid(), 1, 2, 3, 4};
        bsp_array<int> a5 = get_array(base_array);

```

```
        a5.bsp_put(replace_array);
        bsp_sync();

        assertvec(a5.BSPunsafe_access(), replace_array, bsp_pid(), __LINE__);

        if (bsp_pid() == 0) {
            endfun();
            std::cout
                    << "Test 14 : bsp_put(bsp_array<T> other)"
                    << std::endl;
        }

        bsp_array<int> a6 = get_array(base_array);
        bsp_array<int> a7 = get_array(replace_array);

        a6.bsp_put(a7);
        bsp_sync();

        assertvec(a6.BSPunsafe_access(), replace_array, bsp_pid(), __LINE__);

        if (bsp_pid() == 0) {
            endfun();
            std::cout
                    << "Test 15 : bsp_put(std::vector<T> other, int dest)"
                    << std::endl;
        }

        auto control_array1 = std::vector<int>{prevnode, 1, 2, 3, 4};

        bsp_array<int> a8 = get_array(base_array);

        a8.bsp_put(replace_array, nextnode);
        bsp_sync();

        assertvec(a8.BSPunsafe_access(), control_array1, bsp_pid(), __LINE__);

        if (bsp_pid() == 0) {
            endfun();
            std::cout
                    << "Test 16 : bsp_put(bsp_array<T> other, int dest)"
                    << std::endl;
        }

        bsp_array<int> a9 = get_array(base_array);
        bsp_array<int> a10 = get_array(replace_array);

        a9.bsp_put(a10, nextnode);
        bsp_sync();

        assertvec(a9.BSPunsafe_access(), control_array1, bsp_pid(), __LINE__);

        if (bsp_pid() == 0) {
            endfun();
            std::cout
                    << "Test 17 : bsp_put(vector<T>, src_off, dst_off, len)"
                    << std::endl;
        }

```

```
        std::vector<int> substitution{0, 1, 2, 3, 4};
        for (auto& el: substitution) el += bsp_pid();
        std::vector<int> control_array2(base_array);
        std::copy_n(substitution.begin() + 1, 3, control_array2.begin() +
                                                  2);
        // {9, n+1, n+2, n+3, 5, 4, 3, 2, 1, 0}

        bsp_array<int> a11 = get_array(base_array);
        a11.bsp_put(substitution, 1, 2, 3);
        bsp_sync();

        assertvec(a11.BSPunsafe_access(), control_array2, bsp_pid(), __LINE__);

        if (bsp_pid() == 0) {
            endfun();
            std::cout
                    << "Test 18: bsp_put(bsp_array<T>, src_off, dst_off, len)"
                    << std::endl;
        }

        bsp_array<int> a12 = get_array(base_array);
        bsp_array<int> a13 = get_array(substitution);

        a12.bsp_put(a13, 1, 2, 3);
        bsp_sync();

        assertvec(a12.BSPunsafe_access(), control_array2, bsp_pid(), __LINE__);

        if (bsp_pid() == 0) {
            endfun();
            std::cout
                    << "Test 19 : bsp_put(vector<T>, dest, "
                    << "src_off, dst_off, len)"
                    << std::endl;
        }

        std::vector<int> substitution2{0, 1, 2, 3, 4};
        for (auto& el: substitution2) el += prevnode;
        std::vector<int> control_array3(base_array);
        std::copy_n(substitution2.begin() + 1, 3, control_array3.begin() + 2);

        bsp_array<int> a14 = get_array(base_array);
        a14.bsp_put(substitution, nextnode, 1, 2, 3);
        bsp_sync();

        assertvec(a14.BSPunsafe_access(), control_array3, bsp_pid(), __LINE__);

        if (bsp_pid() == 0) {
            endfun();
            std::cout
                    << "Test 20: bsp_put(bsp_array<T>, dest, "
                    << "src_off, dst_off, len)"
                    << std::endl;
        }

        bsp_array<int> a15 = get_array(base_array);
        bsp_array<int> a16 = get_array(substitution);

```

```
        a15.bsp_put(a16, nextnode, 1, 2, 3);
        bsp_sync();

        assertvec(a15.BSPunsafe_access(), control_array3, bsp_pid(), __LINE__);

        if (bsp_pid() == 0) {
            endfun();
            std::cout
                    << "Test 21 : bsp_get(int destination)"
                    << std::endl;
        }

        std::vector<int> fillpids(5, bsp_pid());
        bsp_array<int> a17 = get_array(fillpids);
        bsp_sync();

        a17.bsp_get(nextnode);
        bsp_sync();

        assertvec(a17.BSPunsafe_access(), std::vector<int>(5, nextnode),
                  bsp_pid(), __LINE__);

        if (bsp_pid() == 0) {
            endfun();
            std::cout
                    << "Test 22 : bsp_get(source, src_offs, dest_offs, length)"
                    << std::endl;
        }

        std::vector<int> base_arr2{0, 1, 2, 3, 4};
        for (auto& el: base_arr2) el += bsp_pid();
        std::vector<int> control_array4{nextnode + 1, nextnode + 2,
                                        nextnode + 3, bsp_pid() + 3,
                                        bsp_pid() + 4};
        bsp_array<int> a18 = get_array(base_arr2);
        bsp_sync();

        a18.bsp_get(nextnode, 1, 0, 3);
        bsp_sync();

        assertvec(a18.BSPunsafe_access(), control_array4, bsp_pid(), __LINE__);

        if (bsp_pid() == 0) {
            endfun();
            std::cout
                    << "PART 4: ARRAYS (PAR)"
                    << std::endl;
            std::cout
                    << "Test 23&24: bsp_put(T elem, int dest, int pos)"
                    << std::endl;
            std::cout << "Different positions" << std::endl;
        }

