CONCURRENCY AND COMPUTATION: PRACTICE AND EXPERIENCE
Concurrency Computat.: Pract. Exper. 2012; **24**:533–553
Published online 2 October 2011 in Wiley Online Library (wileyonlinelibrary.com). DOI: 10.1002/cpe.1843

SPECIAL ISSUE PAPER

An object-oriented bulk synchronous parallel library for multicore programming

A. N. Yzelman*,† and Rob H. Bisseling

Mathematical Institute, Utrecht University, P.O. Box 80010, 3508 TA Utrecht, The Netherlands

SUMMARY

We show that the bulk synchronous parallel (BSP) model, originally designed for distributed-memory systems, is also applicable for shared-memory multicore systems and, furthermore, that BSP libraries are useful in scientific computing on these systems. A proof-of-concept MulticoreBSP library has been implemented in Java, and is used to show that BSP algorithms can attain proper speedups on multicore architectures. This library is based on the BSPlib implementation, adapted to an object-oriented setting. In comparison, the number of function primitives is reduced, while the overall design simplicity is improved. We detail applying the BSP model and library on the sparse matrix–vector (SpMV) multiplication problem, and show by performing numerical experiments that the resulting BSP SpMV algorithm attains speedups, in one case reaching a speedup of 3.5 for 4 threads. Whereas not described in detail in this paper, algorithms for the fast Fourier transform and the dense LU decomposition are also investigated; in one case, attaining superlinear speedups of 5 for 4 threads. The predictability of BSP algorithms in the case of the SpMV is also investigated.

Received 15 February 2011; Revised 20 June 2011; Accepted 28 July 2011

KEY WORDS: bulk synchronous parallel; BSP; parallel computing; sparse matrix–vector multiplication; multicore; shared memory; fast Fourier transform; dense LU decomposition

1. INTRODUCTION

Because multicore processors are now in widespread use, writing parallel programs is becoming highly relevant to a previously sequential world. For the parallel world, interest in shared-memory architectures as opposed to classical distributed-memory architectures, is rekindled. A parallel programming model, which is simple in design yet able to cater to both memory models, can be of key importance.

For multicore systems, sharing memory is not for free. Usually, the main memory is accessible only via a cache hierarchy, which may or may not be shared among cores; threads may thus disrupt each other and slow down computation. An extension to the bulk synchronous parallel (BSP) model to take these caches explicitly into account has been proposed by Valiant [1]. Our BSP implementation, however, still follows the original BSP model [2]. This original model as well as BSP communication libraries such as BSPlib [3] and Paderborn University BSP [4] were introduced to enable portable parallel programming, while attaining predictable performance. BSP being meant as a single program, multiple data model, a natural question is if it also performs well in a shared-memory setting. To this end, the *MulticoreBSP* proof-of-concept library has been implemented. This communications library is an object-oriented adaptation of BSPlib, written in Java, and targeting only shared-memory systems.

*Correspondence to: A. N. Yzelman, Mathematical Institute, Utrecht University, P.O. Box 80010, 3508 TA Utrecht, The Netherlands.
†E-mail: A.N.Yzelman@uu.nl

The performance of this library is examined by re-implementing the educational BSPedupack software package [5]. This package contains parallel programs originally written in BSPlib for the following applications:

- Vector inner-product calculation,
- Dense LU decomposition,
- Fast Fourier transform (FFT), and
- Sparse matrix–vector (SpMV) multiplication.

We examine the performance of the MulticoreBSP library using these same applications. As a different system architecture is targeted, however, algorithms can differ to some extent from those originally presented in BSPedupack [5]. Experimental results are given for all these applications, whereas the model and library are introduced by describing the inner-product calculation and SpMV multiplication only. The library and all applications are both freely available via http://www.multicorebsp.com.

The choice of Java as our programming language underscores the portability goal of BSP. As an interpreted language, the library and applications built on it can be distributed to every machine for which a Java interpreter is available, without the need for recompiling. Furthermore, using Java means that the BSP model is implemented in an object-oriented way, making the model easier to use while also reducing the number of BSP primitives from the already low number of 20 primitives. This is important because it enables easy learning, and it helps with keeping the (idealised) parallel machine transparent.

Restricting the number of primitives also restricts the programmer, but this is preferable to simplicity without restriction. With OpenMP [6, 7],[‡] as an example, parallelism can be introduced in code by adding a parallel directive just before a for-loop. This is simple and nonrestrictive but relies on programmer expertise to decide on the scope of variables; care must be taken to avoid race conditions and unintended sharing of variables. In contrast, the MulticoreBSP library adds robustness to algorithms by implicitly assuming that all variables are local unless explicitly defined otherwise, while retaining programming simplicity. Also, by structuring computations in independent, sequential phases, separated by synchronisation barriers, another common trap in parallel programming is avoided: deadlocks. Usually, these errors are subtle and may occur only rarely, making them hard to detect. In BSP, such deadlocks can only occur as a sequential error, not by misuse of communication primitives. In summary, we believe that the BSP model has the following desirable properties. It:

- is easy to learn by programmers and very transparent to them,
- models both distributed-memory and shared-memory architectures,
- predicts performance, and
- is robust with respect to common parallel programming pitfalls.

This article has two aims: first, to introduce the MulticoreBSP communications library, and second, to demonstrate how to design shared-memory algorithms in BSP. In particular, we show that it is possible to exploit differences between distributed-memory and shared-memory architectures on the level of implementation. To this end, we introduce MulticoreBSP by implementing a shared-memory version of the SpMV multiplication, and compare it with the original distributed-memory version from BSPedupack. Similar efforts have been made for dense LU decomposition and the FFT.

Our algorithm listings are close to the actual Java code, to demonstrate briefly but properly how the library operates in practice. Within our listings, all classes, member functions and variables defined in the library are printed in *italic*, whereas class and function names not defined therein are printed in `typewriter` font. Function code in listings is printed in plain roman, with the only further exception of reserved words, such as **for** and **return**, which are printed **boldface**.

The remainder of the article is structured as follows. After a short recapitulation of the original BSP model, Section 2 introduces the MulticoreBSP library. Using this proof-of-concept Java library, Section 3 details the BSP SpMV algorithm for multicore systems, which is put to the

[‡]See also http://www.openmp.org.

