Int J Parallel Prog (2014) 42:619–642
DOI 10.1007/s10766-013-0262-9

MulticoreBSP for C: A High-Performance Library for Shared-Memory Parallel Programming

A. N. Yzelman · R. H. Bisseling · D. Roose · K. Meerbergen

Received: 1 March 2013 / Accepted: 29 July 2013 / Published online: 25 August 2013

Abstract The bulk synchronous parallel (BSP) model, as well as parallel programming interfaces based on BSP, classically target distributed-memory parallel architectures. In earlier work, Yzelman and Bisseling designed a MulticoreBSP for Java library specifically for shared-memory architectures. In the present article, we further investigate this concept and introduce the new high-performance MulticoreBSP for C library. Among other features, this library supports nested BSP runs. We show that existing BSP software performs well regardless whether it runs on distributed-memory or shared-memory architectures, and show that applications in MulticoreBSP can attain high-performance results. The paper details implementing the Fast Fourier Transform and the sparse matrix–vector multiplication in BSP, both of which outperform state-of-the-art implementations written in other shared-memory parallel programming interfaces. We furthermore study the applicability of BSP when working on highly non-uniform memory access architectures.

Keywords High-performance computing · Bulk synchronous parallel · Shared-memory parallel programming · Software library · Fast Fourier transform · Sparse matrix–vector multiplication

A. N. Yzelman
Flanders ExaScience Lab (Intel Labs Europe), Heverlee, Belgium

A. N. Yzelman (✉) · D. Roose · K. Meerbergen
Department of Computer Science, KU Leuven,
Celestijnenlaan 200A - bus 2402, 3001 Heverlee, Belgium
e-mail: albert-jan.yzelman@cs.kuleuven.be

R. H. Bisseling
Department of Mathematics, Utrecht University, Utrecht, The Netherlands

1 Introduction

The bulk synchronous parallel (BSP) model [19], introduced by Valiant, describes a powerful abstraction of parallel computers. It enables the design of theoretically optimal parallel algorithms, and inspired many interfaces for parallel programming. The BSP model consists of three parts: (1) an abstraction of a parallel computer, (2) an abstraction of a parallel algorithm, and (3) a cost model. A BSP computer has p homogeneous processors, each one with access to local memory. They cannot access remote memory, but may communicate through a black-box network interconnect. Preparing the network for all-to-all communication while synchronising the p processors at the start and end of communication costs l units; sending a data word during the all-to-all communication costs g units. Measuring l and g in seconds does not directly relate to any work done; instead, if the speed r of each processor is measured in floating-point operations per second (flop/s), we express l and g in flops as well. The four parameters (p, r, l, g) completely define a BSP computer.

A BSP algorithm runs on a BSP computer and adheres to the Single Program, Multiple Data (SPMD) paradigm. Each BSP process consists of alternating computation and communication phases. During computation, each process executes sequential code and cannot communicate with other BSP processes; during communication all processes are involved in an all-to-all data interchange and cannot perform any computations. BSP synchronises all processors in-between phases. We define one superstep as one computation phase combined with the communication phase that directly follows it.

This definition of a BSP computer and a BSP algorithm immediately leads to the BSP cost model. If the algorithm consists of T supersteps, and if process s has $w_i^{(s)}$ work to perform in superstep i, then the total computation cost is $\sum_{i=0}^{T-1} \max_s w_i^{(s)}$. We assume that the network allows the simultaneous sending and receiving of messages, and that the sending and the receiving of messages form the bottleneck of the communication. Writing $r_i^{(s)}$ for the number of words received by process s during superstep i, $t_i^{(s)}$ for the number of words transmitted (sent), and $h_i = \max\{\max_s r_i^{(s)}, \max_s t_i^{(s)}\}$ for the h-relation of the ith superstep, the communication cost is $g \cdot \sum_{i=0}^{T-1} h_i$. Accounting for the latency costs of synchronisation and network initialisation incurred at each superstep yields the full BSP cost:

$$C = \sum_{i=0}^{T-1} \left(\max_s w_i^{(s)} + g \cdot h_i + l \right). \tag{1}$$

Here, we express C in flops; C/r yields the time taken in seconds.

The h-relation is a central concept of BSP. Just as it is natural to induce load balance (minimise $\max_s w_i^{(s)}$), minimising the h-relation induces low communication requirements while also providing an incentive to balance the communication among the processes; i.e., all processes should send and receive roughly the same amount of data.

To illustrate, consider a one-to-all broadcast involving all p processes. One process transmits $p - 1$ words and receives none, while all other processes transmit none and

receive one word. This is an unbalanced $(p-1)$-relation in the BSP model; the source of the broadcast is the bottleneck. An all-to-all broadcast where each process sends and receives $(p-1)$ words is a balanced $(p-1)$-relation. A situation where each process sends and receives exactly 1 word (e.g., a pairwise swap or a one-directional ring exchange), corresponds to a perfectly balanced 1-relation independent of p.

We present a new shared-memory interface for BSP programming in C, called MulticoreBSP for C. Previous work on BSP programming interfaces include BSP++ [8], a hybrid library targeting systems with both shared and distributed memory such as clusters of symmetric multiprocessors. It is object-oriented and its performance on inner product, FFT, and LU decomposition benchmarks is demonstrated to be similar to that of BSPedupack [1], with a performance gain of 7 % on the LU decomposition of a matrix of size 4,096. BSML [16] is a library that combines the BSP model with the functional programming language Objective Caml. A combination of the high-level Python language and BSP is provided by the Python BSP package [10], part of ScientificPython, which can run on top of BSPlib [9] (an MPI version is also available); thus the Python BSP package can also be run on top of our MulticoreBSP for C library. The Orléans Skeleton Library [13] provides BSP algorithmic skeletons in C++, built on top of MPI. Implementations of the 1D heat equation and the FFT test favourably compared to other skeleton-based implementations.

The remainder of the paper will first briefly introduce all MulticoreBSP updates to the standard BSPlib programming interface in Sect. 2 (the Appendix includes a more detailed discussion of all available primitives). Section 3 describes the two applications we use to attain high performance and demonstrate backward compatibility, as shown by the experiments reported in Sect. 4. Conclusions and suggestions for future work appear in Sect. 5.

2 The MulticoreBSP Programming Interface

The MulticoreBSP for C programming interface adheres to the ANSI C99 standard, directly derives from the BSPlib [9] interface by Hill et al., and is inspired by the Java MulticoreBSP library by Yzelman and Bisseling [24]. Compared to BSPlib, MulticoreBSP for C adds two new high-performance primitives and updates the interface of existing primitives. A compatibility mode ensures full support for existing BSPlib programs. For maximum portability, the library depends on only two established standards: POSIX threads (PThreads) and the POSIX realtime extension [11]. The new library is freely available via http://www.multicorebsp.com.

While MulticoreBSP targets shared-memory computing specifically and thus employs thread-based parallelisation, BSPlib and its implementations aim at distributed-memory supercomputing and thus explicitly start separate processes on supercomputer nodes. We keep the process terminology for the remainder of the paper, but the use of threading does have implications on the user level; Sect. 2.3 discusses these.

The updated interface consists of 22 primitive function calls. For each primitive, we now state the interface declarations.[1] To help users better understand what to

[1] For brevity, we omit the `const` and `restrict` keywords.

expect from calling a BSP primitive, the declarations include the asymptotic run-time complexity of each primitive. The MulticoreBSP for C library indeed attains these asymptotic time complexities. Those in big-Theta notation indicate that a lower bound is not possible, while those in big-Oh notation leave room for improvement. Readers unfamiliar with the original BSPlib interface may find brief descriptions of each primitive in the Appendix. Section 2.3 highlights the differences with respect to the original BSPlib interface. For brevity, we still refer to the updated interface as the BSPlib interface.

First, we list primitives that control the flow of the SPMD sections of BSP programs. MulticoreBSP does not add new primitives in this category.