        bsp_array<int> a19 = get_array(base_array);

        a19.bsp_put(bsp_pid(), 0, bsp_pid());
        bsp_sync();

```

```
        auto compare_array5 = std::vector<int>{0, 1, 2, 6, 5, 4, 3, 2, 1, 0};
        if (bsp_pid() == 0) {
            assertvec(a19.BSPunsafe_access(), compare_array5, bsp_pid(),
                      __LINE__);
            endfun();
            std::cout << "Same position" << std::endl;
        }

        bsp_array<int> a20 = get_array(base_array);

        if (bsp_pid() != 0) a20.bsp_put(bsp_pid(), 0, 0);
        bsp_sync();

        if (bsp_pid() == 0) {
            assertint(a20.BSPunsafe_access().at(0), 1, 2, bsp_pid(), __LINE__);
            endfun();
            std::cout
                    << "Test 25&26 : bsp_put(bsp_variable<T>, dest, pos)"
                    << std::endl;
            std::cout << "Different positions" << std::endl;
        }

        bsp_array<int> a21 = get_array(base_array);
        bsp_variable<int> v11 = get_variable(bsp_pid());

        a21.bsp_put(v11, 0, bsp_pid());
        bsp_sync();

        if (bsp_pid() == 0) {
            assertvec(a21.BSPunsafe_access(), compare_array5, bsp_pid(),
                      __LINE__);
            endfun();
            std::cout << "Same position" << std::endl;
        }

        bsp_array<int> a22 = get_array(base_array);
        bsp_variable<int> v12 = get_variable(bsp_pid());

        if (bsp_pid() != 0) a22.bsp_put(v12, 0, 0);
        bsp_sync();

        if (bsp_pid() == 0) {
            assertint(a22.BSPunsafe_access().at(0), 1, 2, bsp_pid(), __LINE__);
            endfun();
            std::cout
                    << "Test 27 : bsp_put(std::vector<T> other, int dest)"
                    << std::endl;
        }

        bsp_array<int> a23 = get_empty_array<int>(5);

        if (bsp_pid() != 0) a23.bsp_put(std::vector<int>(5, bsp_pid()), 0);
        bsp_sync();

        if (bsp_pid() == 0) {
            assertvec(a23.BSPunsafe_access(), std::vector<int>(5, 1),
                      std::vector<int>(5, 2), bsp_pid(), __LINE__);
            endfun();
```

```
        std::cout
                << "Test 28 : bsp_put(bsp_array<T> other, int dest)"
                << std::endl;
    }

    bsp_array<int> a24 = get_empty_array<int>(5);
    bsp_array<int> a25 = get_array(std::vector<int>(5, bsp_pid()));

    if (bsp_pid() != 0) a24.bsp_put(a25, 0);
    bsp_sync();

    if (bsp_pid() == 0) {
        assertvec(a24.BSPunsafe_access(), std::vector<int>(5, 1),
                  std::vector<int>(5, 2), bsp_pid(), __LINE__);
        endfun();
        std::cout
                << "Test 29&30: bsp_put(vector<T>, dest, "
                << "src_off, dst_off, len)"
                << std::endl;
        std::cout << "Separate positions" << std::endl;
    }

    const std::vector<int>& confront1{9, 1, 2, 3, 5, 4, 3, 98, 99, 0};
    const std::vector<int>& subst1{0, 1, 2, 3, 4, 5};
    const std::vector<int>& subst2{95, 96, 97, 98, 99};

    bsp_array<int> a26 = get_array(base_array);

    if (bsp_pid() == 1) a26.bsp_put(subst1, 0, 1, 1, 3);
    if (bsp_pid() == 2) a26.bsp_put(subst2, 0, 3, 7, 2);
    bsp_sync();

    if (bsp_pid() == 0) {
        assertvec(a26.BSPunsafe_access(), confront1, bsp_pid(), __LINE__);
        endfun();
        std::cout << "Overlapping positions" << std::endl;
    }

    const std::vector<int>& confront2{9, 8, 7, 1, 2, 3, 4, 98, 99, 0};
    const std::vector<int>& confront3{9, 8, 7, 1, 2, 3, 97, 98, 99, 0};

    bsp_array<int> a27 = get_array(base_array);

    if (bsp_pid() == 1) a27.bsp_put(subst1, 0, 1, 3, 4);
    if (bsp_pid() == 2) a27.bsp_put(subst2, 0, 2, 6, 3);
    bsp_sync();

    if (bsp_pid() == 0) {
        assertvec(a27.BSPunsafe_access(), confront2, confront3, bsp_pid(),
                  __LINE__);
        endfun();
        std::cout
                << "Test 31&32 : bsp_put(bsp_array<T>, dest, "
                << "src_off, dst_off, len)"
                << std::endl;
        std::cout << "Separate positions" << std::endl;
    }

```

