Chapter 2

A modern interface for BSP programs

The bulk-synchronous parallel (BSP) model was introduced as a bridging model for parallel programming by Valiant in 1989 [Val90]. It enables a way to structure parallel computations, which aids in the design and analysis of parallel programs.

The BSP model defines an abstract computer, the BSP computer, on which BSP algorithms can run. Such a computer consists of p identical processors, each having access to their own local memory. A communication network is available which can be used by the different processors to communicate data. During the execution of an algorithm, there are points at which bulk synchronizations are performed. The time of such a synchronization, the *latency*, is denoted by l. The communication cost per data word is denoted by g. The parameters l and g are usually expressed in number of floating-point operations (FLOPs). They can be related to *wall-clock time* by considering the computation rate r of the individual processors which is measured in floating-point operations per second (FLOP/s). A BSP computer is captured completely by the parameter tuple (p, g, l, r).

This chapter is based on:

Bulk: A Modern C++ Interface for Bulk-Synchronous Parallel Programs. *JW Buurlage, TR Bannink, RH Bisseling*. European Conference on Parallel Processing, 519-532, 2018

At a high level, a BSP algorithm is a series of supersteps that each consist of a *computation phase* and a *communication phase*. The processors of a BSP computer can simultaneously send and receive data, and they can do so independently. This means that the cost of communication is dominated by the maximum number of words sent or received by any processor. At the end of each superstep a bulk synchronization is performed ensuring that all outstanding communication has been resolved. Each processor runs the same program, but on different data, which means that BSP algorithms adhere to the Single Program Multiple Data (SPMD) paradigm.

The BSP cost of a BSP algorithm can predict the runtime of that algorithm when it is run on a BSP computer. This cost can be expressed completely in the parameters of a BSP computer. For each superstep, the cost depends on 1) $w_i^{(s)}$, the amount of work, measured in FLOPs, performed by processor s in the ith superstep, 2) $r_i^{(s)}$, the number of data words received, and 3) $t_i^{(s)}$, the number of data words transmitted (sent) by processor s in superstep i. The runtime of a parallel algorithm is dominated by the processor that takes the longest time, both for computation and communication. In the case of communication, we therefore require the concept of an h-relation, defined as the maximum number of words transmitted or received by any processor during the superstep, i.e., $h_i = \max_{0 \leq s < p} \max\{t_i^{(s)}, r_i^{(s)}\}$. This leads naturally to the following cost, the BSP cost, of a BSP algorithm consisting of k supersteps:

$$T = \sum_{i=0}^{k-1} \left(\max_{0 \leq s < p} w_i^{(s)} + g\, h_i + l \right).$$

The BSP model has inspired many parallel programming interfaces. BSPlib [Hil+98] describes a collection of a limited set of primitives which can be used for writing BSP programs in the C programming language. Libraries that implement the BSPlib standard include BSPonMPI [Sui] and MulticoreBSP for Java [YB12] and C [Yze+14]. Paderborn University BSP (PUB) [Bon+03] is an alternative BSP library that includes features not included in BSPlib such as subset synchronization and non-blocking collective operations. A functional BSP library is provided in BSML [LGB05] for the multi-paradigm programming language Objective CAML. Big data frameworks based on the BSP model include the popular MapReduce [DG04] and Pregel [Mal+10] frameworks introduced by Google. These frameworks have open-source implementations in respectively Apache Hadoop and Apache Giraph, the latter of which is used for large scale graph computing by, e.g., Facebook [Chi+15]. Apache Hama [Sid+16] is a recent BSPlib alternative for the Java programming language.

For the C++ programming language, high-level parallel programming libraries include HPX [Hel+17], whose current interface focuses on asynchronous and concurrent applications, UPC++ [Zhe+14], which provides a generic and object-oriented partitioned global address space (PGAS) interface, and BSP++ [HFE10] which targets hybrid SMP architectures and implements direct remote memory access but not bulk-synchronous message passing.