- `void bsp_init( void (*spmd)(void), int argc, char **argv );` $\Theta(1)$.
- `void bsp_begin( unsigned int P );` $\mathcal{O}(\mathtt{P})$.
- `void bsp_end();` $\mathcal{O}(l)$.
- `unsigned int bsp_nprocs();` $\Theta(1)$.
- `unsigned int bsp_pid();` $\Theta(1)$.
- `void bsp_sync();` $\Theta(l + g \cdot h_i)$, see Eq. 1.
- `void bsp_abort( char *error, ... );` $\Theta(1)$.
- `double bsp_time();` $\Theta(1)$.

The MulticoreBSP 'hello-world' program in Algorithm 1 illustrates the use of the first five primitives. Note that it consists of a single computation phase; in BSPlib all SPMD code is part of a computation phase. Communications, either in the form of direct remote memory access (DRMA) or bulk synchronous message passing (BSMP), are initiated through calls during a computation phase. Thus BSPlib communication primitives do not immediately execute communication, but instead queue the communication requests. These are processed at the end of the current superstep, as indicated by `bsp_sync`.

Algorithm 1 A 'hello world' example program in MulticoreBSP

```
#include <mcbsp.h> //the MulticoreBSP for C header file
#include <stdio.h>

void spmd() {
        bsp_begin( bsp_nprocs() );
        printf( "Hello world from process %d!\n", bsp_pid() );
        bsp_end();
}

int main( int argc, char **argv ) {
        bsp_init( &spmd, argc, argv );
        spmd();
        return 0;
}
```

BSPlib provides DRMA communication via 'put' and 'get' primitives. When a process issues a put, it copies data from a local memory area into remote memory. The

'get' does the reverse; it retrieves data from a remote memory area and copies it to local memory. If an SPMD program defines a local variable, each of the p processes has its own memory area associated with that variable. To make DRMA work, processes must become aware of which memory address relates to which variable. BSPlib defines two primitives to facilitate this registration process:

- `void bsp_push_reg( void *address, size_t size );` $\Theta(1)$.
- `void bsp_pop_reg( void *address );` $\Theta(1)$.

Registration and deregistration necessitate one all-to-all broadcast during synchronisation per call, in the worst case. These should be taken into account when calculating the BSP h-relations for estimating the execution time of the `bsp_sync`. Registration enables using the following communication primitives:

- `void bsp_put( unsigned int pid, void *source, void *destination, size_t offset, size_t size );` $\Theta(\texttt{size})$.
- `void bsp_get( unsigned int pid, void *source, size_t offset, void *destination, size_t size );` $\Theta(1)$.

Algorithm 2 illustrates the use of DRMA primitives in the computation of an inner product of two vectors. Each process has local vectors x, y of size `np` (which equals the global problem size n divided by p). It calculates the local contribution $\alpha = \langle x, y \rangle$, and then uses `bsp_put` to broadcast α to all other processes. Note the use of offsets to write to unique positions in the p local `ip_buffer` arrays (of length p each). The second computation phase redundantly computes the global inner product and returns the final result. To initialise and register the buffer used in broadcasting α, one call to `ip_init` must precede one or more calls to `ip`; a single initialisation superstep of a BSPlib program typically contains several such initialisation calls.

BSMP enables the sending of messages to remote processes. Messages have two parts: (1) a fixed-size tag that may describe the purpose of the message, and (2) a payload of arbitrary size. Calling the BSP 'send' primitive constructs and queues a BSMP message, which is sent during the next `bsp_sync`. Received messages end up in a local BSMP queue. BSPlib allows querying the number of in-queue messages, allows reading the tag of the first in-queue message, and allows moving the payload of that message into user-managed memory. Moving a message removes it from the queue. No registration process is required for BSMP communication, and the full interface is as follows:

- `void bsp_set_tagsize( size_t *size );` $\Theta(1)$.
- `void bsp_send( unsigned int pid, void *tag, void *payload, size_t size );` $\Theta(\texttt{size})$.
- `void bsp_qsize( unsigned int *packets, size_t *accumulated_size );` $\mathcal{O}(\texttt{packets})$.
- `void bsp_get_tag( size_t *status, void *tag );` $\Theta(1)$.
- `void bsp_move( void *payload, size_t max_copy_size );` $\Theta(\texttt{size})$.

Algorithm 2 A BSP inner-product algorithm using DRMA

```
#include <mcbsp.h>
#include <iostream>

//initialisation function for ip
void ip_init( double **ip_buffer ) {
        const size_t size = bsp_nprocs() * sizeof(double);
        *ip_buffer = malloc( size );
        bsp_push_reg( *ip_buffer, size );
}

//calculates the inner-product from the local vectors
double ip( double *x, double *y, double *ip_buffer, size_t np ) {
        double alpha = 0.0;
        for( size_t i = 0; i < np; ++i )
                alpha += x[ i ] * y[ i ];
        for( unsigned int k = 0; k < bsp_nprocs(); ++k ) {
                bsp_put( k, &alpha, ip_buffer,
                        bsp_pid() * sizeof(double), sizeof(double) );
        }
        bsp_sync();
        for( unsigned int k = 1; k < bsp_nprocs(); ++k )
                ip_buffer[ 0 ] += ip_buffer[ k ];
        return ip_buffer[ 0 ];
}

//example usage
void spmd() {
        bsp_begin( bsp_nprocs() );
        double *ip_buffer, *x, *y;
        size_t np;
        ip_init( &ip_buffer );
        ... //more initialisation calls to set x, y, np, and others
        bsp_sync();
        ... //calculations, until we need alpha=<x,y>:
        double alpha = ip( x, y, ip_buffer, np );
        ... //calculations using alpha
        bsp_end();
}
```

2.1 High-Performance Variants

BSPlib defines high-performance (`hp`) variants of DRMA and BSMP primitives. These are `bsp_hpput`, `bsp_hpget`, and the new `bsp_hpsend`. They allow communication to occur immediately after calling the `hp`-primitive, but still ensure communication to have occurred after the next `bsp_sync`. The gains over non-`hp` primitives are two-fold: `hp`-primitives allow for overlap of computation and communication whenever possible, and they avoid the inefficiency in time and memory of buffering communication. Users must guarantee that the source and destination memory areas remain unchanged until the end of the current computation phase. Errors in the use of `hp`-variants may cause non-deterministic behaviour that cannot be caught by the BSP run-time system; users should consider the added costs of ensuring correctness of their applications when considering `hp`-primitives.

Although the bsp_hpsend avoids buffering-on-send, messages still enter a buffer upon receipt: the BSMP queue. Instead of copying messages from this buffer, the following hp-primitive avoids copying by directly returning pointers to the tag and payload in the receive buffers:

– size_t bsp_hpmove(void **p_tag, void **p_payload); $\Theta(1)$.

The primitive also returns the payload size in bytes, or the largest possible value (SIZE_MAX) if the queue is empty.[2] This is the only hp-primitive for which the interface differs from its non-hp version.

To exploit the shared-memory architecture and avoid synchronisations where possible, Yzelman and Bisseling introduced the bsp_direct_get primitive [24, Section 2]. Its semantics is exactly that of the bsp_hpget, but the primitive immediately starts communication and waits for completion thereof. Like with the bsp_hpget, the user must ensure that the remote data remains unchanged during the current computation phase. Using the direct-get allows this superstep to be merged with the next one if no other communication primitives were called, thus saving a BSP synchronisation. The bsp_direct_get is the only hp-primitive that runs in Θ(size) time instead of $\Theta(1)$ time.

2.2 Hierarchical Execution

MulticoreBSP for C supports hierarchical execution of BSP programs. This means that BSP processes may call bsp_init and bsp_begin within SPMD sections. A BSP process doing this is considered the initialising process for the upcoming nested BSP run, and must adhere to the same rules as a regular initial process that starts a BSP run. After a bsp_begin, the initialising process will spawn the processes required for the nested BSP run, and will itself continue as (nested) process 0. Nested processes have no knowledge of the BSP processes that spawned them; previous variable registrations are no longer valid, and all BSP primitives only relate to the sibling processes corresponding to the nested BSP run.