```
        bsp_array<int> a28 = get_array(base_array);
        bsp_array<int> a29 = get_array(bsp_pid() > 1 ? subst2 : subst1);

        if (bsp_pid() == 1) a28.bsp_put(a29, 0, 1, 1, 3);
        if (bsp_pid() == 2) a28.bsp_put(a29, 0, 3, 7, 2);
        bsp_sync();

        if (bsp_pid() == 0) {
            assertvec(a28.BSPunsafe_access(), confront1, bsp_pid(), __LINE__);
            endfun();
            std::cout << "Overlapping positions" << std::endl;
        }

        bsp_array<int> a30 = get_array(base_array);
        bsp_array<int> a31 = get_array(bsp_pid() > 1 ? subst2 : subst1);

        if (bsp_pid() == 1) a30.bsp_put(a31, 0, 1, 3, 4);
        if (bsp_pid() == 2) a30.bsp_put(a31, 0, 2, 6, 3);
        bsp_sync();

        if (bsp_pid() == 0) {
            assertvec(a30.BSPunsafe_access(), confront2, confront3, bsp_pid(),
                      __LINE__);
            endfun();
            std::cout
                    << "PART 5 : DIRECT GETS"
                    << std::endl;
            std::cout
                    << "Test 33 (var) : T bsp_direct_get(int source)"
                    << std::endl;
        }

        bsp_variable<int> v13 = get_variable(bsp_pid());
        bsp_sync();

        int dg1 = v13.bsp_direct_get(0);
        v13.bsp_put(bsp_pid(), 0);
        bsp_sync();

        assertint(dg1, 0, bsp_pid(), __LINE__);

        if (bsp_pid() == 0) {
            endfun();
            std::cout
                    << "Test 34 (arr): T bsp_direct_get(int source, int pos)"
                    << std::endl;
        }

        bsp_array<int> a32 = get_array(base_array);
        bsp_sync();

        int dg2 = a32.bsp_direct_get(0, 3);
        a32.bsp_put(bsp_pid(), 0, 3);
        bsp_sync();

        assertint(dg2, 6, bsp_pid(), __LINE__);

        if (bsp_pid() == 0) {
```

```
            endfun();
            std::cout
                    << "Test 35 (arr) : vector<T> bsp_direct_get(int source)"
                    << std::endl;
        }

        bsp_array<int> a33 = get_array(fillpids);
        bsp_sync();

        auto dg3 = a33.bsp_direct_get(0);
        a33.bsp_put(std::vector<int>(5, bsp_pid()));
        bsp_sync();

        assertvec(dg3, std::vector<int>(5, 0), bsp_pid(), __LINE__);
        if (bsp_pid() == 0) {
            endfun();
            std::cout
                    << "TEST ENDED"
                    << std::endl;
            endfun(true);
        }
    }

};

int main() {
    std::vector<std::unique_ptr<bsp_node>> nodes;
    for (size_t i{0}; i < 3; ++i) {
        nodes.push_back(std::make_unique<unit_tests>());
    }
    bsp_program mbsp(std::move(nodes));
    mbsp.start();
}
```

B.5 Java programs

BSPpsrs.java

```
import com.multicorebsp.core.*;
import org.apache.commons.math3.random.MersenneTwister;
import org.apache.commons.math3.util.MathArrays;

import java.util.*;
import java.util.stream.IntStream;

public class BSPpsrs extends BSP_PROGRAM {

    private int n;
    private int n_procs;
    private int seed;

    @Override
    protected void main_part() throws InterruptedException {
        try {
            bsp_begin(n_procs);
        } catch (IllegalAccessException | InstantiationException |
                EmptyQueueException | InterruptedException e) {
```

```java
            e.printStackTrace();
        }
    }

    @Override
    protected void parallel_part() throws InterruptedException,
            IllegalAccessException, EmptyQueueException {
        BSP_INTEGER problem_size = new BSP_INTEGER(this, 0);
        if (bsp_pid() == 0) {
            for (int i = 0; i < bsp_nprocs(); ++i) {
                problem_size.bsp_put(n, i);
            }
        }
        bsp_sync();
        n = problem_size.read();

        int p = bsp_nprocs();
        int s = bsp_pid();
        long t = System.currentTimeMillis();
        int numels = n / p;
        BSP_INT_ARRAY to_sort = new BSP_INT_ARRAY(this, numels);

        if (s == 0) {
            MersenneTwister mtw;
            if (seed == -1) {
                mtw = new MersenneTwister();
            } else {
                mtw = new MersenneTwister(seed);
            }
            int[] data = IntStream.range(0, n).toArray();
            MathArrays.shuffle(data, mtw);
            long t2 = System.currentTimeMillis();
            System.out.println("Ended vector generation and shuffling (spent "
                    + (t2 - t) + " ms)");
            System.out.println("Starting parallel part");

            t = System.currentTimeMillis();
            int count = 0;
            for (int i = 0; i < n; i += numels) {
                int last = Math.min(n, i + numels);
                to_sort.bsp_put(data, i, count++, 0, last - i);
            }
        }
        bsp_sync();

        int[] vec = to_sort.getData();

        Arrays.sort(vec);

        ArrayList<Integer> primary_samples = new ArrayList<>();
        int samplesize = vec.length / p;
        int i;
        for (i = 0; i < vec.length; i += samplesize) {
            primary_samples.add(vec[i]);
        }

        if (i != vec.length - 1) {
            primary_samples.add(vec[i - 1]);
```

```
        }

        IntArrayList psam = new IntArrayList();
        psam.array.addAll(primary_samples);

        BSP_ARRAY<IntArrayList> ps_array = new BSP_ARRAY<>(this, p, psam);
        for (i = 0; i < p; ++i) {
            ps_array.bsp_put(psam, i, s);
        }
        bsp_sync();

        ArrayList<Integer> ps_all = new ArrayList<>();
        ArrayList<Integer> secondary_samples = new ArrayList<>();
        try {
            for (i = 0; i < p; ++i) {
                ps_all.addAll(ps_array.read(i).array);
            }
            Collections.sort(ps_all);
            samplesize = ps_all.size() / p;
        } catch (NullPointerException e) {
            e.printStackTrace();
        }

        for (i = 0; i < ps_all.size(); i += samplesize) {
            secondary_samples.add(ps_all.get(i));
        }

        if (i == ps_all.size())
            secondary_samples.add(ps_all.get(i - 1));

        int upperbound;
        int count = 1;

        do {
            upperbound = secondary_samples.get(count++);
        } while (vec[0] > upperbound);

        count--;
        ArrayList<Integer> temp = new ArrayList<>();