DOI: 10.1002/cpe

test in Section 4. This section also reports on performance of the dense LU and FFT algorithms. Conclusions and future work follow in Section 5.

1.1. Bulk synchronous parallel model

The original BSP model [2] targets distributed-memory systems and models them using four parameters: p, g, l, and r. The total number of parallel computing units (cores) of a BSP machine is given by p, whereas r measures the speed of each such core, in flops per second. Each core executes the same given program, usually working on different data; that is, BSP is a single program, multiple data model. The parallel program is broken down in *supersteps*, which are separated by *synchronisation barriers*. During a superstep, a process can execute any instruction, but it cannot communicate with other processes. It can, however, queue communication requests. When a process encounters a synchronisation barrier, execution halts until all processes encounter this barrier, upon which the next superstep is started concurrently. At synchronisation, all queued communication requests from the previous superstep are processed; the only communication thus occurs as part of synchronisation. The communication costs occurring this way are modelled in BSP by l, which models the time required to get past a synchronisation barrier, and g, which models the *communication gap*, the time gap between two data words sent by a process during communication. Thus, g corresponds to the inverse of the bandwidth, the speed at which data is communicated.

Suppose there are t supersteps, where each superstep i contains $w^i(s)$ flops of work for process s, and where each process receives $V_R^i(s)$ or sends $V_S^i(s)$ bytes of communication. Denote the maximum number of bytes communicated by $V^i(s) = \max\{V_R^i(s), V_S^i(s)\}$. Then, the total running time (in seconds) on a BSP machine can be predicted as:

$$T = \sum_{i=0}^{t-1} \left(\frac{1}{r} \cdot \max_s w^i(s) + g \cdot \max_s V^i(s) + l \right). \tag{1}$$

This is the *BSP cost model*, as applicable to distributed-memory parallel computing, and it enables performance prediction of BSP algorithms.

2. OBJECT-ORIENTED BULK SYNCHRONOUS PARALLEL LIBRARY

When communication occurs with object-oriented parallel programming on distributed-memory systems, entire *objects* need to be transferred. This is done by *marshalling* these objects in transferable code, a complex and time-consuming process: other objects that the object to transfer may depend on have to be identified and recursively marshalled as well. Whereas this is less an issue on shared-memory systems, as in principle all objects are stored in shared memory, MulticoreBSP demands that each communicable object can be *cloned* in memory; it must be possible for the library to copy an object and all its dependencies so that the resulting clone is completely independent of the original variable and its dependencies. Note that cloning differs from marshalling in that no transferable code is required.

Object-oriented BSP models have been researched before, for example, by Lecomber [8], targeting distributed-memory systems. Another parallel model is the coarse grained model, by Dehne *et al*. [9]; its library CGMlib [10], by Chan and Dehne, also is object-oriented and targets distributed-memory systems by using an underlying communications library, such as the Message Passing Interface. Yelick *et al*. introduced a Java dialect specifically for high performance computing and parallelism [11], using global memory space extended with advanced memory management based on programmer input. An object-oriented parallel model supporting both shared and distributed memory has been presented by Kaminsky [12]. Similarly, there also exists a BSP library for the Python programming language [13], usable on shared-memory multicore systems as well as on distributed-memory systems (by using a standard BSPlib implementation). A BSP model developed specifically for shared-memory systems has previously been researched by Tiskin [14], who introduced the BSP random access machine. This model assumes that each parallel process can access

DOI: 10.1002/cpe

two types of memory: local memory, on which it can execute local computations, and a shared memory accessible by all processes, facilitating inter-process communication. This model is more akin to OpenMP than to the pure BSP model used in this paper. More recently, Hou *et al*. introduced the Bulk Synchronous GPU Programming language [15], which enables BSP programming for GPUs.

To introduce our object-oriented BSP library for multicore systems, we first recall some basic terminology from object-oriented languages. Programming code is grouped into classes, each class having its own (local) member variables as well as member functions. Multiple *instances* of a class can be created; much like there can be multiple data elements of the same type (e.g., integer). An instance is created by calling the *constructor* function of its class. Such a constructor may require specific parameters, so that an instance cannot be created without supplying those parameters. As an example, in Java, a new string can be created by writing 'String str = new String();'. A class may be extended from a given superclass, meaning that it inherits all the members, including their visibility (see next), from its superclass. Such a subclass normally also defines new member variables or functions. Member variables or functions have one of the following *visibilities*:

- *Private*, visible only to instances of the same class, but excluding subclasses;
- *Protected*, visible only to instances of the same class, including subclasses;
- *Public*, visible to all instances regardless of class.

Member functions may be defined, yet not implemented; in such a case, the function is called *purely virtual*, and its corresponding class *abstract*. A normal virtual member function is already implemented, but can be redefined in subclasses.

A generic BSP program is defined to be a class, *BSP_PROGRAM*, having at least the functions in Table I defined. Any specific parallel algorithm is extended from this superclass, and must implement the following two protected, purely virtual, functions:

- *main_part()*: this code is only executed by a single process, and from the protected functions in Table I, it can only call *bsp_begin(int)*,
- *parallel_part()*: this is the code run in parallel; once this function is reached, all BSP functions can be called, with the exception of *bsp_begin()*.

The sequential phase can be used to prepare data or determine algorithm parameters needed for the parallel phase. From within the sequential code, a call to *bsp_begin(int)* starts the parallel phase using the given number of processes. Each such process executes the exact same code, defined in the *parallel_part()* function. The number of concurrent processes can be queried by calling *bsp_nprocs()*, and a unique process ID can be retrieved by using *bsp_pid()*. Parallel execution can be terminated prematurely by calling *bsp_abort()*. The function *bsp_sync()* is a barrier that synchronises all processes; a process will halt execution when encountering this function, only to continue when all processes were halted.

Defining only this BSP program class, a simple parallel application can already be written; see the Hello World example, Algorithm 1. Because a program is in essence a class, to run it, an instance

Table I. BSP_PROGRAM function list.