It is thus possible to create c groups of p/c BSP processes by first starting a BSP run with c processes, after which each process starts its own BSP run using p/c processes. An example use-case is avoiding global synchronisations: a bsp_sync in the nested run will involve only p/c processes, while one on the top level involves only c processes; nowhere will BSP synchronise over all p processes. When process 0 in a nested run exits by a bsp_end, its ID will reset to its original ID and the parent SPMD program continues as normal. All BSP primitives called now again correspond to the original run over c processes. All previous data is retained, and earlier variable registrations are again valid. This concept is not new for BSP: Valiant allows for nested BSP computers with Multi-BSP [20], as earlier also investigated by de la Torre and Kruskal [5]. The PUB library [2] and NestStep [14] implement similar functionality.

[2] Note that the original BSPlib used an int as return type and returned -1 if the queue was empty.

2.3 Changes with Respect to BSPlib

BSP primitives that existed in the original BSPlib take the same arguments, but their data types have been updated. Values that reflect byte-sizes now have type `size_t`, while values that reflect process IDs and that count incoming BSMP messages now are of type `unsigned int`. These choices can be adapted at compile-time, and compiling in compatibility mode resets all types to those defined in the original BSPlib.

One of the new primitives that MulticoreBSP for C defines, the `bsp_hpsend`, is expected to be of use in a distributed-memory setting as well. The `bsp_direct_get` specifically targets shared-memory architectures, however. Both additions are in the spirit of the `hp`-variants already available in BSPlib; they allow overlap of communication with computation to gain in practical performance. The BSP cost model then remains an upper bound on performance of the algorithm. Just as with the BSPlib `bsp_hpput` and `bsp_hpget`, both new primitives should be used with care.

MulticoreBSP for C employs POSIX threads within its run-time system. The threading model implies that all globally declared variables are visible from all threads; all BSP processes thus share global variables in MulticoreBSP for C. Variables used locally by functions in an SPMD area thus must be declared within functions in the SPMD area. Programs usually already follow this principle: e.g., all applications in BSPedupack [1] run without modification under MulticoreBSP for C (with compatibility mode enabled).

The C language was chosen for this high-performance implementation of MulticoreBSP as it enables BSP programming in both C and C++. We do provide a C++-specific header that includes all BSP primitives described above, and additionally defines a `BSP_program` class that wraps the C interface. The full object-oriented approach of the Java MulticoreBSP library [24] has not been ported. Communicating arbitrary C++ objects often requires explicit marshalling, thus incurring a performance penalty.

A class of type `BSP_program` is an SPMD program, and each class instance corresponds to a single BSP process. It defines three functions:

```
– virtual void BSP_program::spmd()
– virtual BSP_program *BSP_program::newInstance()
– void BSP_program::begin( unsigned int P = bsp_nprocs() )
```

The latter function starts a BSP run corresponding to its class, and replaces calls to `bsp_init`, `bsp_begin`, and `bsp_end`. It spawns $P-1$ sibling processes and creates a new class instance for each process. The first function is the entry-point of the SPMD section, while the second enables MulticoreBSP to create a new class instance. Both functions are purely virtual and must be implemented by the user. Algorithm 3 contains a brief example that is functionally equivalent to the C code in Algorithm 1. Using this wrapper has the advantage that all class-local variables remain local to the BSP processes.[3]

[3] Global variables defined outside of classes remain visible by all BSP processes, however.

Algorithm 3 A BSP 'hello world' in the MulticoreBSP C++ wrapper

```
#include "mcbsp.hpp" //The C++ wrapper for MulticoreBSP
#include <iostream>

class Hello_World: public mcbsp::BSP_program {

      //The SPMD section each BSP process executes from its own class instance
      virtual void spmd() {
            std::cout << "Hello world from process " << bsp_pid() << "!\n";
      }

      //Used by MulticoreBSP to spawn P-1 other Hello_World instances
      virtual BSP_program *newInstance() { return new Hello_World(); }

      Hello_World() {} //A simple constructor
};

int main() {
      Hello_World p; //Construct the Hello_World instance for BSP process 0
      p.begin();     //Spawn bsp_nprocs()-1 sibling processes, execute spmd()
      return 0;      //All sibling processes have exited; process 0 terminates
}
```

3 Two Applications Implemented in BSP

To demonstrate that MulticoreBSP for C performs well on existing BSP software written according to the BSPlib standard, we consider the BSP Fast Fourier Transform (FFT) program described by Bisseling [1, Chapter 3]. To attain performance comparable to that of state-of-the-art parallel implementations, we modify the algorithm to use optimised sequential FFT kernels. The resulting BSP FFT is run on a modern distributed-memory cluster using BSPonMPI [18], and on a shared-memory machine using MulticoreBSP for C.

To show that the library enables writing high-performance parallel codes, we create two BSP versions of the 2D sparse matrix–vector (SpMV) multiplication described in Yzelman and Roose [22]. Their performance is compared against the best-performing state-of-the-art methods considered in the same paper. The following two sections briefly discuss the implementation of both BSP algorithms.

3.1 Fast Fourier Transformation

Given a complex vector x of length $n = 2^m$ (for integer m), the matrix–vector formulation of the discrete Fourier transform reads as $F_n x$. A radix-2 decimation-in-time FFT splits this computation in two:

$$F_n = B_n(I_2 \otimes F_{n/2})S_n, \tag{2}$$

with B_n the butterfly matrix

$$B_n = \begin{pmatrix} I_{n/2} & \Omega_{n/2} \\ I_{n/2} & -\Omega_{n/2} \end{pmatrix},$$

I_n the $n \times n$ identity matrix, and '$\otimes$' the Kronecker matrix product commonly used for expressing FFT computations. The even-odd sorting matrix S_n, when applied to a vector x, permutes all the even-indexed elements of the vector to the top and all odd-indexed elements to the bottom. The diagonal weights-matrix $\Omega_{n/2}$ contains $n/2$ Fourier weights $\{e^{-2\pi\imath k/n}\}$, $0 \leq k < n/2$. The FFT exploits the symmetry in the Fourier weights by recognising $e^{-2\pi\imath(n/2+k)/n} = -e^{-2\pi\imath k/n}$, thus saving half of the multiplications needed for the computation: the first $n/2$ weights need only be multiplied by a constant -1 (a sign change). Further exploitation of symmetry yields higher-radix formulations of the FFT, but the effectiveness deteriorates exponentially fast while the involved constants become increasingly costly to use.

Repeated application of the decomposition in Eq. 2 leads to the full matrix–vector FFT formulation:

$$F_n = \prod_{i=0}^{m-1} (I_{2^i} \otimes B_{n/2^i}) \prod_{i=1}^{m} (I_{n/2^i} \otimes S_{2^i}) = U_n R_n,$$

with $U_n = \prod_{i=0}^{m-1} (I_{2^i} \otimes B_{n/2^i})$ the unordered FFT (UFFT) of size n and R_n the bit-reversal permutation. The order of the product notation $\prod$ is left-to-right for increasing index i.

Applying $R_{n/p}$ to the local elements of a cyclically distributed input vector of size n (over $p = 2^q$ processes, q integer), results in a globally bit-reversed vector that is block-distributed over the same p processes, but with the process IDs themselves in bit-reverse. If $p \leq \sqrt{n}$, we have that $q \leq m - q$, and a block distribution for x suffices to calculate

$$H_n x = \prod_{i=m-q}^{m-1} (I_{2^i} \otimes B_{n/2^i}) x$$

without communicating with other processes. For $i < q$, the butterfly matrix $B_{n/2^i}$ requires multiplication of Fourier weights with input vector elements that are not all process-local in the block distribution. If the vector were distributed cyclically, however, computing

$$G_n x = \prod_{i=0}^{m-q-1} (I_{2^i} \otimes B_{n/2^i}) x$$

would become an entirely process-local operation.

A scheme where the input vector x initially is distributed cyclically, then locally bit-reversed, then multiplied with H_n, then re-distributed again to a cyclic distribution (while repairing the bit-reversed process IDs), and finally multiplied with the remainder computation G_n, is the base idea of this BSP FFT algorithm. Note that the vector distribution is cyclic at input and cyclic at output.