        BSP_ARRAY<IntArrayList> portion =
                    new BSP_ARRAY<>(this, p, new IntArrayList());

        for (i = 0; i < vec.length; ++i) {
            if (vec[i] > upperbound) {
                portion.bsp_put(new IntArrayList(temp), count - 1, s);
                temp.clear();
                upperbound = secondary_samples.get(++count);
            }
            temp.add(vec[i]);
        }

        portion.bsp_put(new IntArrayList(temp), count - 1, s);
        bsp_sync();

        ArrayList<Integer> secondary_block = new ArrayList<>();
        BSP_ARRAY<IntArrayList> final_arr = new BSP_ARRAY<>(this, p,
                new IntArrayList());
```

```java
        for (IntArrayList pbl : portion) {
            secondary_block.addAll(pbl.array);
        }

        Collections.sort(secondary_block);
        final_arr.bsp_put(new IntArrayList(secondary_block), 0, s);
        bsp_sync();

        if (s == 0) {
            ArrayList<Integer> conclusion = new ArrayList<>();
            for (IntArrayList el : final_arr)
                conclusion.addAll(el.array);

            boolean passed = true;
            for (int j = 0; j < n; ++j) {
                if (conclusion.get(j) != j) {
                    passed = false;
                    break;
                }
            }

            System.out.println("Check " + (passed ? "passed" : "failed"));
            long t1 = System.currentTimeMillis();
            System.out.println("Parallel part took " + (t1 - t) + " ms.");
        }

        bsp_sync();
    }

    public static void main(String[] args) {
        if (args.length < 2) {
            System.err.println("Usage: BSPpsrs <N> <P> (seed)");
            System.err.println(" where N is the problem size (power of 2)");
            System.err.println("       P is the number of threads (power of 2)");
            System.err.println("       N must be >= P^3");
            System.err.println("       seed is an optional seed " +
                    "for permutations (leave blank to randomize it");
            System.exit(1);
        }
        int n, p, s;
        n = Integer.parseInt(args[0]);
        p = Integer.parseInt(args[1]);
        if (n < p * p * p) {
            System.err.println("N must be >= P^3");
            System.exit(1);
        }
        s = (args.length >= 3 ? Integer.parseInt(args[2]) : -1);
        BSPpsrs my_computation = new BSPpsrs();
        my_computation.n = n;
        my_computation.n_procs = p;
        my_computation.seed = s;
        my_computation.start();
    }
}
```

IntArrayList.java

```java
import com.multicorebsp.core.CompulsaryCloneable;

import java.util.ArrayList;

public class IntArrayList implements CompulsaryCloneable<IntArrayList> {

    public ArrayList<Integer> array;

    public IntArrayList() {
        array = new ArrayList<>();
    }
    public IntArrayList(ArrayList<Integer> other) {
        array = new ArrayList<>(other);
    }

    @Override
    public Object clone() throws CloneNotSupportedException {
        super.clone();
        return safeClone();
    }

    @Override
    public IntArrayList safeClone() {
        return new IntArrayList(array);
    }
}
```

SequentialInprod.java

```java
public class SequentialInprod {

    public static void main(String[] args) {
        if (args.length < 1) System.exit(1);

        int n = 1;

        try {
            n = Integer.parseInt(args[0]);
        } catch (NumberFormatException e) {
            System.err.println("Argument must be an integer");
            System.exit(-1);
        }

        double[] x = new double[n];

        for (int i = 0; i < n; ++i) {
            x[i] = i + 1;
        }

        double inprod = 0.0;
        long t1 = System.currentTimeMillis();

        for (int i = 0; i < n; ++i) {
            inprod += x[i] * x[i];
        }
```

```
        long t2 = System.currentTimeMillis();

        System.out.println("Processor 0: sum of squares up to " + n + "*"
                    + n + " is " + inprod);
        System.out.println("Processor 0: local time taken is " + (t2 - t1));

        double dn = (double) n;
        dn *= n + 1.0;
        dn *= 2.0 * n + 1.0;
        dn /= 6.0;
        System.out.println("Checksum: " + dn);

        System.exit(0);
    }

}
```

SequentialSort.java

```
import java.util.*;
import java.util.stream.Collectors;

public class SequentialSort {

    private static void arrshuffle(int[] array, Random rd) {
        for (int i = array.length - 1; i > 0; i--) {
            int index = rd.nextInt(i + 1);
            // Simple swap
            int a = array[index];
            array[index] = array[i];
            array[i] = a;
        }
    }

    public static void main(String[] args) {
        if (args.length < 1) {
            System.err.println("Usage: seqsort <N> (seed)");
            System.err.println(" where N is the problem size (power of 2)");
            System.err.println("       seed is an optional seed for " +
                    "permutations (leave blank to randomize it)");
            System.exit(1);
        }
        int n = 1, s = -1;
        try {
            n = Integer.parseInt(args[0]);
        } catch (NumberFormatException e) {
            System.err.println("First argument must be an integer");
            System.exit(-1);
        }
        if (args.length >= 2) {
            try {
                s = Integer.parseInt(args[1]);
            } catch (NumberFormatException e) {
                System.err.println("Second argument must be an integer");
                System.exit(-1);
            }
        }
```

```
        long t = System.currentTimeMillis();
        Random rd = new Random();
        if (s == -1) {
            s = rd.nextInt();
        }
        rd.setSeed(s);
        int[] data = new int[n];
        for (int i = 0; i < n; ++i) {
            data[i] = i;
        }

        arrshuffle(data, rd);