Purely virtual:	
main_part()	Will contain sequential code,
parallel_part()	Will contain parallel code.
Protected:	
bsp_begin(int)	Starts the parallel execution,
bsp_nprocs()	Returns the number of threads executing this algorithm,
bsp_pid()	Returns the current, unique thread identification number,
bsp_sync()	Synchronises all threads,
bsp_abort()	Aborts the parallel execution.
Public:	
start()	Starts the program.

DOI: 10.1002/cpe

Algorithm 1 Hello World example using the MulticoreBSP library

```
Class Hello_World extends BSP_PROGRAM {
protected function main_part():
 1: bsp_begin(4);
protected function parallel_part():
 1: print "Thread "+bsp_pid()+" says: Hello World!";
}
```

Start using:

```
 1: (new Hello_World()).start();
```

of it has to be created first: 'Hello_World myInstance = new Hello_World();'. Creating an instance of a parallel job does not implicitly start it. To do that, the *start()* function is defined, and the job is executed by 'myInstance.*start()*;'. A short-hand way of doing this is shown in Algorithm 1. After starting, the program will print the following four lines, *in an undefined order*:

```
Thread 0 says: Hello World!
Thread 1 says: Hello World!
Thread 2 says: Hello World!
Thread 3 says: Hello World!
```

Every thread has its own instance of its parallel class, such that no two threads share the same variables. For most parallel jobs, this is insufficient because data communication should be allowed to occur between threads, implying that inter-thread data transfer between variables must be possible. In this BSP framework, such variables are called *shared*, but a thread can, at all times, only see its own local value of the variable. Actual communication is only possible through explicit BSP functions, which will be defined shortly. It is essential to note that as such, these BSP-style shared variables are not at all shared in the classical sense: no two threads can read or write to the same memory location at the same time, and a BSP-style shared variable maintains a local instance for each active thread.

To facilitate this, we define an abstract BSP class *BSP_COMM*. Any shareable object type will have to implement (or extend) this class; in other words, all shared variables have *BSP_COMM* as superclass. No other objects can be communicated. Because a shared variable has no meaning if not connected to a parallel program, the constructor of every shared variable must take a *BSP_PROGRAM* as parameter, thus linking the shared variable and the parallel program it is used in. Communication always entails that some source data are transferred from a source process to a destination variable at a destination process. The MulticoreBSP library facilitates three communication methods to do this: one being to *put* data in the memory local to another thread, another being a method to *get* data from variables local to another thread, and, finally, a method to *send* messages to a variable local to another thread. In all cases, the destination variable must be shared, because by definition, any non-shared variable is not visible to other threads. These *bsp_put(...)*, *bsp_get(...)*, and *bsp_send(...)* functions are therefore defined as public functions within *BSP_COMM*; that is, as functions of destination variables. Thus, if 'destination' is a shared variable, or rather, an instance of a class derived from *BSP_COMM*, then the following functions are defined:

- destination.*bsp_put*(source, destination_thread),
- destination.*bsp_get*(source, source_thread),
- destination.*bsp_send*(source, destination_thread).

In all cases, the source parameter is optional, by default referring to the destination variable at the current process. The source and destination threads are designated by referring to their thread IDs: a thread can use the get-directive to extract data from memory local to another thread, whereas the put-directive allows a thread to push its data into memory local to another thread. All communication

DOI: 10.1002/cpe

happens at synchronisation time; multiple puts or gets to the same location at the same process result in only one of them coming through at the destination. This is in contrast to the message-passing *bsp_send*, where messages are queued at the destination variable. Reading this queue is facilitated by *bsp_qsize*, which returns the number of messages still in queue, and *bsp_move*, which moves an item from the queue into its variable. If a queue is not empty when a synchronisation barrier is encountered, all remaining entries are evicted, and hence are not available in the next superstep. A final function, *bsp_unregister*, is used to invalidate a shared variable, freeing up all memory it uses at all threads; this should never be called inside a superstep where the variable is still used.

The functions introduced so far already exist for distributed-memory systems, for example, in BSPlib [3]. MulticoreBSP also introduces a new communication directive, *bsp_direct_get(...)*, so that it can take full advantage of the shared-memory architecture it is designed to work with. The syntax is the same as that for *bsp_get*, but semantics differ in that the communication is not queued: instead, it is executed immediately, within a superstep. This is a major change from the BSP paradigm, but an acceptable one because of the similarity to the plain get-directive and a way to exploit this new directive systematically, as will be described shortly. Also, BSPlib [3], in fact, contained similar primitives: the so-called high-performance versions of the get-primitives and put-primitives, which could initiate communication inside supersteps as well. These function calls did not wait for the communication to finish, however.

A standard distributed-memory BSP algorithm can systematically be adapted to exploit shared-memory systems using this new primitive. When a superstep only communicates via get primitives while the data already is available at the remote process, these get primitives can be substituted for direct-gets, and the next synchronisation barrier can be removed; the direct-get functionality, thus, can reduce the number of supersteps of a classical BSP algorithm on shared-memory architectures. Summarizing, *BSP_COMM* defines the purely virtual functions used for communication shown in Table II.

Note that according to the BSP model, all communication is guaranteed to have been done after a synchronisation barrier is passed. Differences with the original BSPlib are that the bsp_init(), bsp_push_reg(), bsp_pop_reg(), and bsp_end() primitives no longer appear; bsp_end() is in effect assumed after the function *parallel_part* finishes, initialisation is moved to the program constructor, variable registration is moved to the variable constructor, and the bsp_pop_reg is replaced with the *bsp_unregister* function. Regarding the BSP cost model, a difference is caused by the *bsp_direct_get* primitive, which is executed during a superstep. In determining the total communication cost, we assume that the cost of a direct get is bounded by the cost of a normal get performed in synchronising. This way, when determining the total running time, we can again take into account the maximum communication volume; the original BSP cost model shown in Equation (1) is retained.

2.1. Inner-product calculation

Before proceeding with describing the implementation of the SpMV multiplication in MulticoreBSP, a simple description of a parallel inner-product calculation algorithm is given so that several standard communication variables can be introduced. Within the library, a generic shared variable is implemented as *BSP_REGISTER<T>*, which stores a variable of type *T*, accessible

Table II. BSP_COMM virtual function list.