The matrix H_n corresponds to n/p UFFTs of length p; these are all processor-local UFFTs if x is block distributed. The global operation $G_n x$ with x distributed cyclically,

translates to p process-local computations $G_n^{(s)} x^{(s)}$, $0 \leq s < p$. The $(n/p) \times (n/p)$ matrix $G_n^{(s)}$ has its elements $g_{jk}^{(s)}$ equal to $g_{s+jp,s+kp}$ from G_n, and can be succinctly written as

$$G_n^{(s)} = \prod_{i=0}^{m-q-1} I_{2^i} \otimes \begin{pmatrix} I_{2^{m-i-q-1}} & \Omega^{s/p}_{2^{m-i-q-1}} \\ I_{2^{m-i-q-1}} & -\Omega^{s/p}_{2^{m-i-q-1}} \end{pmatrix}.$$

The matrix $\Omega^{\alpha}_{r/2}$ is a diagonal matrix of size $r/2$ with entries $\{e^{-2\pi\imath(k+\alpha)/r}\}$, $0 \leq k < r/2$. In our case, $\alpha = s/p$ and $r = 2^{m-i-q} = n/(2^i p)$, so that $(k+\alpha)/r = (s+kp)2^i/n$. The $\Omega^{\alpha}_{r/2}$ matrix corresponds to the regular matrix-formulation of an α-shifted generalised FFT (GFFT) of length r, defined as

$$y_k = \sum_{j=0}^{r-1} x_j e^{-2\pi\imath j(k+\alpha)/r}.$$

This may be implemented as a regular FFT with modified Fourier weights. Alternatively, a regular FFT preceded by an extra 'twiddling' step, i.e., $y = \mathrm{FFT}(T_n^{\alpha} x)$, can emulate a GFFT:

$$y_k = \sum_{j=0}^{r-1} \left(x_j e^{-2\pi\imath\alpha j/r} \right) e^{-2\pi\imath jk/r}, \tag{3}$$

with T_r^{α} a diagonal matrix with entries $\{e^{-2\pi\imath\alpha j/r}\}$, $0 \leq j < r$. Hence $G_n x$ can be calculated in parallel using unordered GFFTs (UGFFTs) with shifts dependent on the unique process IDs:

$$G_n^{(s)} x^{(s)} = U_{n/p} R_{n/p} T^{s/p}_{n/p} R_{n/p} x^{(s)}.$$

The BSP FFT algorithm can handle $p > \sqrt{n}$ as well, by decomposing $G_n^{(s)}$ into multiple UGFFTs and introducing extra communication phases that redistribute x to group-cyclic distributions with increasing cycles. We refer to Inda and Bisseling [1,12] for details.

The above factorisation of $G_n^{(s)}$ adds n/p complex multiplications, but enables the use of highly optimised sequential UFFT codes. However, state-of-the-art FFT libraries such as FFTW [7] or Spiral [17] do not provide such kernels by default, instead exposing only kernels that compute the regular FFT $F_n x$. Since $R_n = R_n^{-1}$, computing $U_n x$ may then be replaced by computing $F_n R_n x$; this enables the use of FFTW at the cost of additional bit-reversals. Algorithm 4 sketches the resulting BSP FFT implementation for $p \leq \sqrt{n}$; the code we used in experiments handles $p > \sqrt{n}$ as well. This approach always outperforms using the straightforward sequential UFFT kernel from the original BSPedupack implementation.[4]

[4] See http://www.math.uu.nl/people/bisseling/Software/software.

Algorithm 4 A BSP FFT algorithm using FFTW

```
void bspfft( double *x, unsigned long int n, signed char sign, double *tw,
             size_t *rho_k1, size_t *rho_p, fftw_plan consec, fftw_plan one ) {
//x is a complex vector of length n; this is both the input and output vector,
//sign           (1 or -1) indicates a forward or backward (inverse) FFT,
//tw             are the pre-computed weights for the twiddling (Eq. 3),
//rho_p, rho_np  the bit-reversal permutation of length p and n/p, respectively.
//consec, one    the FFTW plans for computing n/(p*p) consecutive FFTs of
//               length p on x and for computing one FFT of length n/p on x,
//               respectively. Both plans depend on the given sign.
        const unsigned int p  = bsp_nprocs();
        const unsigned int s  = bsp_pid();
        const size_t np = n / p; //local vector size

        permute( x, np, rho_np ); //perform local bit-reversal

        for( size_t r = 0; r < np/p; ++r )  //partial undo of bit-reversal,
                permute( x + r, p, rho_p ); //to enable the use of regular
        fftw_execute( consec );             //FFTs instead of UFFTs

        //Go from block to cyclic distribution
        const size_t size  = np / p;             //send a complex vector of
        const size_t SZCPL = 2 * sizeof(double); //length np/p to all other
        double *tmp = malloc( size * SZCPL );    //processes
        for( unsigned int j = 0; j < p; j++ ) {
                //get index of the j-th local element in a block distribution
                const size_t jglob = rho_p[ s ] * np + j;

                //get location in cyclic distribution
                const unsigned int destproc = jglob % p;
                const size_t destindex = jglob / p;

                //buffer all local complex elements at distance p
                for( size_t r = 0; r < size; r++ ) {
                        tmp[ 2*r ] = x[ 2*(j+r*p) ];
                        tmp[2*r+1] = x[2*(j+r*p)+1];
                }

                //put at destination vector
                bsp_put( destproc, tmp, x, destindex*SZCPL, size*SZCPL );
        }

        bsp_sync();  //perform the redistribution,
        free( tmp ); //and free the buffer

        //Perform remaining UGFFT (Eq. 3)
        twiddle( x, np, sign, tw ); //twiddle reduces the UGFFT to an UFFT
        permute( x, np, rho_np );   //undo of bit-reversal reduces the
        fftw_execute( one );        //UFFT to an optimised regular FFT

        //Apply scaling in case of backward transform
        if( sign == -1 )
                for( size_t r = 0; r < 2 * np; ++r )
                        x[ r ] /= (double)n;
}
```

3.2 Sparse Matrix–Vector Multiplication

Yzelman and Roose [22] show the benefit of explicitly distributing an $m \times n$ sparse matrix A when parallelising the SpMV product $y = Ax$ for shared-memory architectures. They consider various one-dimensional methods that distribute A row-wise. One of these is a fine-grained parallelisation scheme implemented in Cilk, the Compressed Sparse Blocks (CSB) [3] scheme. They also introduce a new 1D method which distributes rows of A in exactly p contiguous parts. Greedily balancing the number of nonzeroes in each part induces load-balance during parallel computation, assuming p equals the number of cores of the target machine. Since each processor core handles exactly one part, that part of A with the corresponding part of the output vector y can be allocated within the fast processor-local memory. This is relevant to multi-socket non-uniform memory access (NUMA) architectures, where on-socket data movement (using local memory banks) is faster than inter-socket data movement. The resulting method furthermore applies cache-oblivious optimisation strategies, and is implemented in PThreads. Both 1D methods do not require inter-process communication, nor do they require barrier synchronisations to complete a multiplication. These methods were tested as the two best performing state-of-the-art algorithms [22].

The same paper considers two-dimensional methods, in which individual nonzeroes of A and elements of the vectors x and y are distributed amongst the available processes. As a result, rows and columns of A may become shared amongst these processes, causing explicit communication. Sparse matrix partitioning software such as Mondriaan [21] partitions A into p disjoint parts, while minimising the cost μ of inter-process communication. The partitioner allows a maximum load imbalance $\epsilon \cdot nz/p$, with $\epsilon > 0$ a user-defined parameter and nz the number of nonzeroes in A. The benefit of applying this distributed-memory approach on shared-memory architectures, is that local parts of A can be allocated within processor-local memory, together with their corresponding parts of x and y; no data elements need to be accessible from all processes, and slow inter-socket data movement only occurs on inter-process communication, which is explicitly minimised during partitioning.