        List<Integer> list = Arrays.stream(data).boxed()
                .collect(Collectors.toList());

        long t1 = System.currentTimeMillis();
        System.out.println("Ended vector generation and shuffling (spent " +
                (t1 - t) + " ms)");
        System.out.println("starting sequential part");

        long t2 = System.currentTimeMillis();

        Arrays.sort(data);

        long t3 = System.currentTimeMillis();

        Collections.sort(list);

        long t4 = System.currentTimeMillis();

        boolean passed = true;
        for (int j = 0; j < n; j++) {
            if (data[j] != j) {
                passed = false;
                break;
            }
        }

        System.out.println("Check " + (passed ? "passed" : "failed"));
        System.out.println("Parallel part took " + (t3 - t2) + " ms.");
        System.out.println("Collections part took " + (t4 - t3) + " ms.");
    }
}
```

SequentialFFT.java

```
import java.util.Arrays;

public class SequentialFFT {

    private final static double PI = 3.141592653589793;

    private static void ufft(double[] x, int offset, int n, boolean sign,
                             double[] w) {
        for (int k = 2; k <= n; k *= 2) {
            int nk = n / k;
```

```
                for (int r = 0; r < nk; ++r) {
                    int rk = 2 * r * k;
                    for (int j = 0; j < k; j += 2) {
                        double wr = w[j * nk];
                        double wi;
                        if (sign) {
                            wi = w[j * nk + 1];
                        } else {
                            wi = -w[j * nk + 1];
                        }

                        int j0 = rk + j + offset;
                        int j1 = j0 + 1;
                        int j2 = j0 + k;
                        int j3 = j2 + 1;

                        double taur = wr * x[j2] - wi * x[j3];
                        double taui = wi * x[j2] + wr * x[j3];

                        x[j2] = x[j0] - taur;
                        x[j3] = x[j1] - taui;
                        x[j0] += taur;
                        x[j1] += taui;
                    }
                }
            }
        }

        private static void ufft_init(int n, double[] w) {
            assert (w.length == n);

            if (n == 1) return;

            w[0] = 1.0;
            w[1] = 0.0;

            if (n == 4) {
                w[2] = 0.0;
                w[3] = -1.0;
            } else if (n >= 8) {
                double theta = -2.0 * PI / (double) (n);
                for (int j = 1; j <= n / 8; j++) {
                    w[2 * j] = Math.cos(j * theta);
                    w[2 * j + 1] = Math.sin(j * theta);
                }
                for (int j = 0; j < n / 8; j++) {
                    int n4j = n / 4 - j;
                    w[2 * n4j] = -w[2 * j + 1];
                    w[2 * n4j + 1] = -w[2 * j];
                }
                for (int j = 1; j < n / 4; j++) {
                    int n2j = n / 2 - j;
                    w[2 * n2j] = -w[2 * j];
                    w[2 * n2j + 1] = w[2 * j + 1];
                }
            }
        }
```

```
    private static void twiddle_init(int n, double alpha, int[] rho,
                                     double[] w, int offset) {
        double theta = -2.0 * PI * alpha / (double) (n);
        for (int j = 0; j < n; ++j) {
            double rt = (double) (rho[j]) * theta;
            w[offset + 2 * j] = Math.cos(rt);
            w[offset + 2 * j + 1] = Math.sin(rt);
        }
    }

    private static void permute(double[] x, int n, int[] sigma) {
        assert (x.length / 2 == sigma.length);

        for (int j = 0; j < n; ++j) {
            if (j < sigma[j]) {
                int j0 = 2 * j;
                int j1 = j0 + 1;
                int j2 = 2 * sigma[j];
                int j3 = j2 + 1;
                double tmpr = x[j0];
                double tmpi = x[j1];
                x[j0] = x[j2];
                x[j1] = x[j3];
                x[j2] = tmpr;
                x[j3] = tmpi;
            }
        }
    }

    private static void bitrev_init(int[] rho) {
        int n = rho.length;

        int binary_len = (int) (Math.ceil(Math.log((double)(n))/Math.log(2.0)));
        boolean[] bits = new boolean[binary_len];
        int[] pwrs = new int[binary_len];
        pwrs[0] = 1;
        for (int j = 1; j < binary_len; ++j) {
            pwrs[j] = pwrs[j - 1] * 2;
        }
        int j = 0;
        while (j < n - 1) {
            j++;
            int lastbit = 0;
            while (bits[lastbit]) {
                bits[lastbit] = false;
                lastbit++;
            }
            bits[lastbit] = true;
            int val = 0;
            for (int k = 0; k < binary_len; ++k) {
                if (bits[k]) {
                    val += pwrs[binary_len - k - 1];
                }
            }
            rho[j] = val;
        }
    }
```