Public:	
bsp_put(source, destination_pid)	Puts source to this variable at destination_pid,
bsp_get(source, source_pid)	Gets source from source_pid and stores it here,
bsp_direct_get(source, source_pid)	Like *bsp_get*, but does not wait for sync,
bsp_send(source, destination_pid)	Sends source to the queue of destination ID,
bsp_qsize()	Gets the number of messages still in queue,
bsp_move()	Moves an item from the queue to this variable,
bsp_unregister()	Frees the local memory used by this variable.

DOI: 10.1002/cpe

Table III. BSP_COMM_ARR virtual function list, excluding those functions already defined by BSP_COMM.

Public:	
bsp_put	*(source, s_offset, destination_pid, d_offset, length)*
bsp_get	*(source, source_pid, s_offset, d_offset, length)*
bsp_direct_get	*(source, source_pid, s_offset, d_offset, length)*

using read and write methods. It extends *BSP_COMM*, and, as discussed earlier, T must support cloning. This class alone, however, although versatile, is not yet enough to attain efficient code.

Given two real vectors $x, y \in \mathbb{R}^n$, we are interested in the parallel calculation of the inner product $(x, y) = \sum_{i=0}^{n-1} x_i y_i$. Assuming p threads are available, an intuitive approach is to divide the vectors x and y into p blocks of (roughly) equal size, where each thread calculates the sum of the corresponding block. This is followed by communicating the partial results and summing them to obtain the final result. Assuming that this inner-product calculation is part of a larger parallel scheme, the vectors x and y already have been distributed in memory, and the only remaining task is to compute partial inner products and communicate the resulting partial sums. Each thread can define a shared array of variables storing double values, to which partial sums can be communicated. Such an array could be defined using a *BSP_REGISTER* and standard Java array constructs, as in 'ArrayList<*BSP_REGISTER*<Double>> sums;'. Afterwards, putting a value x at index i at process s would be done by 'sums.get(i).*bsp_put*(x, s);'. This has several disadvantages:

- Low performance, a single entry must be accessed through multiple classes, and each register added to the array must be constructed separately;
- Verbosity, a syntax-heavy approach to control a single element of the array;
- Memory overhead, each vector element is registered separately.

Even worse is the syntax and overhead needed to put or get ranges of vectors, as, for example, required by the dense LU algorithm.

To improve this, we introduce the *BSP_COMM_ARR* subclass of *BSP_COMM*. The functions that *BSP_COMM_ARR* defines are in addition to those in Table II, and are shown in Table III. The put, get, and direct_get functions have been modified to copy a number of *length* elements of the array in a single communication request. If this number of elements is smaller than the array length, the use of *offsets* in either the source or destination array is convenient. For example, if x and y are arrays implementing the *BSP_COMM_ARR* class, the code 'y.*bsp_put*(x, i, s, j, l);' would replace y at process s with

$$(y_0, \ldots, y_{j-1}, x_i, \ldots, x_{i+l-1}, y_{j+l}, \ldots).$$

The (direct) get function works in the expected similar fashion. The main implementation of *BSP_COMM_ARR* is *BSP_ARRAY*<T>, which makes available an array with elements of type T at each process. The type T must still support cloning. For efficiency,[§] when T should be an `int` or `double` primitive type, we define the specialised *BSP_INT_ARRAY* and *BSP_DOUBLE_ARRAY* classes. Both of the specialised variants also define the *getData()* method, giving the programmer access to the underlying raw array. This reference is guaranteed constant with respect to the array object, from construction up until de-registration or end of the parallel phase. Note that any variable implementing *BSP_COMM_ARR* actually stores $p \cdot n$ values, where p is the number of threads used and n the array size; just as a regular shared variable implementing *BSP_COMM* keeps track of p local instances. Algorithm 2 shows a MulticoreBSP inner-product algorithm, using the constructs introduced earlier.

[§]In Java, 'Double' differs from '`double`', the first being an object wrapping a raw double type. Accessing such a class incurs overhead, making *BSP_ARRAY*<Double> an inefficient construct.

DOI: 10.1002/cpe

Algorithm 2 Inner-product calculation for identically pre-distributed vectors x, y

Calculates the inner product $(x, y) = \sum_i x_i y_i$. Local subvectors $\tilde{x}$ and $\tilde{y}$ are assumed available, and distributed identically.

```
protected function bsp_ip() {
 1: sums = new BSP_DOUBLE_ARRAY(this, bsp_nprocs());
 2: α = 0;
 3: for i = 0 to x̃.length−1 do
 4:     α = α + x̃[i] · ỹ[i];
 5: for i = 0 to bsp_nprocs() − 1 do
 6:     sums.bsp_put(α, 0, i, bsp_pid(), 1);
 7: bsp_sync();
 8: α = 0, a =sums.getData();
 9: for i = 0 to bsp_nprocs() − 1 do
10:     α = α + a[i];
11: return α;
}
```

3. SPARSE MATRIX–VECTOR MULTIPLICATION IN MULTICOREBSP

To enable parallel computing of $y = Ax$, the matrix A and both vectors x and y must be distributed. If p processes are used, such distributions are given by the maps

$$\begin{aligned} \pi_A &: [0, m-1] \times [0, n-1] \to [0, p-1], \\ \pi_x &: [0, n-1] \to [0, p-1], \\ \pi_y &: [0, m-1] \to [0, p-1]. \end{aligned} \tag{2}$$

Partitioners such as Mondriaan [16] or Zoltan [17] preprocess sparse matrices to find distributions optimised for parallel SpMV multiplication; this paper will not deal with partitioning methods, and simply assumes the maps from Equation (2) are available. Such maps which contain in principle all necessary information, still need further preprocessing to make them directly suitable for parallel SpMV multiplication. The latest version of the Mondriaan package (version 3.11) can perform this step automatically and gives us the following maps and local matrices. For each thread $s \in [0, p-1]$, a local matrix A_s of size $m_s \times n_s$ is stored; this local matrix can be smaller than A itself because it stores only those entries $a_{ij} \in A$ for which $\pi_A(i, j) = s$, while removing empty rows and columns and renumbering nonempty rows and columns accordingly. A local input vector x^s is also stored, so that it contains the elements from x with index j for which $\pi_x(j) = s$, and similarly for y^s and π_y. Then, during local multiplication with elements from A_s, information is required at which process and index the corresponding element from the input vector x resides, and similarly for the output vector y. Thus, the following maps are preferable to those from Equation (2):

- $\pi_r^s : [0, m_s - 1] \to [0, p-1]$, so that $\pi_r^s(i)$ is the process ID for the element of the output vector corresponding to the ith row of A_s,
- $\pi_c^s : [0, n_s - 1] \to [0, p-1]$, so that $\pi_c^s(j)$ is the process ID for the element of the input vector corresponding to the jth column of A_s,
- $I_r^s : [0, m_s - 1] \to [0, \max_k m_k - 1]$, so that $I_r^s(i)$ is the index of the local output vector $y^{\pi_r^s(i)}$ corresponding to the ith row of A_s, and
- $I_c^s : [0, n_s - 1] \to [0, \max_k n_k - 1]$, so that $I_c^s(j)$ is the index of the local input vector $x^{\pi_c^s(j)}$ corresponding to the jth column of A_s.