Modern hardware is increasingly hierarchical. In a typical HPC cluster there are many nodes, each consisting of (several) multi-core processors together with accelerators such as GPUs or many-core coprocessors. Furthermore, there are multiple layers of random-access memory and caches which all differ in, e.g., size, latency, and read and write speed. In 2011, Valiant introduced Multi-BSP [Val11], a hierarchical execution model based on BSP. The nested execution of BSP programs is available in, e.g., the PUB, MulticoreBSP, and NestStep [Keß00] libraries.

In this chapter we introduce Bulk, a library for the C++ programming language. The current version is based on the C++17 standard [ISO17]. By leveraging common idioms and features of modern C++ we increase memory safety and code reuse, and we are able to eliminate boilerplate code from BSP programs. Furthermore, the flexible backend architecture ensures that programs written on top of Bulk are able to simultaneously target systems with shared memory, distributed memory, or even hybrid systems. The remainder of this chapter is structured as follows. In Section 2.1 we introduce the Bulk library, and highlight the differences with previous BSP libraries. In Section 2.2 we discuss two applications, *regular sample sort* and the *fast Fourier transform (FFT)*. In Section 2.3, we provide experimental results for these applications. Finally, in Section 2.4, we present our conclusions and discuss possibilities for future work.

2.1 The Bulk library

The Bulk library is a modern BSPlib replacement which focuses on the memory safety, portability, code reuse, and ease of implementation of BSP

```
bulk::backend::environment env;
env.spawn(env.available_processors(), [](auto& world) {
    world.log("Hello world from %d / %d\n",
              world.rank(), world.active_processors());
});
```

Listing 2.1: The entry point for parallelism using Bulk. We create an environment, where `backend` is a placeholder for a concrete backend such as MPI or C++ threads. Next, we spawn an SPMD block using all the available processors.

algorithms. Additionally, Bulk provides the possibility to program hybrid systems and it has several new features compared to existing BSP libraries. First, we present all the concepts of the library that are necessary to implement classic BSP algorithms.

Bulk interface Here, we give an overview of the Bulk C++ interface[1]. We use a `monospace` font in the running text for C++ code and symbols. A BSP computer is captured in an `environment`. This can be an object encapsulating, e.g., an MPI cluster, a multi-core processor or a many-core coprocessor. Within this BSP computer, an SPMD block can be spawned. Collectively, the processors running this block form a *parallel world* that is captured in a `world` object. This object can be used to communicate, and for obtaining information about the local process, such as the processor identifier (PID, in Bulk denoted `rank`) and the number of active processors. In all the code examples, `s` refers to the local rank, and `t` to an arbitrary target rank.

A simple program written using Bulk first instantiates an environment object, which is then used to spawn an SPMD block (in the form of a C++ function) on each processor, to which the local world is passed. See Listing 2.1 for a code example, and Table 2.1 for a table with the relevant methods.

The spawned SPMD section, which is a function that takes the world as a parameter, consists of a number of supersteps. These supersteps are delimited with a call to `world::sync`. The basic mechanism for communication

[1]Although we try to be as complete as possible, we do not give a detailed and exhaustive list of all the methods and functions provided by the library. For such a list, together with all the function signatures and further examples we refer to the online documentation which can be found at `https://jwbuurlage.github.com/Bulk/`.

Table 2.1: Available methods for `environment` and `world` objects.

class	*method*	*description*
`environment`	`spawn`	starts an SPMD block
	`available_processors`	returns maximum p
`world`	`active_processors`	returns chosen p
	`rank`	returns local processor ID s
	`next_rank`	returns $s+1 \pmod{p}$
	`prev_rank`	returns $s-1 \pmod{p}$
	`sync`	ends the current superstep
	`log`	logs a string message

revolves around the concept of a distributed variable, which is captured in a `var` object. These variables should be constructed in the same superstep by each processor. Although each processor defines this distributed variable, its value is generally different on each processor. The value contained in the distributed variable on the local processor is called the *local value*, while the concrete values on remote processors are called the *remote values*.