```java
    private static void fft_init(int n, double[] w,
                                 double[] tw, int[] rho_np) {
        bitrev_init(rho_np);
        ufft_init(n, w);

        twiddle_init(n, 0, rho_np, tw, 0);
    }

    private static void calcError(double[] xlocal, double[] xarr) {
        double error = 0.0;
        for (int c = 0; c < xlocal.length; c++) {
            double lerror = Math.abs(xlocal[c] - xarr[c]);
            error += lerror;
        }
        System.out.println("local error is " + (error/(double)(xlocal.length)));
    }

    public static void main(String[] args) {
        if (args.length < 1) {
            System.err.println("Usage: seqfft <N>");
            System.err.println("where N is the problem size (power of 2)");
            System.exit(-1);
        }
        int n = 1;
        try {
            n = Integer.parseInt(args[0]);
            if (!(n > 0 && (n & n - 1) == 0))
                throw new NumberFormatException("not power of 2");
        } catch (NumberFormatException e) {
            System.err.println("Argument must be an integer (power of 2)");
            System.exit(-1);
        }
        double[] data = new double[n];

        n /= 2;
        for (int i = 0; i < n; i += 2) {
            data[i] = (double) (i) / 2.0;
        }

        double[] old = Arrays.copyOf(data, data.length);

        double[] w = new double[n];
        double[] tw = new double[(2 * n)];
        int[] rho_np = new int[(n)];

        long t = System.currentTimeMillis();
        fft_init(n, w, tw, rho_np);
        // forward fft
        permute(data, n, rho_np);
        ufft(data, 0, n, true, w);

        // inverse fft
        permute(data, n, rho_np);
        ufft(data, 0, n, false, w);

        double ninv = 1.0 / (double) (n);
        for (int j = 0; j < 2 * n; ++j) {
            data[j] *= ninv;
```

```
        }

        calcError(data, old);
        long t1 = System.currentTimeMillis();


        System.out.println("Sequential part took " + (t1 - t) + " ms.");

        calcError(data, old);


        System.exit(0);
    }
}
```

SequentialLU_commons.java

```
import org.apache.commons.math3.linear.LUDecomposition;
import org.apache.commons.math3.linear.MatrixUtils;
import org.apache.commons.math3.linear.RealMatrix;

import java.util.Random;

public class SequentialLU_commons {

    public static void main(String[] args) {
        if (args.length < 1) {
            System.err.println("Usage: seqlu <n>");
            System.err.println("\twill stat LU decomposition on a " +
                    "n by n matrix.");
            System.exit(1);
        }

        int n = 1;
        try {
            n = Integer.parseInt(args[0]);
        } catch (NumberFormatException e) {
            System.err.println("Argument must be an integer");
            System.exit(-1);
        }

        Random rd = new Random();

        double[][] a = new double[n][n];
        double[][] l = new double[n][n];
        double[][] u = new double[n][n];

        for (int i = 0; i < n; ++i) {
            for (int j = 0; j < n; ++j) {
                double val = Math.random();
                a[i][j] = val;
                l[i][j] = u[i][j] = 0.0;
            }
        }

        RealMatrix A = MatrixUtils.createRealMatrix(a);

```

```
        long t1 = System.currentTimeMillis();
        LUDecomposition lucalc = new LUDecomposition(A);
        l = lucalc.getL().getData();
        u = lucalc.getU().getData();
        long t2 = System.currentTimeMillis();

