MulticoreBSP for C: a high-performance library for shared-memory parallel programming

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

Report TW 624, March 2013

KU Leuven
Department of Computer Science
Celestijnenlaan 200A – B-3001 Heverlee (Belgium)

MulticoreBSP for C: a high-performance library for shared-memory parallel programming

A. N. Yzelman
R. H. Bisseling
D. Roose
K. Meerbergen
Report TW 624, March 2013

Department of Computer Science, KU Leuven

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 also show that applications in MulticoreBSP can perform on-par with state-of-the-art counterparts written in other shared-memory parallel programming interfaces. We furthermore study the applicability of BSP when working on highly non-uniform memory access (NUMA) architectures.

Keywords : Sparse matrix–vector multiplication, shared-memory parallelism, cache-oblivious, sparse matrix partitioning, matrix reordering, Hilbert space-filling curve, high-performance computing, NUMA architectures.
MSC : Primary : 65Y05, Secondary : 65T50, 65F50, 68W10.

MULTICOREBSP FOR C: A HIGH-PERFORMANCE LIBRARY FOR SHARED-MEMORY PARALLEL PROGRAMMING

A. N. YZELMAN, R. H. BISSELING, D. ROOSE, AND K. MEERBERGEN

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 also show that applications in MulticoreBSP can perform on-par with state-of-the-art counterparts written in other shared-memory parallel programming interfaces. We furthermore study the applicability of BSP when working on highly non-uniform memory access (NUMA) architectures. high-performance computing and bulk synchronous parallel and parallel programming and programming library

1. Introduction

The Bulk Synchronous Parallel (BSP) model [Val90], introduced by Valiant, describes a powerful abstraction of parallel computers. It enables the design of theoretically optimal parallel algorithms, and inspired many programming 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 with access to local memory. They cannot access remote memory, but may communicate through a black-box network interconnect. Synchronising the p processes using this network costs l units, while sending a data word during an all-to-all communication costs g units. Measuring l and g in seconds does not directly relate to the speed of the various processors; 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) thus 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. Each phase of either kind is a superstep, and BSP algorithms synchronise after each superstep.

This definition of a BSP computer and a BSP algorithm immediately yields 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 time spent in computation 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 reception of messages form the bottleneck of communication. Writing $r_i^{(s)}$ for the number of words received by process s during superstep i, and $t_i^{(s)}$ for the number of words transmitted (sent), the communication cost is $g \cdot \sum_{i=0}^{T-1} h_i$ with $h_i = \max\{\max_s r_i^{(s)}, \max_s t_i^{(s)}\}$ the h-relation of the ith superstep. Accounting for the $T-1$ barrier synchronisations yields the full BSP cost:

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

For brevity, 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. By contrast, a situation where each process sends and receives exactly 1 word (e.g., a pairwise swap or a one-directional ring exchange), results in a perfectly balanced 1-relation independent of p.

2. The MulticoreBSP programming interface

The MulticoreBSP for C programming interface adheres to the ANSI C99 standard, directly derives from the BSPlib [HMS+98] interface by Hill et al., and is inspired by the Java MulticoreBSP library by Yzelman and Bisseling [YB12]. 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 [IEE08]. 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; Section 2.7 discusses these.

The updated interface consists of 22 primitive function calls. For each primitive, we now state the (shortened[1]) interface declarations, the asymptotic run-time complexity, and a brief description. The reported time complexities are attained by MulticoreBSP for C and may help users better understand what to expect when calling BSP primitives. Those in big-Theta notation indicate that a lower asymptotic bound is not possible, while those in big-Oh notation leave room for improvement. Section 2.6 highlights the differences with respect to the original BSPlib interface. For brevity, we still refer to the updated interface as the BSPlib interface.

2.1. Single program multiple data (SPMD).

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

- `void bsp_init( void (*spmd)(void), int argc, char **argv )`; $\Theta(1)$. 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. This function 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 )`; $\mathcal{O}(\mathtt{P})$. 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 $\mathtt{P}-1$ sibling processes with BSP IDs in $\{1, 2, \ldots, \mathtt{P}-1\}$, and then joins the same SPMD group with BSP ID 0; the calling process will thus retain all its process-local data. Only code in-between `bsp_begin` and `bsp_end` may use the communication primitives defined in Sections 2.2 and 2.3.