The 2D parallel multiplication itself proceeds in three supersteps [1, Chapter 4]. First, input elements required but not locally available are requested from remote processes (fan-in). The second superstep consists of a local SpMV multiplication, after which locally computed output elements that should be stored at remote processes are sent out (fan-out). The final step adds all incoming remote contributions to the local output vector. By using the MulticoreBSP `bsp_direct_get`, only two supersteps are required [24]. Both versions incur $\Theta(m+n)$ additional data movement to cope with the fan-in and fan-out steps. Yzelman and Roose reduce this overhead to $\Theta(\mu)$ [22] by exploiting the doubly Separated Block Diagonal (SBD) form of sparse matrices [23]. Reordering of A to obtain a doubly SBD form can be done simultaneously with partitioning. We obtain the MulticoreBSP for C implementation in Algorithm 5 by performing the local SpMV multiplication using the Compressed BICRS data structure [22] with nonzeroes in row-major order. We further reduce data movement during the fan-in and fan-out steps by transferring ranges of input and output vector elements (instead of communicating element-by-element). Note the use of the new

bsp_hpsend function as an additional improvement, and that this example is written in pure C++.

Algorithm 5 The BSP 2D cache-oblivious $z = Ax$ SpMV multiplication

```
#include <mcbsp.hpp> //The header file for the C++ wrapper (Sect. 2.3)

struct fanQuadlet {
        //Encodes a single fan-in or fan-out message: remote process ID,
        //local start index, remote start index, and the message length
        unsigned long int remoteP, localI, remoteI, length;
};

void BSP_SpMV_2D::mv() {
        //fan-in step; fanIn is a vector containing fanQuadlet structs
        for( size_t i = 0; i < fanIn.size(); ++i ) {
                bsp_direct_get( fanIn[i].remoteP, x,
                    fanIn[i].remoteI * sizeof(double), x + fanIn[i].localI,
                    fanIn[i].length  * sizeof(double) );
        }

        //local optimised SpMV; A is stored in Compressed BICRS form
        if( A != NULL ) A->zax( x, z ); //'zax' stands for z=Ax

        //fan-out step; the tag contains two unsigned long ints: the remote
        //              start index (remoteI), and the message length.
        //              fanOut is a vector containing fanQuadlet structs
        for( unsigned long int i = 0; i < fanOut.size(); ++i ) {
                bsp_hpsend( fanOut[i].remoteP, &( fanOut[i].remoteI ),
                    z + fanOut[i].localI, fanOut[i].length * sizeof(double) );
        }

        //sync to ensure fan-out is done
        bsp_sync();

        //collect remote contributions
        double *msg_payload;
        unsigned long int *msg_tag;
        while( bsp_hpmove((void**)&msg_tag, (void**)&msg_payload) != SIZE_MAX )
                for( unsigned long int i = 0; i < msg_tag[ 1 ]; ++i )
                        z[ msg_tag[ 0 ] + i ] += msg_payload[ i ];
}
```

4 Experiments

For experiments on distributed-memory architectures, we use a cluster named Lynx, located at the ExaScience Lab in Leuven, Belgium. Lynx has 32 nodes, connected by a two-switch Infiniband network, with each node containing two 6-core Intel Xeon X5660 processors. For experiments on shared-memory architectures, we use two machines: (1) a 64-core, 8-socket HP DL980 with eight 8-core Intel Xeon E7-2830 processors, and (2) a 40-core, 4-socket HP DL580 with four 10-core Intel Xeon E7-4870 processors. When the number of BSP processes requested is lower than the actual number of cores, processes may be allocated on as few sockets as possible

(compact affinity), or evenly spread over all sockets instead (scattered affinity). Compact affinity minimises inter-socket data movement, while scattered affinity maximises the total available bandwidth. MulticoreBSP for C also supports manual affinity strategies.

The FFT experiments in Sect. 4.1 are set up to run with both the MulticoreBSP for C and the BSPonMPI communication libraries; the computational kernels used do not change. This enables comparison between shared-memory and distributed-memory systems (the DL980 and Lynx, respectively). By running both variants on the DL980, the communication libraries themselves may be compared as well. Experiments for the SpMV multiplication in Sect. 4.2 compare different algorithms on various parallelisation frameworks. This checks whether the BSP 2D SpMV indeed attains state-of-the-art performance. We also provide a PThreads implementation of the same 2D SpMV algorithm, derived from the original code from Yzelman and Roose [22]. The 1D PThreads and all 2D methods employ the same sequential kernel for local SpMV multiplication.

All software used in the experiments is freely available. The PThreads 1D and 2D SpMV multiplication codes are included with the Sparse Library,[5] and the updated BSP FFT and the BSP 2D SpMV multiplication codes are available stand-alone.[6] The BSP FFT code can be compiled with support for FFTW3 [7] and Spiral [17]. The MulticoreBSP for C library and BSPonMPI library are freely available as well.[7] All preceding software was compiled using GCC 4.4.3. We used the 2011 Cilk CSB code[8] and compiled it with ICC 13.0.1.

4.1 Fast Fourier Transformation

We perform 300 forward and backward FFTs and average the execution time. We compare this result against a sequential FFT using FFTW 3.3.3.[9] All backward FFTs are performed without scaling, since this is not done automatically within FFTW. Table 1 shows the speedups for $n = 2^{26}$ on both Lynx and the DL980. We use BSPonMPI 0.3 with MVAPICH 1.8.1 [15] on both architectures. MulticoreBSP for C is used on the shared-memory DL980 only. We also compare the effect of compact and scattered affinity strategies. Figure 1 shows results for $2^9 \leq n \leq 2^{26}$ with $p = 64$ on the DL980. Note that there is no difference between a scattered or a compact affinity in this case.

The case of $p = 1$ in Table 1 shows the factor-two overhead of the multiple bit-reversals in our BSP FFT implementation; using a sequential UFFT kernel would completely eliminate this initial slowdown. For $p = 2$, the overhead of synchronisation and data redistribution tempers the initial speedup. Higher values of p incur no additional overheads and may scale up freely.

[5] See http://albert-jan.yzelman.net/software/#SL.

[6] See http://albert-jan.yzelman.net/software/#HPBSP.

[7] Via http://www.multicorebsp.com and bsponmpi.sourceforge.net, respectively.

[8] See http://gauss.cs.ucsb.edu/~aydin/software.html.

[9] Auto-tuning proceeds in the `FFTW_PATIENT` mode.

Table 1 Speedups of the BSP FFT on a vector of length $n = 2^{26}$ relative to a sequential FFTW time of 3.4 (Lynx) or 3.9 (DL980) seconds

Machine	$p = 1$	$p = 2$	$p = 4$	$p = 8$	$p = 16$	$p = 32$	$p = 64$
Lynx (BSPonMPI)	0.6	0.8	1.7	3.4	7.2	15.2	24.4
DL980 (BSPonMPI)	0.5	0.7	1.1	1.7	3.3	6.2	10.5
DL980 (McBSP compact)	0.5	0.7	1.0	1.2	2.6	6.3	13.1
DL980 (McBSP scattered)	0.5	0.7	1.3	2.6	5.1	10.8	13.7

For the DL980, the table also compares BSPonMPI versus MulticoreBSP for C (McBSP), as well as the two affinity strategies

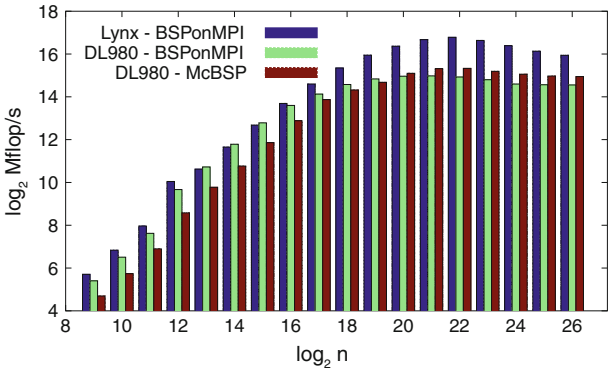

Fig. 1 Speed of the BSP FFT for $p = 64$ and varying n. Note that both axes are in logarithmic scale; the highest measured speed is 112 Gflop/s on Lynx for $n = 2^{22}$

Lynx demonstrates a superlinear relative speedup for $4 \leq p \leq 32$ every time the number of processes doubles. For $p = 64$ this reduces because a single node then contains two BSP processes. The relative speedups of the DL980 with compact affinity are superlinear for $16 \leq p \leq 64$. With $p = 4$ and 8, the shared memory hierarchy slows down execution; threads contend with each other for caches and for the available bandwidth. After $p = 8$, processes start to use other sockets. They communicate through the slower inter-socket interconnect, retain the relative local efficiency of the computation phases, and make use of the additionally available caches. Scattered affinity behaves similarly with a sublinear relative speedup when doubling p from 32 to 64; this coincides with doubling the number of processes per socket from 4 to 8 in the compact affinity. For $4 \leq p \leq 32$ good relative speedups are attained, which are but a constant factor apart from those attained on Lynx.