        System.out.println("Sequential computation took " + (t2 - t1) + " ms.");
        System.exit(0);
    }
}
```

Bibliography

[1] S.G. Akl and W. Rheinboldt. *Parallel Sorting Algorithms*. Notes and reports in computer science and applied mathematics. Elsevier Science, 2014. ISBN: 9781483268088. URL: https://books.google.it/books?id=jhHjBQAAQBAJ.

[2] Marco Aldinucci, Salvatore Ruggieri, and Massimo Torquati. "Porting decision tree algorithms to multicore using FastFlow". In: *Joint European Conference on Machine Learning and Knowledge Discovery in Databases*. Springer. 2010, pp. 7–23.

[3] Marco Aldinucci et al. "Fastflow: high-level and efficient streaming on multi-core". In: *Programming multi-core and many-core computing systems, parallel and distributed computing* (2017).

[4] Marco Aldinucci et al. "Targeting distributed systems in fastflow". In: *European Conference on Parallel Processing*. Springer. 2012, pp. 47–56.

[5] B. Andrist and V. Sehr. *C++ High Performance: Boost and optimize the performance of your C++17 code*. Packt Publishing, 2018. ISBN: 9781787124776. URL: https://books.google.it/books?id=EyBKDwAAQBAJ.

[6] *Apache Giraph*. URL: https://giraph.apache.org/ (visited on 04/20/2020).

[7] *Apache Hama*. URL: https://hama.apache.org/ (visited on 04/20/2020).

[8] R.H. Bisseling. *Author's page on the University of Utrecht website*. URL: http://www.staff.science.uu.nl/~bisse101/Software/software.html (visited on 04/21/2020).

[9] R.H. Bisseling. *Parallel Scientific Computation: A Structured Approach Using BSP and MPI*. Oxford University Press, 2004. ISBN: 9780198529392. URL: https://books.google.it/books?id=iCsTDAAAQBAJ.

[10] Dina Bitton et al. "A taxonomy of parallel sorting". In: *ACM Computing Surveys (CSUR)* 16.3 (1984), pp. 287–318.

[11] Å. Björck. *Numerical Methods in Matrix Computations*. Texts in Applied Mathematics. Springer International Publishing, 2014. ISBN: 9783319050898. URL: https://books.google.it/books?id=joO8BAAAQBAJ.

[12] Olaf Bonorden et al. "The paderborn university BSP (PUB) library". In: *Parallel Computing* 29.2 (2003), pp. 187–207.

[13] Jeff Bonwick. "The Slab Allocator: An Object-Caching Kernel Memory Allocator". In: *Proceedings of the USENIX Summer 1994 Technical Conference on USENIX Summer 1994 Technical Conference - Volume 1*. USTC'94. Boston, Massachusetts: USENIX Association, 1994, p. 6.

[14] *BSPlib Home Page*. URL: `http://www.bsp-worldwide.org/` (visited on 04/23/2020).

[15] Rainer Buchty et al. "A survey on hardware-aware and heterogeneous computing on multicore processors and accelerators". In: *Concurrency and Computation: Practice and Experience* 24.7 (2012), pp. 663–675. DOI: `10.1002/cpe.1904`. eprint: `https://onlinelibrary.wiley.com/doi/pdf/10.1002/cpe.1904`. URL: `https://onlinelibrary.wiley.com/doi/abs/10.1002/cpe.1904`.

[16] Jan-Willem Buurlage, Tom Bannink, and Rob H. Bisseling. "Bulk: A Modern C++ Interface for Bulk-Synchronous Parallel Programs". In: *Euro-Par 2018: Parallel Processing*. Ed. by Marco Aldinucci, Luca Padovani, and Massimo Torquati. Cham: Springer International Publishing, 2018, pp. 519–532. ISBN: 978-3-319-96983-1.

[17] Avery Ching et al. "One trillion edges: Graph processing at facebook-scale". In: *Proceedings of the VLDB Endowment* 8.12 (2015), pp. 1804–1815.

[18] James W Cooley and John W Tukey. "An algorithm for the machine calculation of complex Fourier series". In: *Mathematics of computation* 19.90 (1965), pp. 297–301.

[19] *FastFlow commit `8b9105d`*. URL: `https://github.com/fastflow/fastflow/commit/8b9105de8784335fc2fcf28c3ba955cd68dfed5b` (visited on 04/22/2020).

[20] *FastFlow official repository on GitHub*. URL: `https://github.com/fastflow/fastflow` (visited on 04/22/2020).

[21] *FastFlow online tutorial*. URL: `http://calvados.di.unipi.it/dokuwiki/doku.php/ffnamespace:tutorial` (visited on 04/23/2020).

[22] Alexandros V Gerbessiotis. "Extending the BSP model for multi-core and out-of-core computing: MBSP". In: *Parallel Computing* 41 (2015), pp. 90–102.

[23] Alexandros V. Gerbessiotis and Constantinos J. Siniolakis. *Efficient Deterministic Sorting on the BSP Model*. Tech. rep. PARALLEL PROCESSING LETTERS, 1996.

[24] Michael T Goodrich. "Communication-efficient parallel sorting". In: *SIAM Journal on Computing* 29.2 (1999), pp. 416–432.

[25] Khaled Hamidouche, Joel Falcou, and Daniel Etiemble. "Hybrid Bulk Synchronous Parallelism Library for Clustered Smp Architectures". In: *Proceedings of the Fourth International Workshop on High-Level Parallel Programming and Applications*. HLPP '10. Baltimore, Maryland, USA: Association for Computing Machinery, 2010, pp. 55–62. ISBN: 9781450302548. DOI: `10.1145/1863482.1863494`. URL: `https://doi.org/10.1145/1863482.1863494`.

[26] Thomas Heller et al. "Hpx–an open source c++ standard library for parallelism and concurrency". In: *Proceedings of OpenSuCo* (2017), p. 5.

[27] Jonathan MD Hill et al. "BSPlib: The BSP programming library". In: *Parallel Computing* 24.14 (1998), pp. 1947–1980.

[28] *Intel Xeon Phi x200 Family specifications*. URL: `https://ark.intel.com/content/www/us/en/ark/products/series/92650/intel-xeon-phi-x200-product-family.html` (visited on 04/22/2020).

[29] Peter Kankowski. *Hash Functions: An Empirical Comparison*. 2009. URL: https://www.codeproject.com/Articles/32829/Hash-Functions-An-Empirical-Comparison.

[30] Richard M. Karp. *A Survey of Parallel Algorithms for Shared-Memory Machines*. Tech. rep. USA, 1988.

[31] David Lecomber. *An Object-Oriented Programming Model for BSP Computations*. Tech. rep. In Proc. PPECC Workshop on Parallel and Distributed Computing, 1994.

[32] Xiaobo Li et al. "On the versatility of parallel sorting by regular sampling". In: *Parallel Computing* 19.10 (1993), pp. 1079–1103.

[33] Frédéric Loulergue, Frédéric Gava, and David Billiet. "Bulk synchronous parallel ML: modular implementation and performance prediction". In: *International Conference on Computational Science*. Springer. 2005, pp. 1046–1054.