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

- `void bsp_end()`; $\mathcal{O}(l)$. 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()`; $\Theta(1)$. 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()`; $\Theta(1)$. Returns the BSP ID of the current process. The process-unique integer returned is in between 0 and `bsp_nprocs`. Calling `bsp_pid` outside an SPMD section raises a run-time error.

The MulticoreBSP 'hello-world' program in Algorithm 1 illustrates the use of these five primitives. Note that it consists of a single computation phase; in BSPlib all SPMD code is part of a computation phase. The communication primitives in Sections 2.2 and 2.3 therefore do not immediately initiate communication, but instead queue communication requests. These are processed after indicating the end of the current superstep:

- `void bsp_sync()`; $\Theta(l + g \cdot h_i)$, see Eq. 1. Signals the end of the current computation superstep and starts a global synchronisation. It then starts a BSP communication phase which executes all communication requests made prior to calling `bsp_sync()`. As per the BSP model, another synchronisation follows the communication phase. This guarantees that all communication requested in the previous computation phase is executed before starting the next computation superstep, 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.

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;
}
```

The following utility functions, like `bsp_sync()`, can only be called within SPMD sections.

- `void bsp_abort( char *error, ... )`; $\Theta(1)$. 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()`; $\Theta(1)$. 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.

2.2. **Direct remote memory access (DRMA).** BSPlib provides DRMA 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 inverse; it retrieves data from a remote memory area and copies it to local memory. If an SPMD program defines a local variable, however, 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)`; $\Theta(1)$. 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 )`; $\Theta(1)$. 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 de-registration 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.

Registration and de-registration 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 the following communication primitives:

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

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 (each of length p). The second computation superstep 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.

2.3. **Bulk-synchronous message passing (BSMP).** BSMP enables sending messages of arbitrary lengths to remote processes. A message will be sent during the `bsp_sync` following a BSP send primitive. Received messages will reside in a local BSMP queue afterwards. BSMP communication does not require registration. Its interface is as follows:

- `bsp_set_tagsize( size_t *size )`; $\Theta(1)$. Each BSMP message has a tag to help receiving processes discern the purpose of each message. The amount of memory reserved for message tags is constant during supersteps, but can be changed from one computation 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.

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 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 < P; ++k ) {
                bsp_put( k, &alpha, ip_buffer,
                         s*sizeof(double), sizeof(double) );
        }
        bsp_sync();
        for( unsigned int k = 1; k < P; ++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();
}
```

- `void bsp_send( unsigned int pid, void *tag, void *payload, size_t size )`; $\Theta($`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 )`; $\mathcal{O}($`packets`$)$. 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 )`; $\Theta(1)$. Retrieves the tag value from the first message in the BSMP queue and stores it in `tag`. Does not modify the queue. The variable corresponding to `status` will be set to the payload size of the first message, or to the highest possible value `size_t` can take if there are no messages left[2].

[2] in the original BSPlib interface -1 was returned instead.

- `void bsp_move( void *payload, size_t max_copy_size )`; Θ(`size`). Removes the first message from the local BSMP queue, while copying the data into `payload`. At most `max_copy_size` bytes are copied.

2.4. **High-performance variants.** BSPlib defines high-performance (`hp`) variants of DRMA and BSMP primitives. These are `bsp_hpput`, `bsp_hpget`, and `bsp_hpsend`, and allow for communication to occur somewhere between the call to the `hp`-primitive and the end of 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 of buffering communication. Users must guarantee that the source and destination memory areas remain unchanged until the end of the current superstep. Errors in the use of `hp`-variants may cause non-deterministic behaviour that cannot be caught by the BSP run-time; users should consider the added costs of ensuring correctness of their applications when thinking of using `hp`-primitives.

Although `bsp_hpsend` avoids buffering-on-send, messages still enter a receiving buffer: the BSMP queue. The `bsp_move` primitive copies messages from this buffer; the `size_t bsp_hpmove(void **p_tag, void **p_payload)` primitive avoids this buffer-on-receive by directly returning pointers to the tag and payload in the receive buffers. It also returns the payload size in bytes, or the largest possible value if the queue is empty. 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 [YB12, Section 2]. Its semantics is exactly that of the `bsp_hpget`, except that it guarantees that communication is done after the call to the primitive has finished. Like with `bsp_hpget`, the user must ensure that the remote data remains unchanged during the current superstep. Using the direct-get allows this superstep to be merged with the next one, if no other communication primitives were called, thus saving a synchronisation barrier. The `bsp_direct_get` is the only `hp`-primitive that runs in Θ(`size`) time instead of $\Theta(1)$ time.