A distributed variable is of little use if it does not provide a way to access remote values of the variable. Bulk provides encapsulated references to the local and remote values of a distributed variable. We call these references *image* objects. Images of remote values can be used for reading, e.g., `auto y = x(t).get()` to read from processor `t`, and for writing, e.g., `x(t) = value`, both with the usual bulk-synchronous semantics. See Listing 2.2 for a more elaborate example. Since the value of a remote image is not immediately available upon getting it, it is contained in a `future` object. In the next superstep, its value can be obtained using `future::value`, e.g., `y.value()`.

In this simple example, we already see some major benefits of Bulk over existing BSP libraries; 1) we avoid accessing and manipulating raw memory locations in user code, making the code more memory safe and 2) the resulting code is shorter, more readable and therefore less prone to errors. Note that these benefits do not come at a performance cost, since it can be seen as syntactic sugar that resolves to calls to internal functions that resemble common BSP primitives.

```
bulk::var<int> x(world);

auto t = world.next_rank();
x(t) = 2 * world.rank();
world.sync();
// x now contains two times the ID of the previous logical processor

auto b = x(t).get();
world.sync();
// b.value() now contains two times the local ID
```

Listing 2.2: The basic usage of a distributed variable. The variable is created on each processor running the SPMD block. Its images can then be written to by using the convenient syntax `x(processor) = value`. Remote values are obtained by using the syntax `x(processor).get()`.

When restricting ourselves to communication based on distributed variables, we lose the possibility of performing communication based on slices or arrays. Distributed variables whose images are arrays have a special status in Bulk, and are captured in `coarray` objects. The functionality of these objects is inspired by Coarray Fortran [NR98]. Coarrays provide a convenient way to share data across processors. Instead of manually sending and receiving individual data elements, coarrays model distributed data as a 2D array, where the first dimension is over the processors, and the second dimension is over local 1D array indices. The local elements of a coarray can be accessed as if the coarray were a regular 1D array. Images to the remote arrays belonging to a coarray `xs` are obtained in the same way as for variables, by using the syntax `xs(t)`. These images can be used to access the remote array. For example, `xs(t)[5] = 3` puts the value 3 in the array element at index 5 of the local array at processor `t`. Furthermore, convenient syntax makes it easy to work with slices of coarrays. A basic slice for the element interval `[start, end)`, i.e., including `start` but excluding `end`, is obtained using `xs(t)[{start, end}]`. See Listing 2.3 for examples of common coarray operations. We summarize the most important put and get operations for distributed variables and coarrays in Table 2.2.

```
auto xs = bulk::coarray<int>(world, 4);

auto t = world.next_rank();
xs[0] = 1;
xs(t)[1] = 2 + world.rank();
xs(t)[{2, 4}] = {123, 321};

world.sync();
// xs is now [1, 2 + world.prev_rank(), 123, 321]
```

Listing 2.3: The basic syntax for dealing with coarrays.

Instead of using distributed variables, it is also possible to perform one-sided *mailbox communication* using message passing, which in Bulk is carried out using a `queue`. The message passing syntax is greatly simplified compared to previous BSP interfaces, without losing power or flexibility. This is possible for two reasons. First, it is possible to construct several queues, removing a common use case for *tags* to distinguish different kinds of messages. Second, messages consisting of multiple components can be constructed on demand using a syntax based on variadic templates. This gives us the possibility of *optionally* attaching tags to messages in a queue, or even denoting the message structure in the construction of the queue itself. For example, `queue<int, float[]>` is a queue with messages that consist of a single integer, and zero or more real numbers. See Listing 2.4 for the basic usage of these queues.

In addition to distributed variables and queues, common communication patterns such as `gather_all`, `foldl`, and `broadcast` are also available. The Bulk library also has various utility features for, e.g., logging and benchmarking. We note furthermore that it is straightforward to implement generic skeletons on top of Bulk, since all distributed objects are implemented in a generic manner.

Backends and nested execution Bulk has a powerful backend mechanism. The initial release provides backends for *distributed memory* based on MPI [MPI94], *shared memory* based on the standard C++ threading library, and *data streaming* for the Epiphany many-core coprocessor [ONU14]. Note that for a shared-memory system, only standard C++ has to be used.