Figure 1 indicates that this BSP algorithm requires large problem sizes before parallelisation with $p = 64$ becomes efficient. For small n, BSPonMPI performs better than MulticoreBSP for C, while for larger n MulticoreBSP performs better. Small vectors fit into cache and benefit from the corresponding higher bandwidths, resulting in high compute speeds. As the lower-bandwidth L3 cache is referenced more often (i.e., as n increases), the calculation speed decreases until most of the vector is streamed from main memory. Lynx has a combined L3 cache size of $64 \cdot 12 = 768$ MB,

Table 2 Sparse matrices used in SpMV multiplication

Matrix	Rows	Columns	Nonzeroes	Origin
Freescale1	3,428,755	3,428,755	17,052,626	Semiconductor industry
ldoor	952,203	952,203	42,493,817	Structural engineering
cage15	5,154,859	5,154,859	99,199,551	DNA electrophoresis
adaptive	6,815,744	6,815,744	27,248,640	Numerical simulation
road_usa	23,947,347	23,947,347	57,708,624	Road network
wiki2007	3,566,907	3,566,907	45,030,389	Link matrix

while the DL980 has $8 \cdot 24 = 192$ MB. These caches can store vectors of length 2^{25} and 2^{23}, respectively,[10] but no larger powers of two. Figure 1 indeed shows that the speed stabilises for Lynx and the DL980, as the values of n exceed their respective L3 vector lengths. Although our current BSP FFT implementation performs more bit-reversals than is theoretically optimal, it still outperforms the parallel shared-memory FFT provided by FFTW[11] for $32 \leq p \leq 64$ and $2^{15} \leq n \leq 2^{26}$ on the DL980; e.g., for $p = 64$ and $n = 2^{26}$, the BSP FFT attains a speed of 31 Gflop/s while parallel FFTW attains 22 Gflop/s.

4.2 Sparse Matrix–Vector Multiplication

We use a subset of large matrices from Yzelman and Roose [22] to test the performance of two implementations of the BSP 2D SpMV multiplication. One implementation uses the new `bsp_hpsend` and `bsp_direct_get` primitives, while the other uses the regular BSPlib primitives. Both are tested with on the DL580 and DL980 machines. Table 2 shows the sparse matrices we use, which are preprocessed using Mondriaan [21] version 3.11 with $\epsilon = 0.03$ and SBD reordering enabled. We compare their performance to that of the Cilk CSB and the PThreads 1D SpMV multiplication methods discussed in Sect. 3.2, and to that of a PThreads implementation of Algorithm 5.

We take the average execution time of 1,000 SpMV multiplications for each matrix while varying the number of threads on the DL580 and DL980 machines. For each matrix and machine, we compare timings with a sequential SpMV multiplication using the standard Compressed Row Storage (CRS) scheme. To maximise bandwidth use, we employ a scattered affinity in all experiments. The results in Table 3 show significantly higher speedups for the 2D methods compared to the competing methods. The *cage15* matrix is a notable exception, but this matrix is known to be hard to partition [22].

Note that the non-`hp` BSP 2D SpMV usually outperforms the PThreads 2D SpMV implementation. While the fan-out step is similar and the local SpMV multiplication step uses the same computational kernel, the fan-in step does contain a major implementation difference. In the BSP implementation, processes 'put' non-local contribu-

[10] 2^{26} complex values take 16 bytes each, resulting in 2^{30} bytes of storage. This equals 1 GB of data. A 2^{24}-length vector thus takes 256 MB.

[11] FFTW 3.3.3 linked against `fftw3_threads`.

Table 3 Maximum speedups for the SpMV multiplication on a 40-core HP DL580 and a 64-core HP DL980 using Cilk, PThreads, and BSP implementations

	Seq. speed (Mflop/s)	Cilk CSB	PThreads 1D	PThreads 2D	BSP 2D	BSP (hp) 2D
DL580						
Freescale1	396	12.2 (30)	16.7 (40)	16.3 (32)	17.3 (40)	18.3 (40)
ldoor	298	24.3 (40)	18.5 (40)	15.9 (40)	16.1 (40)	16.2 (40)
cage15	482	12.9 (40)	14.6 (40)	12.6 (40)	13.0 (40)	13.8 (40)
adaptive	124	28.7 (40)	13.8 (40)	19.5 (30)	22.3 (40)	23.3 (40)
road_usa	95	26.0 (40)	14.5 (40)	23.0 (32)	25.3 (40)	25.9 (40)
wiki2007	191	22.8 (40)	22.1 (40)	22.0 (40)	19.2 (40)	22.8 (40)
DL980						
Freescale1	435	16.4 (64)	15.8 (64)	14.6 (56)	20.7 (64)	22.7 (64)
ldoor	341	20.2 (64)	15.3 (64)	15.2 (64)	17.6 (64)	18.8 (64)
cage15	540	17.7 (64)	16.1 (64)	9.3 (56)	13.8 (56)	13.5 (56)
adaptive	123	18.0 (64)	19.2 (64)	23.3 (40)	34.5 (64)	36.3 (64)
road_usa	93	15.1 (64)	19.2 (64)	31.2 (56)	43.4 (64)	46.6 (64)
wiki2007	178	21.6 (64)	27.7 (64)	25.1 (64)	23.5 (64)	29.7 (64)

The number of processes for which the reported maxima are attained appears in parentheses. The speed of sequential SpMV multiplication in plain CRS is added for reference

tions into remote BSMP queues in one phase, and add remote contributions locally in the following phase. In the PThreads implementation, processes synchronise once, and then 'get' remote contributions from remote memory and immediately add the value locally. The results indicate that BSP programming accelerates computation as it induces greater data locality.

The 2D methods exhibit better scalability than the 1D methods. The parallel efficiencies[12] of the 1D methods are lower on the highly NUMA DL980 machine than on the DL580, while the efficiencies for the 2D methods remain similar. Consider for example the road_usa matrix. The parallel efficiency of the 1D PThreads method decreases from 36 to 30 % as we move from the DL580 to the DL980. The 2D `hp`-BSP version attains a higher efficiency of 64 % on the DL580, which furthermore increases to 73 % on the DL980. Yzelman and Roose already noted that as the NUMA complexity of architectures increases, 2D methods will outperform 1D methods [22]. We now indeed achieve these predicted results, due to the improved scalability of barrier synchronisation; MulticoreBSP for C uses spin-locks instead of mutex-based synchronisation since version 1.1, as do the updated 1D and 2D PThreads implementations.

5 Conclusion

We propose an update to the BSPlib standard, which includes two new high-performance primitives. The proposed interface has been implemented in Multi-

[12] The parallel efficiency of an algorithm is defined as $\frac{\text{speedup}}{p}$.

coreBSP for C, written specifically for shared-memory architectures. We ran an existing distributed-memory BSP algorithm for the Fast Fourier Transform on a shared-memory machine, and showed that the algorithm behaves similarly as on a distributed-memory cluster. Differences were explained by comparing the compact and scattered affinities on a highly-NUMA architecture. The algorithm exceeded the performance of shared-memory parallel FFTW code for large vectors and a large number of processors in a highly-NUMA configuration. We demonstrated the use of the new high-performance primitives by extending a state-of-the-art 2D SpMV multiplication algorithm. This extended algorithm tested faster than the non-`hp` version in all experiments but one. The BSP 2D SpMV implementation exceeded the performance of a PThreads implementation, and that of other state-of-the-art SpMV multiplication kernels as well.