2.5. **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 the avoidance of global synchronisations: a `bsp_sync` in the nested run will involve only p/c processes. When a 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 correspond to the original run over c processes, while all old data is retained; in particular, earlier variable registrations become valid again. Note that a synchronisation on this original top-level domain again involves only c processes, and that subset synchronisation is not a new concept for BSP; the PUB library [BJOR03] contains similar functionality.

2.6. **Changes with respect to BSPlib.** BSP primitives that existed in the original BSPlib take the same arguments semantically, but their 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 by the BSPlib standard.

One of the new primitives that MulticoreBSP for C defines, the `bsp_hpsend`, can also be used efficiently in distributed-memory settings. In contrast, the `bsp_direct_get` is only efficient on shared-memory architectures. Both additions are in the spirit of the BSPlib standard `hp`-variants; they allow for communication inside computation supersteps, thus breaking the BSP paradigm 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 applied with care.

2.7. **User-level implications of threading.** 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 [Bis04] run without modification under MulticoreBSP for C (with compatibility mode enabled).

2.8. **MulticoreBSP for C++.** The C language was chosen for this high-performance implementation of MulticoreBSP as it enables BSP programming in both C and C++. We provide a C++-specific header that automatically includes all BSP primitives described above, and additionally defines a `BSP_program` class that wraps the C interface. This abstract class defines two purely virtual functions: `void BSP_program::spmd()` and `BSP_program *BSP_program::newInstance()`. The first one will be the entry-point of the SPMD section, while the second one enables the BSP system to create a new class instance for each sibling process. The `void BSP_program::begin( unsigned int P = bsp_nprocs() )` function starts a BSP run corresponding to its class. This automatically wraps calls to `bsp_init`, `bsp_begin`, and `bsp_end`, and has the further advantage that all class-local variables remain local to the BSP processes[3]. The full object-oriented approach of the Java MulticoreBSP library [YB12] has not been ported; communicating C++ objects may require marshalling by the user.

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 [Bis04, Chapter 3]. The BSP FFT is run on a modern distributed-memory cluster using BSPonMPI [Sui], and on a shared-memory machine using MulticoreBSP for C, thus enabling a direct performance comparison.

To show that the library enables writing high-performance parallel codes, we create two BSP versions of the 2D sparse matrix–vector multiplication described from Yzelman and Roose [YR13]. One uses the new `bsp_hpsend` and `bsp_direct_get` primitives, while the other uses the regular BSP primitives. 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 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 computation; the first $n/2$ weights need only be multiplied by a constant -1 (a sign change). Further exploitation of symmetry yields higher-radix

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

formulations of the FFT, but the effectiveness deteriorates exponentially fast while the involved constants become increasingly costly to use.

Repeated application of the division 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:

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

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:

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

This factorisation of $G_n^{(s)}$ adds n/p complex multiplications, but this cost can be offset by using highly optimised sequential UFFT kernels.

By subdividing $G_n^{(s)}$ into multiple UGFFTs and introducing extra communication phases that redistribute x to group-cyclic distributions with increasing cycles, the BSP FFT algorithm can handle $p > \sqrt{n}$ as well; we refer to Bisseling [Bis04, Chapter 3] for details. A BSPlib implementation of this algorithm is freely available within BSPedupack[4].

As state-of-the-art FFT libraries such as Spiral [PFV11] and FFTW [Fri99] do not provide unordered variants by default, we rewrite the first computation superstep of the BSP FFT to use n/p regular FFTs of length p, instead of n/p UFFTs of length p. These are preceded by the partial bit-reversal

$$\prod_{i=q+1}^{m} \left(I_{n/2^i} \otimes S_{2^i}\right) x, \tag{4}$$

instead of the full bit-reversal $R_n x$. We employ the highly optimised FFT kernels from Spiral[5], but use the unoptimised UFFT implementation included with BSPedupack for the remaining supersteps; the majority of flops thus remain unoptimised. Algorithm 3 sketches the resulting BSP FFT implementation for $p \leq \sqrt{n}$. Note that Spiral offers streaming FFT kernels which need separate input and output vectors, which doubles the memory requirements.