[34] *LU decomposition documentation for the ALGLIB library*. URL: https://www.alglib.net/matrixops/lu.php (visited on 04/22/2020).

[35] *LU decomposition documentation for the Apache Commons Java library*. URL: http://commons.apache.org/proper/commons-math/javadocs/api-3.6/org/apache/commons/math3/linear/LUDecomposition.html (visited on 04/22/2020).

[36] Grzegorz Malewicz et al. "Pregel: a system for large-scale graph processing". In: *Proceedings of the 2010 international conference on Management of data*. New York, NY, USA, 2010, pp. 135–146. URL: http://doi.acm.org/10.1145/1807167.1807184.

[37] Michael McCool, James Reinders, and Arch Robison. *Structured parallel programming: patterns for efficient computation*. Elsevier, 2012.

[38] *MulticoreBSP for C*. URL: http://www.multicorebsp.com/download/c/ (visited on 04/20/2020).

[39] *MulticoreBSP for Java*. URL: http://www.multicorebsp.com/download/java/ (visited on 04/20/2020).

[40] *MulticoreBSP for Java documentation*. URL: http://www.multicorebsp.com/javadoc/ (visited on 04/16/2020).

[41] *Repository for the library implemented as part of this thesis.* URL: https://gitlab.com/leombro/master-thesis (visited on 04/16/2020).

[42] Luis FG Sarmenta. "An adaptive, fault-tolerant implementation of BSP for Java-based volunteer computing systems". In: *International Parallel Processing Symposium*. Springer. 1999, pp. 763–780.

[43] Hideyuki Shamoto et al. "Large-scale distributed sorting for GPU-based heterogeneous supercomputers". In: *2014 IEEE International Conference on Big Data (Big Data)*. IEEE. 2014, pp. 510–518.

[44] R. Shams et al. "A Survey of Medical Image Registration on Multicore and the GPU". In: *IEEE Signal Processing Magazine* 27.2 (2010), pp. 50–60.

[45] R. Shaposhnik, C. Martella, and D. Logothetis. *Practical Graph Analytics with Apache Giraph*. Apress, 2015. ISBN: 9781484212516. URL: https://books.google.it/books?id=FWb%5C_CgAAQBAJ.

[46] Hanmao Shi and Jonathan Schaeffer. "Parallel sorting by regular sampling". In: *Journal of parallel and distributed computing* 14.4 (1992), pp. 361–372.

[47] Kamran Siddique, Zahid Akhtar, and Yangwoo Kim. "Researching Apache Hama: A Pure BSP Computing Framework". In: *Advanced Multimedia and Ubiquitous Engineering*. Ed. by James J. (Jong Hyuk) Park et al. Springer Singapore, 2016, pp. 215–221. ISBN: 978-981-10-1536-6.

[48] Dhirendra Pratap Singh, Ishan Joshi, and Jaytrilok Choudhary. "Survey of GPU Based Sorting Algorithms". In: *Int. J. Parallel Program.* 46.6 (Dec. 2018), pp. 1017–1034. ISSN: 0885-7458. DOI: `10.1007/s10766-017-0502-5`. URL: `https://doi.org/10.1007/s10766-017-0502-5`.

[49] *`std::allocate_shared` documentation*. URL: `https://en.cppreference.com/w/cpp/memory/shared_ptr/allocate_shared` (visited on 04/16/2020).

[50] Maarten van Steen and Andrew S. Tanenbaum. *Distributed Systems*. 3rd ed. 2017. URL: `https://www.distributed-systems.net/`.

[51] Bjarne Stroustrup. "What is object-oriented programming?" In: *IEEE software* 5.3 (1988), pp. 10–20.

[52] *Ten and Now: Almost 10 Years of Intel CPUs compared*. URL: `https://www.techspot.com/article/1039-ten-years-intel-cpu-compared/` (visited on 04/22/2020).

[53] *The `djb2` hash function*. 1991. URL: `https://groups.google.com/forum/#!msg/comp.lang.c/lSKWXiuNOAk/zstZ3SRhCjgJ` (visited on 04/15/2020).

[54] *The Doolittle Algorithm*. URL: `https://vismor.com/documents/network_analysis/matrix_algorithms/S4.SS2.php` (visited on 04/22/2020).

[55] Alexandre Tiskin. "The design and analysis of bulk-synchronous parallel algorithms". PhD thesis. 1998.

[56] Massimo Torquati. *Introduction to FastFlow programming: allocators*. URL: `http://didawiki.di.unipi.it/lib/exe/fetch.php/magistraleinformaticanetworking/spm/spm1617dic15-2.pdf` (visited on 04/16/2020).

[57] Massimo Torquati. "Single-Producer/Single-Consumer Queues on Shared Cache Multi-Core Systems". In: *CoRR* abs/1012.1824 (2010). arXiv: `1012.1824`. URL: `http://arxiv.org/abs/1012.1824`.

[58] Matthew Travers. "Cpu power consumption experiments and results analysis of intel i7-4820k". In: (2015).

[59] L. G. Valiant. "General Purpose Parallel Architectures". In: *Handbook of Theoretical Computer Science (Vol. A): Algorithms and Complexity*. Cambridge, MA, USA: MIT Press, 1991, pp. 943–973. ISBN: 0444880712.

[60] Leslie G Valiant. "A bridging model for multi-core computing". In: *Journal of Computer and System Sciences* 77.1 (2011), pp. 154–166.

[61] Leslie G. Valiant. "A Bridging Model for Parallel Computation". In: *Commun. ACM* 33.8 (Aug. 1990), pp. 103–111. ISSN: 0001-0782. DOI: `10.1145/79173.79181`. URL: `https://doi.org/10.1145/79173.79181`.

[62] *Wikipedia article on Daniel J. Bernstein*. URL: `https://en.wikipedia.org/wiki/Daniel_J._Bernstein` (visited on 04/15/2020).

[63] A. Williams. *C++ Concurrency in Action: Practical Multithreading*. Manning Pubs Co Series. Manning, 2012. ISBN: 9781933988771. URL: `https://books.google.it/books?id=EttPPgAACAAJ`.

[64] J. C. Wyllie. "The Complexity of Parallel Computations". PhD thesis. Computer Science Department, Conell University, Ithaca, NY, 1981.

[65] Yangsuk Kee and Soonhoi Ha. "xBSP: an efficient BSP implementation for cLAN". In: *Proceedings First IEEE/ACM International Symposium on Cluster Computing and the Grid*. 2001, pp. 237–244.

[66] A.N. Yzelman and Rob H. Bisseling. "An Object-Oriented Bulk Synchronous Parallel Library for Multicore Programming". In: *Concurr. Comput. : Pract. Exper.* 24.5 (Apr. 2012), pp. 533–553. ISSN: 1532-0626. DOI: `10.1002/cpe.1843`. URL: `https://doi.org/10.1002/cpe.1843`.

[67] Albert-Jan Yzelman et al. "MulticoreBSP for C: A High-Performance Library for Shared-Memory Parallel Programming". In: *International Journal of Parallel Programming* 42 (Aug. 2014). DOI: `10.1007/s10766-013-0262-9`.