Table 2.2: An overview of the syntax for puts and gets in Bulk. Here, `x` and `xs` are a distributed variable and a coarray, respectively, e.g., `auto x = bulk::var<int>(world)`, `auto xs = bulk::coarray<int>(world, 10)`

object	*image*	*description*	*code*
`var`	local [(*)]	set	`x = 5`
		use	`auto y = x + 3`
	remote	put	`x(t) = 5`
		get	`auto y = x(t).get()`
`coarray`	local [(*)]	set	`xs[idx] = 5`
		use	`auto y = xs[idx] + 3`
	remote	put	`xs(t)[idx] = 5`
		get	`auto y = xs(t)[idx].get()`
		put slice[(**)]	`xs(t)[{start, end}] = {values...}`
		get slice[(**)]	`auto ys = xs(t)[{start, end}].get()`

[(*)]: a local image of a value of type `T` gets implicitly cast to a `T&` reference to the underlying value.
[(**)]: subarrays corresponding to slices are represented using `std::vector` containers.

This means that a parallel program written using Bulk can run on a variety of systems, simply by changing the environment that spawns the SPMD function. No other changes are required. In addition, libraries that build on top of Bulk can be written completely independently from the environment, and only have to manipulate the world object.

Different backends can be used together. For example, distinct compute nodes can communicate using MPI while locally performing shared-memory multi-threaded parallel computations, all using a single programming interface. Hybrid shared/distributed-memory programs can be written simply by nesting `environment` objects with different backends.

```
// queue containing simple data
auto numbers = bulk::queue<int>(world);
numbers(t).send(1);
numbers(t).send(2);
world.sync();
for (auto value : numbers)
    world.log("%d", value);

// queue containing multiple components
auto index_tuples = bulk::queue<int, int, float>(world);
index_tuples(t).send({1, 2, 3.0f});
index_tuples(t).send({3, 4, 5.0f});
world.sync();
for (auto [i, j, k] : index_tuples)
    world.log("(%d, %d, %f)", i, j, k);
```

Listing 2.4: The use of message passing queues. The local inbox acts as a regular container, so we can use a range-based for loop. The messages can be accessed in a concise way using structured bindings.

2.2 Applications

2.2.1 Parallel regular sample sort

Here, we present our BSP variant of the parallel regular sample sort proposed by Shi and Schaeffer in 1992 [SS92]. Hill, Donaldson, and Skillicorn [HDS97] presented a BSP version, and Gerbessiotis [Ger15] studied variants with regular oversampling. Our version reduces the required number of supersteps by performing a redundant mergesort of the samples on all processors.

Our BSP variant is summarized in Algorithm 1. Every processor first sorts its local block of size $b = n/p$ by a quicksort of the interval $[sb, (s+1)b-1]$, where s is the local processor identity. The processor then takes p regular samples at distance b/p and broadcasts these to all processors. We assume for simplicity that p divides b, and, for the purpose of explanation, that there are no duplicates (which can be achieved by using the original ordering as a secondary criterion). All processors then synchronize, which ends the first superstep. In the second superstep, the samples are concatenated and sorted. A mergesort is used, since the samples originating in the same processor were already sorted. Thus, p parts have to be merged. The start of part t is given by $\mathit{start}[t]$ and the end by $\mathit{start}[t+1]-1$. From these samples, p splitters are chosen at distance p, and they are used to split the local block into p parts. At the end of the second superstep, a local contribution X_{st} is sent to processor $P(t)$. In the third and final superstep, the received parts are concatenated and sorted, again using a mergesort because each received part has already been sorted. See Listing 2.5 for an illustration of Bulk implementations of the two communication phases of Algorithm 1.

Shi and Schaeffer have proven that the block size at the end of the algorithm is at most twice the block size at the start, thus bounding the size by $b_s \leq 2b$. A small optimization made possible by our redundant computation of the samples is that not all samples need to be sorted, but only the ones relevant for the local processor. The other samples merely need to be counted, separately for those larger and for those smaller than the values in the current block.