3.2. **Sparse matrix–vector multiplication.** Yzelman and Roose [YR13] show the benefit of explicitly distributing an $m \times n$ sparse matrix A when parallelising the sparse matrix–vector (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) [BFF+09] 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 om multi-socket 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 [YR13].

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. This may cause that rows and columns of A become shared amongst these processes, causing explicit communication. Sparse matrix partitioning software such as Mondriaan [VB05] partitions A in 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 four supersteps [Bis04, Chapter 4]. First, input elements required but not locally available are requested from remote processes (fan-in). Then processes perform a local SpMV multiplication, and 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 three supersteps are required [YB12]. Both versions incurred $\Theta(m+n)$ extra data movement to cope with

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

[5]the freely available Spiral-generated DFT code from `http://www.spiral.net`.

Algorithm 3 A BSP FFT algorithm using Spiral

```
void bspfft( double *x, double *buffer, unsigned long int n, signed char sign,
             double *w , double *tw, size_t *rho_k1, size_t *rho_p ) {
//x is a complex vector of length n; this is both the input and output vector,
//buffer is a vector of length n required for using Spiral,
//sign   (1 or -1) indicates a forward or backward (inverse) FFT,
//tw, w  are the pre-computed weights for the twiddling, resp., UFFT (Eq. 3)
//rho_p  is the bit-reversal permutation of length p
//rho_k1 is the partial bit-reversal permutation of length n/p (Eq. 4)
        const unsigned long int p  = bsp_nprocs();
        const unsigned long int s  = bsp_pid();
        const unsigned long int np = n / p; //local vector size

        //Perform global permutes, leave local permutes to Spiral (Eq. 4)
        permute( x, buffer, np, rho_k1 );

        //Perform optimised FFTs
        for( unsigned long int r = 0; r < np/p; ++r )
                spiral_fft_double( (int)p, (int)(-sign), buffer+r*p, x+r*p );

        //Go from block to cyclic distribution
        const unsigned long int size  = np / p;
        const unsigned long int SZCPL = 2 * sizeof(double);
        double *tmp = malloc( size * SZCPL );

        //send a complex vector of length size to each process
        for( unsigned int j = 0; j < p; j++ ) {
                //get index of the j-th local element in a block distribution
                const unsigned long int 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 );
        }

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

        //Perform remaining UGFFT (Eq. 3)
        twiddle( x, np, sign, tw );
        ufft( x, np, sign, w );

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

the fan-in and fan-out steps. Yzelman and Roose [YR13] reduced this overhead to $\Theta(\mu)$ [YR13] by exploiting doubly Separated Block Diagonal (SBD) forms of sparse matrices [YB11]. Reordering of A to obtain doubly SBD forms can occur simultaneously with partitioning. Performing the local SpMV multiplication using the Compressed BICRS data structure [YR13] with nonzeroes in a row-major ordering, and by further reducing 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), we obtain the MulticoreBSP for C implementation in Algorithm 4. Note the use of the new `bsp_hpsend` function as an additional improvement. We also provide a four-superstep non-`hp` BSP implementation.

Algorithm 4 The BSP 2D cache-oblivious $z = Ax$ SpMV multiplication

```
#include <mcbsp.hpp> //The header file for the C++ wrapper (Sec. 2.8)

struct fanQuadlet {
        unsigned long int remoteP, remoteI; //remote location
        unsigned long int localI,  length;  //local range
};

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

        //fanOut; the tag contains 2 unsigned long ints: remoteStart and length
        for( unsigned long int i = 0; i < fanOut.size(); ++i ) {
                //fanOut is a vector containing fanQuadlet structs
                bsp_hpsend( fanOut[i].remoteP, &( fanOut[i].remoteI ),
                    z + fanOut[i].localI, fanOut[i].length * sizeof(double) );
        }

        //sync to ensure fanOut is done
        bsp_sync();