5.1 Future Work

While we only investigated two problems here, the ratio of flops per data element for the FFT and the SpMV multiply is low; hence the communication library has a great impact on algorithm performance. Nevertheless, future research should involve a wider scope of applications. Further optimisation of the MulticoreBSP for C library is warranted as well, as it was outperformed by BSPonMPI for the FFT on small input vectors.

The FFT algorithm can be further improved by using an optimised sequential kernel for the unordered FFT. A comparison with techniques for multi-threaded FFT [6] may suggest further improvements, or indicate whether there are limits to the applicability of BSP in high-performance shared-memory computing. The SpMV algorithms used here are high-level and would benefit from known low-level optimisations for the SpMV multiplication [4].

NUMA issues demand a good strategy for distributing BSP processes over the available hardware. This does not fit into the BSP model since the processor interconnect is assumed uniform. Since MulticoreBSP for C supports nested BSP runs, the exploration of algorithms designed in the Multi-BSP model [20] is a next logical step. The FFT algorithm is an especially good candidate to re-implement using nested SPMD sections; instead of calling sequential UFFT (or FFTW) kernels, we can recursively call the same BSP FFT implementation. In this fashion, we may first distribute the computation over the available sockets of a machine, upon which each top-level BSP process delegates its local UFFTs to a parallel BSP FFT that runs within its assigned socket.

Although this increases the number of synchronisations and data redistributions to at least three, the top-level and bottom-level values for g and l are typically lower than the (g, l) corresponding to the entire machine. Depending on these machine parameters, and on the input vector length, the BSP cost model can furthermore decide when a Multi-BSP FFT is preferred over a flat BSP version.

Acknowledgments Part of this work is funded by Intel and by the Institute for the Promotion of Innovation through Science and Technology in Flanders (IWT).

Appendix: Semantics of BSP Primitives

This appendix briefly describes each BSP primitive listed in Sect. 2.

- `void bsp_init( void(*spmd)(void),int argc,char **argv )`. Indicates which function is the entry-point of the parallel SPMD part of the program; new processes spawned by `bsp_begin` will start at the function pointed to by the `spmd` parameter. `bsp_init` must be called before `bsp_begin`, unless the SPMD section is the C main-function itself. The `spmd` function should have `bsp_begin` as its first executable statement, and `bsp_end` as its last. The parameters `argc` and `argv` should correspond with those supplied to the C main function. MulticoreBSP does not require them for successful initialisation, but other BSP libraries might; the arguments are retained for portability.
- `void bsp_begin( unsigned int P )`. Indicates the start of an SPMD section using `P` processes. This should be the first executable statement of the function designated as the SPMD entry-point. The process that initially calls this function spawns $\texttt{P}-1$ sibling processes each with a unique BSP ID from $\{1, 2, \ldots, \texttt{P}-1\}$, and then joins the same SPMD group with BSP ID 0; the calling process will thus retain all its process-local data. Only code between `bsp_begin` and `bsp_end` may call the DRMA or BSMP communication primitives listed in Sect. 2.
- `void bsp_end()`. The last statement of a parallel SPMD section, thus necessarily following a `bsp_begin`. Threads with BSP ID larger than 0 will terminate after calling this function. The process with ID 0 will continue sequentially with any remaining statements. Subsequent calls to `bsp_init` and `bsp_begin` can be used to start other parallel SPMD sections.
- `unsigned int bsp_nprocs()`. When called outside an SPMD environment, `bsp_nprocs()` returns the number of hardware-supported processes. When called within an SPMD environment, it returns the number of processes involved with the current parallel SPMD run.
- `unsigned int bsp_pid()`. Returns the BSP ID of the current process. The process-unique integer returned is between 0 (inclusive) and `bsp_nprocs` (exclusive). Calling `bsp_pid` outside an SPMD section raises a run-time error.
- `void bsp_sync()`. Signals the end of the current computation phase and starts a global synchronisation. It then starts a BSP communication phase which executes all communication requests made prior to calling `bsp_sync()`. The communication phase is then followed by another global synchronisation. This guarantees that all communication requested in the previous computation phase is executed before starting the next computation phase, which starts with code following this `bsp_sync`. All processes should issue an equal number of `bsp_syncs`; otherwise, a mismatched `bsp_end` and `bsp_sync` will raise a run-time error.
- `void bsp_abort( char *error, ... )`. This will halt parallel execution at the earliest opportunity, which may be earlier than the end of the current computation phase. The format of the `error` message equals that of the standard C *printf* function, and takes a variable number of parameters.
- `double bsp_time()`. Returns the elapsed time since the start of the current process within the current SPMD section. The time returned is in seconds and

is in high precision. Timers among the various SPMD processes need not be synchronised. MulticoreBSP depends on the POSIX realtime extension to deliver high-resolution timers.

- `void bsp_push_reg( void *address, size_t size )`. Registers the memory area defined by its starting `address` and its `size` (in bytes) for DRMA communications, after the next `bsp_sync`. The order of registration of variables must be the same across all SPMD processes, but the `size` parameter may differ across processes. Registering a `NULL` pointer indicates that the current process will never communicate using the registered remote addresses. Multiple registrations of the same addresses are allowed; newer ones will (temporarily) replace the older registrations.
- `void bsp_pop_reg( void *address )`. Removes the registration of a variable previously registered using `bsp_push_reg`. Like registration, this only takes effect in the next superstep. The order of deregistration has to match across all SPMD processes. If the same variable was registered several times (while, e.g., using different values for the `size` parameter), `bsp_pop_reg` removes the last registration only.
- `void bsp_put( unsigned int pid, void *source, void *destination, size_t offset, size_t size )`. Copies the local data from address `source` up to and including `source + size − 1` into the memory of process `pid`. The destination address is determined by the previously registered `destination` variable, at the given `offset` (with the size and offset in bytes). Changing the source memory area after a `bsp_put` will not affect the communication request. Communication occurs during the next `bsp_sync`, ensuring remote availability upon exiting the synchronisation.
- `void bsp_get( unsigned int pid, void *source, size_t offset, void *destination, size_t size )`. Requests `size` bytes of data from the previously registered `source` variable at process `pid`, at the given `offset` (in bytes). The communication remains queued until the next `bsp_sync`, after which the requested data is locally available at address `destination`. Unlike `bsp_put`, the `bsp_get` does not (and cannot) buffer the requested data when called; the communicated data corresponds to the source memory area at the time the next `bsp_sync` was entered at process `pid`.
- `void bsp_set_tagsize( size_t *size )`. Each BSMP message has a tag to help receiving processes discern the purpose of that message. The amount of memory reserved for message tags is constant during supersteps, but can be changed from one superstep to the next by using this primitive. The new `size` of the BSMP tags is in bytes. All SPMD processes should request identical tag sizes, or BSP will abort. On function exit, `size` will be set equal to the old tag size.
- `void bsp_send( unsigned int pid, void *tag, void *payload, size_t size )`. Combines `size` bytes of data starting at the local address `payload` with the given `tag`, and sends this message to process `pid`. A `bsp_send` buffers the tag and payload; like `bsp_put`, the contents of the tag and payload may change after issuing a `bsp_send` without affecting the queued message. Processes receive BSMP messages in an unspecified order.