The total BSP cost of the algorithm, assuming p is a power of two, is

$$T_{\text{sort}} \leq \frac{n}{p}\log_2\frac{n}{p} + p^2\log_2 p + \frac{2n}{p}\cdot\log_2 p + \left(p(p-1) + 2\frac{n}{p}\right)g + 2l. \quad (2.1)$$

This is efficient in the range $p \leq n^{1/3}$, since the sorting of the array data then dominates the redundant computation and sorting of the samples.

2.2.2 Fast Fourier Transform

The discrete Fourier transform (DFT) of a complex vector $\mathbf{x}$ of length n is the complex vector $\mathbf{y}$ of length n defined by

$$y_k = \sum_{j=0}^{n-1} x_j \mathrm{e}^{-2\pi i jk/n} = \sum_{j=0}^{n-1} x_j {\omega_n}^{jk}, \quad \text{for } 0 \leq k < n, \quad (2.2)$$

where we use the notation $\omega_n = \mathrm{e}^{-2\pi i/n}$. The DFT can be computed in $5n\log_2 n$ floating-point operations by using a radix-2 Fast Fourier Transform (FFT) algorithm assuming that n is a power of two.

Algorithm 1 Regular sample sort for processor $P(s)$, with $0 \leq s < p$.

```
input:   $\mathbf{x}$: vector of length $n$, $n \bmod p^2 = 0$, block distributed with block
    size $b = n/p$.
output:  $\mathbf{x}$ sorted in increasing order, block distributed with variable block
    size $b_s \leq 2b$.

    Quicksort($\mathbf{x}, sb, (s+1)b-1$);
    for $i := 0$ to $p-1$ do
        $sample_s[i] := x[sb + i \cdot \frac{b}{p}]$;

    for $t := 0$ to $p-1$ do
        put $sample_s$ in $P(t)$;
    Sync;

    for $t := 0$ to $p-1$ do
        $start[t] := tp$;
        for $i := 0$ to $p-1$ do
            $sample[tp+i] := sample_t[i]$;
    $start[p] := p^2$;
    Mergesort($sample, start, p$);

    for $t := 0$ to $p-1$ do
        $splitter[t] := sample[tp]$;
    $splitter[p] := \infty$;

    for $t := 0$ to $p-1$ do
        $X_{st} := \{x_i \ : \ sb \leq i < (s+1)b \wedge splitter[t] \leq x_i < splitter[t+1]\}$;
        put $X_{st}$ in $P(t)$;
    Sync;

    $\mathbf{x}_s := \cup_{t=0}^{p-1} X_{ts}$;

    $start_s[0] := 0$;
    for $t := 1$ to $p$ do
        $start_s[t] := start_s[t-1] + |X_{t-1,s}|$;
    $b_s := start_s[p]$;
    Mergesort($\mathbf{x}_s, start_s, p$);
```

Listing 2.5: Two communication phases in the regular sample sort algorithm.

```
auto samples = bulk::coarray<T>(world, p * p); // Broadcast samples
for (int t = 0; t < p; ++t)
    samples(t)[{s * p, (s + 1) * p}] = local_samples;
world.sync();

auto q = bulk::queue<int, T[]>(world); // Contribution from P(s) to P(t)
for (int t = 0; t < p; ++t)
    q(t).send(block_sizes[t], blocks[t]);
world.sync();
```

Our parallel algorithm for computing the DFT uses the *group-cyclic distribution* with cycle $c \leq p$, and is based on the algorithm presented in [IB01] and explained in detail in [Bis04]. The group-cyclic distribution first assigns a block of the vector $\mathbf{x}$ to a group of c processors and then assigns the vector components within that block cyclically. The number of processor groups (and blocks) is p/c. The block size of a group is nc/p. Here, we assume that n, p, c are powers of two. For $c = 1$, we retrieve the regular block distribution, and for $c = p$ the cyclic distribution.