        //collect remote contributions
        unsigned long int *msg_tag;
        double *msg_payload;
        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 ];
}
```

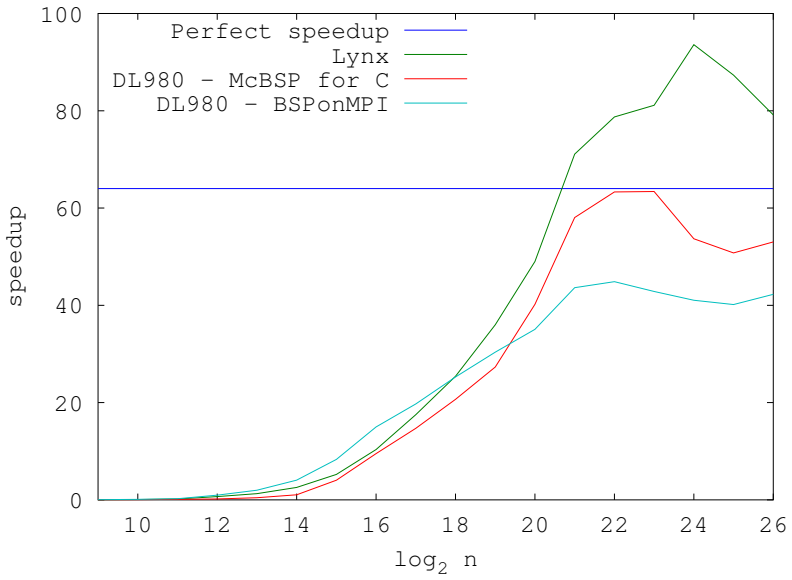
4. Experiments

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

TABLE 1. Speedups of the BSP FFT on a vector of length $n = 2^{26}$

Machine	$p = 2$	$p = 4$	$p = 8$	$p = 16$	$p = 32$	$p = 64$
Lynx (BSPonMPI)	2.0	4.3	9.2	21.9	38.4	79.2
DL980 (Compact)	1.8	3.6	5.4	11.2	23.7	53.0
DL980 (Scattered)	1.8	3.9	9.0	18.8	37.0	53.0

FIGURE 1. Speedups of the BSP FFT for $p = 64$ and varying n



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.

All software used in the experiments is freely available. The 1D and 2D PThreads SpMV multiplication codes are included with the Sparse Library[6], and the updated BSP FFT and the BSP 2D SpMV multiplication codes are available stand-alone[7]. The MulticoreBSP for C library and BSPonMPI library are freely available as well[8].

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 that calculates $U_n R_n x$ in-place, using the unoptimised UFFT kernel from BSPedupack. Table 1 shows the speedups on Lynx (using BSPonMPI and MVAPICH 2 as communication back-end), and on the shared-memory nodes (using MulticoreBSP for C) for $n = 2^{26}$. 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, and includes results using BSPonMPI on both Lynx and the DL980. Note that there is no difference between scattered and compact affinities for this combination of p and architecture.

Lynx demonstrates a super-linear speedup each time the number of processes doubles (Table 1), except when going from 16 to 32 nodes, when communication needs to involve the second Infiniband switch. The speedup from 32 to 64 processes is also more modest, since a single node

[6]see `http://albert-jan.yzelman.net/software/#SL`.
[7]see `http://albert-jan.yzelman.net/software/#HPBSP`.
[8]via `http://www.multicorebsp.com` and `bsponmpi.sourceforge.net`.

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 problem
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
wikipedia-2007	3 566 907	3 566 907	45 030 389	Link matrix

then contains two BSP processes. With compact affinity, the relative speedups of the DL980 are superlinear as p doubles, except between $p = 4$ and 8; there, sharing the same memory hierarchy begins to slow down execution times. For $p \geq 16$, processes start to use other sockets, thus communicating through the slower inter-socket interconnect but retaining the relative local efficiency of the computation phases, and making use of the additionally available caches. Scattered affinity behaves similarly with superlinear speedups for $p \leq 8$, and sublinear speedup for $p = 32$ to 64; this coincides with the slowdown for the compact affinity when doubling the number of processes per socket from 4 to 8.

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 best. A drop in superlinear speedup occurs for Lynx from 2^{25} on, precisely when the 1GB[9] of global data no longer fits into the 64 processor-local 12MB L3 caches. For similar reasons, we observe a slowdown on the DL980 for $n \geq 2^{23}$; the 256MB of local data exceeds the combined size of the eight 30MB L3 caches. Although not observed for $p = 64$, small superlinear speedups occur for $p \leq 32$ for input data that fit into local caches.

4.2. **Sparse matrix–vector multiplication.** We use a subset of large matrices from Yzelman and Roose [YR13] to test the performance of the BSP 2D SpMV multiplication on the DL580 and DL980. Table 2 shows the sparse matrices we use, which are processed using Mondriaan [VB05] version 3.11 with $\epsilon = 0.03$ and SBD reordering enabled. We compare against the Cilk and PThreads 1D SpMV multiplication methods discussed in Section 3.2, and against a PThreads implementation of Algorithm 4.