3.1. Partially buffered multiplication

We assume that the local matrices A_s and vectors x^s, y^s as well as the appropriate mappings from the previous section are available and examine the MulticoreBSP SpMV multiplication algorithm.

DOI: 10.1002/cpe

During local multiplication, non-local elements from the input vector may be required. These values are buffered beforehand, by allocating an input vector buffer of size n_s and copying local and non-local input vector elements into this buffer. This buffering of the input vector is called the *fan-out* step [5, Chapter 4]; the only difference with the algorithm described there is that the shared-memory variant will make use of the direct-get primitive, which prevents the fan-out operation from having to be done in a separate superstep. This brings the parallel SpMV multiplication algorithm down from three supersteps to only two.

After fan-out, upon entering the SpMV multiplication kernel, each process starts traversing the nonzeroes in its local matrix A_s in a row-major order. For each encountered row, it looks up whether the corresponding output element from y is local or not. If so, products of nonzero values and elements from x are immediately added to the local y^s. If not, they are added to a local temporary variable, which, upon switching to the next row, is sent to the correct process using the *bsp_send* primitive. After local multiplication, all processes synchronise to prepare for the second and last superstep: the *fan-in*. In this step, the incoming messages are processed. From each message, the target index and the remote contribution is read, and the contribution is added to the correct element of the local output vector.

An implementation detail regards the use of the send primitive in this setting; this directive on a shared array transmits objects of the same type, which are arrays. For the SpMV multiplication, however, we require instead sending only an index and a double value, so that a partial sum for y can be added at the correct position of the local vector at the target process. To this end, we define a Pair class, storing both an integer and a double value, and create a shared value of this class; on this type of variable, message passing with *bsp_send* will perform as required. Algorithm 3 clarifies this by showing the resulting implementation.

If we introduce the notation $|\{\pi_r^s = k\}|$ as the number of elements in π_r^s equal to k, the number of flops performed by process s can be expressed as:

$$\begin{cases} 2 \cdot \mathrm{nz}(A_s) & \text{in superstep1, and} \\ \displaystyle\sum_{i=0, i\neq s}^{p-1} |\{\pi_r^i = s\}| & \text{in superstep 2.} \end{cases}$$

Here, we count each nonzero as two flops, and the number of nonzeroes of a matrix A is given by $\mathrm{nz}(A)$. The communication volume sent $V_S(s)$ or received $V_R(s)$ by process s is:

$$\begin{aligned} V_S(s) &= |\{\pi_r^s \neq s\}| + \sum_{i=0, i\neq s}^{p-1} |\{\pi_c^i = s\}|\,, \\ V_R(s) &= |\{\pi_c^s \neq s\}| + \sum_{i=0, i\neq s}^{p-1} |\{\pi_r^i = s\}|\,, \text{ and} \\ V(s) &= \max\{V_S(s), V_R(s)\}. \end{aligned} \tag{3}$$

The time taken by this algorithm according to the BSP model then is:

$$T_{\mathrm{SpMV}} = \frac{1}{r} \cdot \left(2 \cdot \max_s \mathrm{nz}(A_s) + \max_s \sum_{i=0, i\neq s}^{p-1} |\{\pi_r^i = s\}| \right) + g \cdot \max_s V(s) + l\,. \tag{4}$$

4. EXPERIMENTS

SpMV experiments have been performed using the datasets presented in Table IV. These matrices are preprocessed using the Mondriaan software package,[¶] which partitions them for parallel SpMV

[¶]Available freely at: http://www.math.uu.nl/people/bisselin/Mondriaan.

DOI: 10.1002/cpe

Algorithm 3 Parallel sparse matrix–vector multiplication

Calculates $y = Ax$ in parallel. Based on buffering of input vector elements.

Input: local $m_s \times n_s$ submatrix A_s of A for process s,
local subvector x^s of x,
local subvector y^s of y,
mappings π_r^s, π_c^s, I_r^s, I_c^s.

```
class Pair() {
    int tag; double value;
    Pair( int _t, double _v ) {
        tag = _t; value = _v;
    }
}

protected function bsp_spmv() {
```

1: $s = bsp\_pid()$;
2: $\tilde{x} =$ new BSP_DOUBLE_ARRAY(this, n_s);
3: pairs = new $BSP\_REGISTER$<Pair>(this);
4: **for** $j = 0$ **to** $n_s - 1$ **do**
5: $\quad \tilde{x}.bsp\_direct\_get(x^s,\ \pi_c^s(j),\ I_c^s(j),\ j,\ 1)$;
6: **for** $i = 0$ **to** $m_s - 1$ **do**
7: $\quad$ **if** $\pi_r^s(i) = s$ **then**
8: $\quad\quad$ **for all** $a_{ij} \neq 0$ in the ith row of A_s **do**
9: $\quad\quad\quad y^s_{I_r^s(i)} = y^s_{I_r^s(i)} + a_{ij} \cdot \tilde{x}_j$;
10: $\quad$ **else**
11: $\quad\quad \alpha = 0$;
12: $\quad\quad$ **for all** $a_{ij} \neq 0$ in the ith row of A_s **do**
13: $\quad\quad\quad \alpha = \alpha + a_{ij} \cdot \tilde{x}_j$;
14: $\quad\quad$ pairs.$bsp_send($new Pair$(I_r^s(i), \alpha), \pi_r^s(i))$;
15: $bsp\_sync()$;
16: **while** pairs.$bsp\_qsize() > 0$ **do**
17: $\quad$ pairs.$bsp\_move()$;
18: $\quad y^s_{\text{pairs.}read()\text{.tag}} = y^s_{\text{pairs.}read()\text{.tag}} +$ pairs.$read()$.value;

```
}
```

and gives local versions of the input matrix, together with the final mappings introduced in Section 3. The preprocessing time required is reported in Table IV as well. The BSP SpMV driver reads in the Mondriaan output, then performs 100 parallel SpMV multiplications as described in Algorithm 3, and reports the average time taken. This SpMV multiplication is part of a re-implementation of BSPedupack and is freely available.[‖]