The parallel FFT algorithm starts and ends in a cyclic distribution. First, the algorithm permutes the local vector with components

$$x_s, x_{s+p}, x_{s+2p}, \ldots, x_{s+n-p},$$

by swapping pairs of components with bit-reversed local indices. The resulting storage format of the data can be viewed as a block distribution, but with the processor identities bit-reversed. The processor numbering is reversed later, during the first data redistribution. After the local bit reversal, a sequence of butterfly operations is performed, just as in the sequential FFT, but with every processor performing the pairwise operations on its local vector components. In the common case $p \leq \sqrt{n}$, the BSP cost of this algorithm is given by

$$T_{\text{FFT},\ p \leq \sqrt{n}} = \frac{5n \log_2 n}{p} + 2\frac{n}{p} g + l. \tag{2.3}$$

Table 2.3: Speedups of parallel sort (top) and parallel FFT compared to `std::sort` from libstdc++, and the sequential algorithm from FFTW 3.3.7, respectively. Also given is the sequential time t_{seq}.

		$p=1$	$p=2$	$p=4$	$p=8$	$p=16$	$p=32$	t_{seq}(s)
Sort	$n=2^{20}$	0.93	1.95	3.83	6.13	8.10	12.00	0.08
	$n=2^{21}$	1.01	2.08	4.11	7.28	10.15	15.31	0.19
	$n=2^{22}$	0.88	1.82	3.58	5.99	10.27	13.92	0.33
	$n=2^{23}$	0.97	1.90	3.63	6.19	11.99	16.22	0.72
	$n=2^{24}$	0.93	1.79	3.21	6.33	8.47	14.76	1.39
FFT	$n=2^{23}$	0.99	1.07	2.08	2.77	5.60	5.51	0.20
	$n=2^{24}$	1.00	1.26	2.14	3.07	5.68	6.08	0.45
	$n=2^{25}$	1.00	1.23	2.22	3.09	5.80	6.05	0.96
	$n=2^{26}$	0.99	1.24	2.01	3.28	5.48	5.97	1.93

2.3 Results

We evaluate the performance of Bulk implementations of the BSP algorithms sample sort and FFT outlined in the previous section. The numbers presented are obtained on a single computer with two Intel Xeon Silver 4110 CPUs, each with 8 cores and 16 hardware threads for a total of 32 hardware threads, using the C++ threads backend. The benchmark programs are compiled with GCC 7.2.1. The results are shown in Table 2.3. The parallel sort implementation is a direct translation of Algorithm 1, except that we opt for a three-phase communication protocol instead of relying on bulk-synchronous message passing to avoid potentially superfluous buffer allocations. The parallel FFT implementation is as described in Section 2.2.2, where we use FFTW [FJ98] as a sequential kernel[2]. The input arrays for both algorithms have size n, and they are run on p processors.

For the parallel sorting algorithm, the array contains uniformly distributed random integers between 0 and 2×10^5. We observe that good speedups are obtained compared to the sequential implementation. The maximum speedup seen is about 16× with $p = 32$ and $n = 2^{23}$.

For the FFT results, the vector has size n. We observe good scalability up to $p = 16$, where we seem to hit a limit presumably because of the shared floating-point unit (FPU) between two logical threads on the same physical core, and possibly also due to the memory requirements in the

[2]We use plans with the so-called planning-rigor flag `FFTW_MEASURE`.

redistribution phase.

Various other algorithms and applications have been implemented on top of Bulk. The current library release includes a number of examples, such as simple implementations for the *inner product*, or the *word count* problem. Future releases of the library are planned to have additional components. One such component is support for arbitrary data distributions, which is already available as an experimental feature. Furthermore, an open-source application in computed tomography, Tomos, has been developed on top of Bulk, illustrating that the library can be used for the implementation of more complicated software.

2.3.1 Bulk vs. BSPlib

We believe the main goal of Bulk, which is to improve memory safety, portability, code reuse, and ease of implementation compared to BSPlib, has been largely achieved. In Listing 2.6, we show a Bulk and a BSPlib implementation of a common operation. The Bulk implementation avoids the use of raw pointers, uses generic objects, requires significantly fewer lines of code, and is more readable.