We take the average execution time of 1000 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 against that of sequential SpMV multiplications using the industry-standard Compressed Row Storage (CRS) scheme. Table 3 reports the resulting speedups. To maximise bandwidth use, we employ a scattered affinity in all experiments.

The results show significant speedups for the 2D methods. On the highly-NUMA DL980 machine, the speedups stall or deteriorate compared to those obtained on the DL580, while speedups of the 2D methods continue to increase. The *cage15* matrix is a notable exception; this matrix is known to be hard to partition [YR13]. Note that the non-`hp` BSP code usually outperforms the 2D PThreads implementation. The major implementation difference is in the fan-in step. In the BSP implementation, processes 'put' non-local contributions into remote BSMP queues in one superstep, and add remote contributions locally in the following superstep. 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 through better cache use, as data accesses have a greater temporal and spatial locality.

[9] 2^{25} complex values take 16 bytes each. Spiral requires a buffer of equal length, resulting in 2^{30} bytes of storage. This exactly equals one gigabyte.

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. The number of processes for which the reported maxima are attained, appears in parenthesis.

DL580	Cilk	PThr. 1D	PThr. 2D	BSP 2D	BSP 2D (`hp`)
Freescale1	12.2 (30)	16.7 (40)	16.3 (32)	17.3 (40)	18.3 (40)
ldoor	24.3 (40)	18.5 (40)	15.9 (40)	16.1 (40)	16.2 (40)
cage15	12.9 (40)	14.6 (40)	12.6 (40)	13.0 (40)	13.8 (40)
adaptive	28.7 (40)	13.8 (40)	19.5 (30)	22.3 (40)	23.3 (40)
road_usa	26.0 (40)	14.5 (40)	23.0 (32)	25.3 (40)	25.9 (40)
wikipedia-2007	22.8 (40)	22.1 (40)	22.0 (40)	19.2 (40)	22.8 (40)
DL980					
Freescale1	16.4 (64)	15.8 (64)	14.6 (56)	20.7 (64)	22.7 (64)
ldoor	20.2 (64)	15.3 (64)	15.2 (64)	17.6 (64)	18.8 (64)
cage15	17.7 (64)	16.1 (64)	9.3 (56)	13.8 (56)	13.5 (56)
adaptive	18.0 (64)	19.2 (64)	23.3 (40)	34.5 (64)	36.3 (64)
road_usa	15.1 (64)	19.2 (64)	31.2 (56)	43.4 (64)	46.6 (64)
wikipedia-20070206	21.6 (64)	27.7 (64)	25.1 (64)	23.5 (64)	29.7 (64)

Yzelman and Roose already noted that the 2D performance increases with the NUMA complexity of architectures [YR13], but the performance became on-par only with this new BSP implementation. The cause was the method of synchronisation; for improved scalability, MulticoreBSP for C uses spin-locks since version 1.1, instead of mutex-based synchronisation.

5. Conclusion

We presented an update to the BSPlib standard, which includes two new high-performance primitives. This interface has been implemented in MulticoreBSP 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 on a distributed-memory cluster. Differences were explained by comparing the compact and scattered affinities on a highly-NUMA architecture. We demonstrated the use of the new high-performance algorithms by extending a state-of-the-art 2D sparse matrix–vector multiplication algorithm. The 2D BSP 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 investigate two problems here, the ratio of flops per data element for the FFT and 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 to use optimised sequential kernels only. A comparison with techniques for multi-threaded FFT [FPV+09] will 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 known low-level optimisations for the SpMV multiplication [BWOD11] should be integrated into the 2D methods.