Experiments are run on three different architectures, namely:

- the AMD Phenom II 945e quad-core processor,
- the Intel Core 2 Q6600 quad-core processor, and
- the Sun UltraSPARC T2 processor.

The AMD and Intel systems both run a Linux operating system, and use the 1.6.0_23 version of the Sun Java compiler and runtime environment; the Sun platform runs on the Solaris operating system and uses the 1.5.0_27 version of Sun Java. These architectures follow very different design principles. The AMD processor has an L3 cache of 6 MB, which is shared among all four cores, whereas

[‖]See: http://www.multicorebsp.com/BSPedupack.

DOI: 10.1002/cpe

Table IV. The matrices used in our experiments. The matrices are grouped into two sets by relative size, where the first set typically fits into the L2 cache, and the second does not. The last two columns show preprocessing times required for distributing the matrix over p processes, using the Mondriaan software package with the default strategy. This reordering was done on an AMD Opteron 2378.

Name	Rows	Columns	Nonzeroes	$p = 4$	$p = 64$
west0497	497	497	1721	1 s	1 s
fidap037	3565	3565	67 591	1 s	2 s
s3rmt3m3	5357	5357	106 526	1 s	5 s
memplus	17 758	17 758	126 150	1 s	8 s
cavity17	4562	4562	138 187	1 s	3 s
bcsstk17	10 974	10 974	219 812	3 s	8 s
lhr34	35 152	35 152	764 014	6 s	16 s
bcsstk32	44 609	44 609	1 029 655	15 s	31 s
nug30	52 260	379 350	1 567 800	43 s	4 min
s3dkt3m2	90 449	90 449	1 921 955	39 s	1 min
tbdlinux	112 757	21 067	2 157 675	3 min	6 min
stanford	281 903	281 903	2 312 497	5 min	8 min
stan-ber	683 446	683 446	7 583 376	13 min	21 min
cage14	1 505 785	1 505 785	27 130 349	16 min	45 min
wiki05	1 634 989	1 634 989	19 753 078	6 h	9 h
wiki06	2 983 494	2 983 494	37 269 096	19 h	23 h

each single core has its own 512-kB L2 cache and 64-kB L1 cache. This is in contrast to the Intel design: the Intel Q6600 has no L3 cache, whereas the two 4-MB L2 caches available are each shared by two cores. Each core does have a local 64-kB L1 cache. Thus, with the AMD architecture, data transfer between cores is uniform, whereas in the Intel case, by connectivity of the L2 cache, a core has a faster transfer speed with one specific other core, and the other two cores must be accessed through the main memory. As such, the Intel processor is said to be a *cache-coherent non-uniform memory access* architecture. The BSP model used in this paper does not take this into account; this would require the Multi-BSP model [1] as well as an adaptation of the Mondriaan partitioner. The BSP algorithms presented here are expected to perform better on the AMD architecture, because of the more uniform memory access.

The Sun UltraSPARC T2 architecture, being built specifically for throughput-intensive applications, follows a completely different strategy. While typically operating at lower clock rates, there are eight cores available on a single processor with each core supporting fast interleaved execution of eight concurrent threads; a single processor thus can execute 64 threads. The idea is that, on each core, when one of the threads stalls (e.g. caused by a cache miss), one of the other threads can proceed. This hides the latency of data transfer. One UltraSPARC T2 core has two calculation units, meaning two threads can work simultaneously while the other six wait, ideally for data still to arrive.

The UltraSPARC T2 has one 4-MB L2 cache, shared among all cores by means of a crossbar. Main memory is accessible through this cache and is divided over four different memory controllers. The system I/O is connected via the same crossbar. Each single core has its own 8-kB L1 data cache, but note that this cache is shared among eight threads. It also has more extensive (compared with the Intel and AMD architectures) instruction caches and translation lookaside buffers. Overall, this architecture is expected to offer the eight cores a uniform memory access time. It also should offer good scalability, especially for smaller problems.

For all BSP algorithms, we will report speedups for various problem instances, using a varying number of threads, as appropriate for the architecture experimented on. Speedups are relative to the parallel program run with $p = 1$, as comparable sequential algorithms in Java are not readily available. For the SpMV multiply and the FFT algorithm, however, the main computational kernel is, in fact, plain (but unoptimised) sequential code. Still, the reported values are optimistic speedups as some parallel overheads remain present for one thread, even if they are constant and do not scale with the problem size.

DOI: 10.1002/cpe

4.1. Benchmarking and prediction

The parameters r, g and l are estimated via a benchmarking program, described and included in BSPedupack [5]. In short, this benchmark sends messages of varying length to all processors, and measures the time it takes to send h messages and synchronise, a total of 1000 times. This measurement is conducted for $h = 1, \ldots, 128$. Afterwards, a least-squares fit is calculated to find the affine relation ($gh + l = t/1000$) between the message size h and the time t required to send 1000 messages. The speed of a single core is measured using dense vector operations. Results are shown in Table V.