We compare the performance of Bulk to a state-of-the-art BSPlib implementation, MulticoreBSP for C (MCBSP) [YR14], version 2.0.3 released in May 2018. We use the implementations of BSPedupack [Bis04], version 2.0.0-beta, as the basis of our BSPlib programs.

Table 2.4 shows the performance of Bulk compared to BSPlib. For sorting, the Bulk implementation is significantly faster, presumably because the internal sorting algorithm used is different. The Bulk implementation uses the sorting algorithm from the C++ standard library, whereas the BSPlib implementation uses the quicksort from the C standard library. The BSPedupack FFT implementation has been modified to use FFTW for the sequential kernel. For the FFT, MCBSP outperforms Bulk slightly on larger problem sizes.

In Table 2.5, the BSP parameters are measured for Bulk and MCBSP. The computation rate r is measured by applying a simple arithmetic transformation involving two multiplications, one addition and one subtraction, to an array of 2^{23} double-precision floating-point numbers. The latency l is measured by averaging over 100 bulk synchronizations without communication. The communication-to-computation ratio g is measured by

```
// BSPlib
int* xs = malloc(10 * sizeof(int));
bsp_push_reg(xs, 10 * sizeof(int));
bsp_sync();

int ys[3] = {2, 3, 4};
bsp_put((s + 1) % p, ys, xs, 2, 3 * sizeof(int));
bsp_sync();
...
bsp_pop_reg(xs);
free(xs);

// Bulk
auto xs = bulk::coarray<int>(world, 10);
xs(world.next_rank())[{2, 5}] = {2, 3, 4};
world.sync();
```

Listing 2.6: A comparison between Bulk and BSPlib for putting a subarray.

Table 2.4: Comparing implementations of BSPedupack running on top of MCBSP, to our implementations on top of Bulk.

Sort			FFT		
size	t_{MCBSP} (s)	t_{Bulk} (s)	size	t_{MCBSP} (s)	t_{Bulk} (s)
$n = 2^{20}$	24.49	13.80	$n = 2^{22}$	0.153	0.144
$n = 2^{21}$	53.00	28.76	$n = 2^{23}$	0.305	0.320
$n = 2^{22}$	113.6	62.42	$n = 2^{24}$	0.629	0.694
$n = 2^{23}$	237.2	142.8			

communicating subarrays of various sizes, consisting of up to 10^7 double-precision floating-point numbers, between various processor pairs.

The MCBSP library uses a barrier based on a spinlock mechanism by default. This barrier gives better performance, leading to a low value for l. Alternatively, a more energy-efficient barrier based on a mutex can be used, which is similar to the barrier that is implemented in the C++ backend for Bulk. With this choice, the latency of MCBSP and Bulk are comparable. MCBSP is able to obtain a better value for g. We plan to include a spin-

Table 2.5: The BSP parameters for MCBSP and the C++ thread backend for Bulk.

Method	r (GFLOP/s)	g (FLOPs/word)	l (FLOPs)
MCBSP (spinlock)	0.44	2.93	326
MCBSP (mutex)	0.44	2.86	10484
Bulk	0.44	5.65	11702

lock barrier in a future release of Bulk, and to improve the communication performance further[3].

2.4 Conclusion

We present Bulk, a modern BSP interface and library implementation with many desirable features such as memory safety, support for generic implementations of algorithms, portability, and encapsulated state, and show that it allows for clear and concise implementations of BSP algorithms. Furthermore, we show the scalability of two important applications implemented in Bulk by providing experimental results. Even though both algorithms have $\mathcal{O}(n \log n)$ complexity, and nearly all input data have to be communicated during the algorithm, we still are able to obtain good speedups with our straightforward implementations. The performance of Bulk is close to that of a state-of-the-art BSPlib implementation, except for the mutex-based barrier.

[3]Spinlock barriers were introduced in Bulk version 1.1.0 which was released after the publication on which this chapter is based. With this implementation the measured latency l for Bulk is reduced to 467 FLOPs.