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 [Val11] is a next logical step.

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

References

[BFF+09] Aydin Buluç, Jeremy T. Fineman, Matteo Frigo, John R. Gilbert, and Charles E. Leiserson, 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, USA, 2009, ISBN 978-1-60558-606-9.

[Bis04] Rob H. Bisseling, *Parallel Scientific Computation: A Structured Approach using BSP and MPI*, Oxford University Press, Oxford, UK, 2004, ISBN 0-19-852939-2.

[BJOR03] O. Bonorden, B. Juurlink, I. von Otte, and I. Rieping, The Paderborn University BSP (PUB) library, Parallel Computing, volume 29 (2); pp. 187–207, 2003.

[BWOD11] A. Buluç, S. Williams, L. Oliker, and J. Demmel, Reduced-Bandwidth Multithreaded Algorithms for Sparse Matrix-Vector Multiplication, in *International Parallel & Distributed Processing Symposium (IPDPS)*, pp. 721–733, IEEE Press, 2011.

[FPV+09] Franz Franchetti, Markus Püschel, Yevgen Voronenko, Srinivas Chellappa, and José M. F. Moura, Discrete Fourier Transform on Multicore, IEEE Signal Processing Magazine, special issue on "Signal Processing on Platforms with Multiple Cores", volume 26 (6); pp. 90–102, 2009.

[Fri99] Matteo Frigo, 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, USA, 1999, ISBN 1-58113-094-5.

[HMS+98] Jonathan M. D. Hill, Bill McColl, Dan C. Stefanescu, Mark W. Goudreau, Kevin Lang, Satish B. Rao, Torsten Suel, Thanasis Tsantilas, and Rob H. Bisseling, BSPlib: The BSP Programming Library, Parallel Computing, volume 24 (14); pp. 1947–1980, 1998.

[IEE08] IEEE, *Std. 1003.1-2008 Portable Operating System Interface (POSIX) Base Specifications, Issue 7*, IEEE Standards for Information Technology, IEEE Press, 2008, ISBN 978-0-7381-5799-3.

[PFV11] Markus Püschel, Franz Franchetti, and Yevgen Voronenko, *Encyclopedia of Parallel Computing*, chapter Spiral, Springer, 2011.

[Sui] W. Suijlen, BSPonMPI, `http://sourceforge.net/projects/bsponmpi/`, accessed on the 10th of February, 2013.

[Val90] Leslie G. Valiant, A bridging model for parallel computation, Commun. ACM, volume 33 (8); pp. 103–111, 1990, ISSN 0001-0782.

[Val11] Leslie G Valiant, A bridging model for multi-core computing, Journal of Computer and System Sciences, volume 77 (1); pp. 154–166, 2011.

[VB05] B. Vastenhouw and R. H. Bisseling, A two-dimensional data distribution method for parallel sparse matrix-vector multiplication, SIAM Rev., volume 47 (1); pp. 67–95, 2005.

[YB11] A. N. Yzelman and Rob H. Bisseling, Two-dimensional cache-oblivious sparse matrix–vector multiplication, Parallel Computing, volume 37 (12); pp. 806 – 819, 2011, ISSN 0167-8191.

[YB12] A. N. Yzelman and Rob H. Bisseling, An object-oriented bulk synchronous parallel library for multicore programming, Concurrency and Computation: Practice and Experience, volume 24 (5); pp. 533–553, 2012, ISSN 1532-0634.

[YR13] A. N. Yzelman and D. Roose, High-level strategies for parallel shared-memory sparse matrix–vector multiplication, IEEE Transactions on Parallel and Distributed Systems, 2013, in press.

(A. N. Yzelman) Flanders ExaScience Lab (Intel Labs Europe) and Department of Computer Science, KU Leuven, Celestijnenlaan 200A - bus 2402, 3001 Heverlee, Belgium, Tel.: +32 163 275 38, Fax: +32 163 279 96

E-mail address: `albert-jan.yzelman@cs.kuleuven.be`

URL: `http://people.cs.kuleuven.be/~albert-jan.yzelman/`

(R. H. Bisseling) Department of Mathematics, Utrecht University, the Netherlands

(D. Roose and K. Meerbergen) Department of Computer Science, KU Leuven, Belgium