The Intel architecture shows a constant flops per second rate, whereas latency increases as the number of cores p is increased. As expected because of the sharing of the L2 caches, the communication gap and the latencies are much lower for $p = 2$ than for $p > 2$, where the communication gap is constant for $p = 3$ and $p = 4$. The AMD processor shows a more variable flop rate and even shows a drop in performance for $p = 4$. When using more than one core, the latency is constant but slightly higher than that of the Intel processor. Like the Intel processor, the communication gap increases for $p > 2$ but stabilises after $p = 3$, indicating that L3 cache sharing is faster for $p = 2$ than for higher p. The Sun machine behaves as expected for smaller p: r is constant, and g, l increase only slightly as more processes are used. This indicates that processes are first allocated to a single core and spread over cores when more than eight processes are required. The large increase in g and l from $p = 8$ to $p = 16$ supports this, as using more than one core means communication has to go through the crossbar. This effect increases as more and more cores are utilised. An anomaly is the decrease in process speed for $p \geqslant 32$; there, it is possible that the number of threads is too high with respect to the data throughput, such that there is no room for more threads to work while other threads wait for data to arrive. Indeed, while benchmarking, the 4 kB of data used is quite local and does not cause long latency times. If the required data would take longer to arrive, the processor cores would have more time available to interleave other threads, thus retaining the performance speed per thread. This indicates that benchmarking with localised data for performance prediction of applications with irregular data accesses may not yield the correct value of r, especially for large p.

The benchmarks can be used to predict SpMV performance, as after benchmarking a BSP machine and partitioning an input matrix, all values in Equation (4) are known, so that the theoretical running time of the SpMV algorithm can be obtained. From these values, the theoretical speedups

Table V. BSP benchmark data for an Intel Core 2 Q6600 quadcore system, an AMD Phenom II 945e quadcore system, and the Sun UltraSPARC T2 system consisting of eight cores with eight threads each.

	Intel Q6600			AMD 945e		
p	r (Mflop/s)	g (flops)	l (flops)	r (Mflop/s)	g (flops)	l (flops)
1	593.62	76.51	666	819.20	84.9	379
2	597.99	174.52	7764	890.43	357.7	31921
3	605.35	281.22	12342	851.23	1001.1	28748
4	600.17	323.79	28947	663.91	935.1	32950

	Sun UltraSPARC T2		
p	r (Mflop/s)	g (flops)	l (flops)
1	47.08	50.19	228
2	47.98	74.78	3979
4	45.66	97.31	10579
8	44.66	119.17	24232
16	47.45	220.71	54409
32	34.43	577.39	76150
64	22.43	633.60	134631

DOI: 10.1002/cpe

can be calculated as well. Using the actual running times reported in the following subsection, predicted values can be compared with actual measurements. Dividing the measured values by the predicted values then gives the error factors; Tables VI and VII show the result of this experiment. Actual running time measurements are averaged over 100 SpMV multiplications.

Observed are errors of up to 11 times, compared with the actual measurements on the Intel platform. On the AMD platform, slightly better error factors are obtained with a maximum of 8. Because these factors are not constant, the BSP model does not make very accurate predictions on the running time of the algorithm, when applied to the SpMV multiplication. The main cause is that the BSP benchmark measures process speed according to *dense* operations; sparse operations such as the SpMV multiply are known to perform at a fraction of peak performance, as a result of inefficient cache use [18–20]. If inefficient cache use indeed is the cause, prediction should be more successful for smaller and for structured sparse matrices [18, 21]. Input and output vectors corresponding to the Stanford matrix do not fit entirely into the L2 caches of both architectures, and the matrix, representing a links within the Stanford domain from 2002, inherently is unstructured. On the other hand, the s3dkt3m2 matrix is smaller, so its vectors do fit into L2 cache, whereas additionally, the matrix comes from a FEM application on a regular grid and, thus, is structured. On all architectures, the prediction errors indeed are smaller for the s3dkt3m2 matrix, especially on the UltraSPARC T2 with large p. Other causes for discrepancy may include overhead caused by the MulticoreBSP library,

Table VI. Prediction of MulticoreBSP SpMV running time (in ms) and speedups with their error factors defined as the measured value divided by the predicted value; an error factor of 7 in timing thus means the parallel run took seven times longer than predicted, and an error factor of 1 indicates prediction matched exactly. Factors smaller than one indicate that parallel execution was faster than predicted. For speedup, this is reversed: for error factors smaller than one, speedup was less than predicted. Values are given for two test matrices. The BSP parameters r, g, l for the architectures tested are taken from Table V.

		Stanford			s3dkt3m2		
		$p=2$	$p=3$	$p=4$	$p=2$	$p=3$	$p=4$
Partitioner	Superstep 1	2 418 012	1 667 496	1 237 054	4 120 988	2 719 368	2 056 482
output	Superstep 2	225	600	607	609	603	0
	$\max_s V(s)$	225	601	608	609	1206	840
Intel	Time	4.1	3.1	2.4	7.1	5.1	3.9
prediction	Error factor	7.3	9.3	10.8	2.1	3.3	3.7
	Speedup	1.9	2.6	3.2	1.8	2.5	3.2
	Error factor	1.0	0.6	0.7	0.7	0.4	0.4
AMD	Time	2.8	2.7	2.8	4.9	4.6	4.3
prediction	Error factor	7.3	5.8	5.6	2.5	2.9	1.7
	Speedup	2.0	2.1	2.0	1.9	2.0	2.1
	Error factor	0.7	1.0	1.0	0.8	0.7	1.1
Sun	Time	50.8	–	28.6	86.9	–	47.1
prediction	Error factor	3.7	–	3.5	2.0	–	1.7
	Speedup	1.9	–	3.4	1.8	–	3.4
	Error factor	0.9	–	0.9	0.9	–	0.8

Table VII. As Table VI, but for a larger number of concurrent threads as supported by the Sun UltraSPARC T2.

		Stanford			s3dkt3m2		
		$p=16$	$p=32$	$p=64$	$p=16$	$p=32$	$p=64$
Partitioner	Superstep 1	317 822	158 942	79 484	516 096	258 048	129 024
output	Superstep 2	683	829	737	474	558	408
	$\max_s V(s)$	684	829	738	804	624	424
Sun	Time	11.1	20.8	30.4	15.8	20.2	23.7
prediction	Error factor	3.5	2.1	1.5	2.2	2.1	2.0
	Speedup	8.9	4.7	3.2	10.1	7.9	6.7
	Error factor	0.8	1.7	2.0	0.7	1.0	1.0

DOI: 10.1002/cpe

which can depend on the parallel algorithm executed and can amplify the effects of cache misses. Further factors can be hardware behaviour not taken into account by the BSP model (such as is the case with the cache-coherent non-uniform memory access Intel Q6600), operating system jitter or virtual machine jitter.