- `void bsp_qsize( unsigned int *packets, size_t *accumulated_size )`. Queries the size of the local BSMP queue, and sets `packets` to the number of messages received during the last communication phase. If `accumulated_size` is not `NULL`, it will be set to the combined size of all message payloads (in bytes).
- `void bsp_get_tag( size_t *status, void *tag )`. Retrieves the tag value from the first message in the BSMP queue and stores it in `tag`. Does not modify the BSMP queue. The variable corresponding to `status` will be set to the payload size of the first message. If there are no messages in the queue, it will instead be set to the highest possible value `size_t` can take. (In the original BSPlib interface, a signed integer -1 was returned instead.)
- `void bsp_move( void *payload, size_t max_copy_size )`. Removes the first message from the local BSMP queue, while copying the data into `payload`. At most `max_copy_size` bytes are copied.
- `void bsp_hpput( unsigned int pid, void *source, void *destination, size_t offset, size_t size )`. Copies the local data from address `source` up to and including `source + size − 1` into the memory of process `pid`. The destination address is determined by the previously registered `destination` variable, at the given `offset` (with the size and offset in bytes). Unlike `bsp_put`, transmitted data is not buffered, and the requested communication may occur at any time between calling this primitive and the end of the next `bsp_sync`; the source memory area must remain unchanged during the current superstep, while the destination memory area may only be changed by this communication request.
- `void bsp_hpget( unsigned int pid, void *source, size_t offset, void *destination, size_t size )`. Requests `size` bytes of data from the previously registered `source` variable at process `pid`, at the given `offset` (in bytes). Unlike `bsp_get`, the requested communication may occur at any time between calling this primitive and the end of the next `bsp_sync`; the source memory area must remain unchanged during the current superstep, while the destination memory area may only be changed by this communication request.
- `void bsp_hpsend( unsigned int pid, void *tag, void *payload, size_t size )`. Combines `size` bytes of data starting at the local address `payload` with the given `tag`, and sends this message to process `pid`. Unlike `bsp_send`, transmitted data is not buffered, and the requested communication may occur at any time between calling this primitive and the end of the next `bsp_sync`. The message is available for inspection at the destination process after the next `bsp_sync`. The source memory area must remain unchanged during the current superstep. Processes receive BSMP messages in an unspecified order.
- `size_t bsp_hpmove( void **p_tag, void **p_payload )`. Removes the first message from the local BSMP queue, sets the provided pointers to the tag and payload memory addresses inside the incoming BSMP buffer, and returns the payload size (in bytes). Pointers will remain valid until the end of this computation phase. Only messages sent in the previous superstep are available for

inspection. If the local queue is empty, the primitive returns the maximum `size_t` value (`SIZE_MAX`). Note that this differs from the original BSPlib where a signed integer with value -1 was returned instead. The message resides in memory managed by the BSP run-time system: message data must not be freed manually, and erroneously touching data outside of the message bounds may result in undefined behaviour of the run-time system.

– `void bsp_direct_get( unsigned int pid, void *source, size_t offset, void *destination, size_t size )`. Copies `size` bytes of data from the previously registered `source` variable at process `pid` at the given `offset` (in bytes), into local memory starting at address `destination`. Communication is executed immediately; this is a blocking one-sided communication which does *not* require a subsequent `bsp_sync` to ensure communication has completed. The source memory area must remain unchanged during the current computation phase.

References

1. Bisseling, R.H.: Parallel Scientific Computation: A Structured Approach Using BSP and MPI. Oxford University Press, Oxford (2004)
2. Bonorden, O., Juurlink, B., von Otte, I., Rieping, I.: The Paderborn University BSP (PUB) library. Parallel Comput. **29**(2), 187–207 (2003)
3. Buluç, A., Fineman, J.T., Frigo, M., Gilbert, J.R., Leiserson, C.E.: Parallel sparse matrix-vector and matrix-transpose-vector multiplication using compressed sparse blocks. In: SPAA'09: Proceedings of the Twenty-first Annual Symposium on Parallelism in Algorithms and Architectures, pp. 233–244. ACM, New York, NY (2009)
4. Buluç, A., Williams, S., Oliker, L., Demmel, J.: Reduced-bandwidth multithreaded algorithms for sparse matrix-vector multiplication. In: International Parallel and Distributed Processing Symposium (IPDPS), pp. 721–733. IEEE Press, Piscataway, NJ (2011)
5. De la Torre, P., Kruskal, C.P.: Submachine locality in the bulk synchronous setting. In: Bougé, L., Fraigniaud, P., Mignotte, A., Robert, Y. (eds.) Euro-Par'96 Parallel Processing. Lecture Notes in Computer Science, vol. 1124, pp. 352–358. Springer, Berlin (1996)
6. Franchetti, F., Püschel, M., Voronenko, Y., Chellappa, S., Moura, J.M.F.: Discrete Fourier transform on multicore. IEEE Signal Process. Mag., special issue on Signal Processing on Platforms with Multiple Cores. **26**(6), 90–102 (2009)
7. Frigo, M.: A fast Fourier transform compiler. In: Proceedings of the ACM SIGPLAN 1999 Conference on Programming Language Design and Implementation, PLDI'99, pp. 169–180. ACM, New York, NY (1999)
8. Hamidouche, K., Falcou, J., Etiemble, D.: Hybrid bulk synchronous parallelism library for clustered SMP architectures. In: Proceedings Fourth International Workshop on High-level Parallel Programming and Applications, pp. 55–62. ACM, New York, NY (2010)
9. Hill, J.M.D., McColl, B., Stefanescu, D.C., Goudreau, M.W., Lang, K., Rao, S.B., Suel, T., Tsantilas, T., Bisseling, R.H.: BSPlib: the BSP programming library. Parallel Comput. **24**(14), 1947–1980 (1998)
10. Hinsen, K.: High-level parallel software development with Python and BSP. Parallel Process. Lett. **13**(03), 473–484 (2003)
11. IEEE: Std. 1003.1-2008 Portable Operating System Interface (POSIX) Base Specifications, Issue 7. IEEE Standards for Information Technology. IEEE Press, Piscataway, NJ (2008)
12. Inda, M.A., Bisseling, R.H.: A simple and efficient parallel FFT algorithm using the BSP model. Parallel Comput. **27**(14), 1847–1878 (2001)
13. Javed, N., Loulergue, F.: OSL: Optimized bulk synchronous parallel skeletons on distributed arrays. In: Dou, Y., Gruber, R., Joller, J. (eds.) Advanced Parallel Processing Technologies. Lecture Notes in Computer Science, vol. 5737, pp. 436–451. Springer, Berlin (2009)
14. Keßler, C.W.: NestStep: nested parallelism and virtual shared memory for the BSP model. J. Supercomput. **17**, 245–262 (2000)

15. Liu, J., Wu, J., Panda, D.K.: High performance RDMA-based MPI implementation over Infiniband. Int. J. Parallel Program. **32**(3), 167–198 (2004)
16. Loulergue, F., Gava, F., Billiet, D.: Bulk synchronous parallel ML: Modular implementation and performance prediction. In: International Conference on Computational Science, Part II, Lecture Notes in Computer Science, vol. 3515, pp. 1046–1054. Springer, Berlin (2005)
17. Püschel, M., Franchetti, F., Voronenko, Y.: Spiral. In: Encyclopedia of Parallel Computing. Springer, Berlin (2011)
18. Suijlen, W.: BSPonMPI. http://sourceforge.net/projects/bsponmpi/. Accessed 10 Feb 2013
19. Valiant, L.G.: A bridging model for parallel computation. Commun. ACM **33**(8), 103–111 (1990)
20. Valiant, L.G.: A bridging model for multi-core computing. J. Comput. Syst. Sci. **77**(1), 154–166 (2011)
21. Vastenhouw, B., Bisseling, R.H.: A two-dimensional data distribution method for parallel sparse matrix-vector multiplication. SIAM Rev. **47**(1), 67–95 (2005)
22. Yzelman, A.N., Roose, D.: High-level strategies for parallel shared-memory sparse matrix-vector multiplication. IEEE Trans. Parallel Distrib. Syst. (2013, in press)
23. Yzelman, A.N., Bisseling, R.H.: Two-dimensional cache-oblivious sparse matrix-vector multiplication. Parallel Comput. **37**(12), 806–819 (2011)
24. Yzelman, A.N., Bisseling, R.H.: An object-oriented bulk synchronous parallel library for multicore programming. Concurr. Comput. Pract. Exper. **24**(5), 533–553 (2012)