If library overheads or hardware peculiarities are the causes for bad prediction and these overheads are constant, regardless of the number of processors involved, speedup prediction should perform much better. Indeed, the speedup prediction error factors are much smaller compared with the running time error factors. It is possible to use the predicted speedup for instance to decide how many processes to allocate for a specific job, but otherwise it has limited practical applicability. Estimating speedups is more successful on the Stanford matrix and for the AMD architecture.

4.2. Sparse matrix–vector multiplication

The SpMV multiplication results for the Intel and AMD architectures are found in Table VIII, and those for the Sun platform in Table IX. For the classical cache-based architectures, that is, the Intel and the AMD processors, speedups are only attained for larger matrices. This should come as no surprise, because for smaller matrices, the input and output vectors both fit into L1 cache, which is not shared among cores. This differs for the UltraSPARC T2 architecture, where the L1 cache is much smaller and is shared among eight threads simultaneously. For this machine, speedups are obtained even for the smaller matrices. The highest speedup obtained is 9.31 for the Stanford–Berkeley matrix using $p = 32$ processes.

In general, better speedups are attained for the larger matrices, as the ratio of local work versus data movement caused by communication is more favourable there. Increasing matrix size is not automatically beneficial for scalability, however; larger vectors may no longer fit into lower-level caches, and for higher-level caches the bandwidth is more limited, thus making minimisation of inter-core communication proportionally more important. In other words, for small matrices there is not enough work to attain efficient parallelisation, whereas for larger matrices, communication costs may increase, as do the effects of inefficient cache use, especially when caches are shared.

Table VIII. Measured speedup of MulticoreBSP sparse matrix–vector multiplication on the Intel Q6600 and AMD945e architectures. The speedup is calculated as the time taken for $p = 1$ divided by the time taken for the parallel run, and these timings are taken averaged over 100 SpMV runs. The best speedup for each architecture is shown in boldface.

	Intel Q6600			AMD 945e		
Matrix	$p = 2$	$p = 3$	$p = 4$	$p = 2$	$p = 3$	$p = 4$
west0497	0.61	0.69	0.59	0.97	**1.08**	0.54
fidap037	0.27	0.25	0.19	0.41	0.26	0.22
s3rmt3m3	0.28	0.23	0.34	0.44	0.32	0.29
memplus	0.96	0.35	0.32	0.95	0.48	0.37
cavity17	0.26	0.24	0.18	0.30	0.30	0.25
bcsstk17	0.96	0.41	**1.18**	**1.20**	0.45	0.78
lhr34	**1.22**	1.08	0.53	1.17	**1.77**	0.55
bcsstk32	0.73	0.89	0.87	1.04	0.99	**1.20**
nug30	0.69	0.64	0.46	0.74	0.75	0.59
s3dkt3m2	1.20	1.09	**1.26**	1.46	1.29	**2.33**
tbdlinux	**1.20**	0.96	0.87	**1.52**	1.23	1.46
stanford	1.72	1.79	**1.98**	1.46	1.92	**1.93**
stan-ber	1.22	1.25	**1.26**	1.51	1.79	**1.82**
cage14	0.85	0.93	0.78	1.14	**1.46**	1.34
wiki05	1.97	**2.31**	2.21	2.33	2.68	**3.34**
wiki06	1.83	2.06	**2.17**	1.38	**2.17**	2.00

DOI: 10.1002/cpe

Table IX. Measured speedup of MulticoreBSP sparse matrix–vector multiplication on a Sun UltraSPARC T2 machine, obtained in the same way as for Table VIII.

Matrix	$p = 2$	$p = 4$	$p = 8$	$p = 16$	$p = 32$	$p = 64$
west0497	**1.20**	0.94	0.81	0.62	0.39	0.24
fidap037	0.58	0.54	0.64	0.50	0.68	0.37
s3rmt3m3	0.91	1.42	1.47	**1.63**	0.93	0.68
memplus	**1.50**	1.18	1.01	0.86	0.78	0.44
cavity17	0.85	0.98	1.27	**1.33**	0.72	0.60
bcsstk17	1.79	2.95	2.57	**2.96**	2.20	1.36
lhr34	1.98	2.60	**4.34**	4.33	3.68	2.19
bcsstk32	1.59	2.71	4.68	6.40	**7.02**	4.43
nug30	1.24	**1.41**	1.21	1.03	0.79	0.55
s3dkt3m2	1.65	3.56	5.55	8.50	**8.59**	6.46
tbdlinux	1.89	3.01	3.97	**4.19**	2.75	1.51
stanford	1.64	3.05	5.03	6.80	**8.02**	6.42
stan-ber	1.64	2.95	5.00	7.31	**9.31**	9.27
cage14	1.31	2.04	2.43	**2.62**	1.98	1.63
wiki05	1.86	3.40	**6.02**	5.38	2.96	1.97
wiki06	1.82	3.38	**6.12**	2.50	1.98	2.56

4.3. The fast Fourier transform

The FFT application from BSPedupack maps a complex-valued input vector of length n to its discrete Fourier transform and does this in two supersteps, if $p \leqslant \sqrt{n}$. For algorithm details, we refer to Bisseling [5, Chapter 3]; what now follows is but a brief description. The input vector is assumed to be distributed cyclically over the p processors. The first superstep of the algorithm first performs a parallel bit reversion, followed by a number of concurrent unordered sequential FFTs, and finally redistributes data and synchronises. The second and last superstep then proceeds with p concurrent unordered sequential generalised FFTs. If n is not large enough compared with the number of processors, more supersteps consisting of redistribution and concurrent unordered generalised FFTs are required. Both the input vector size and the number of processors used by the algorithm are required to be powers of two. The advantage of this algorithm is that the sequential FFTs can be provided by external libraries like FFTW [22], or can even be recursively parallelised, for example when applied within a hierarchy of parallel machines.

Table X shows scalability results of this algorithm on our shared-memory architectures. It is immediately seen that speedups on the Intel and AMD platforms are only attained for large n; this must be caused by the smaller vectors fitting into the local caches, so that communicating between cores is more expensive than doing the entire computation on a single core. The AMD 945e computation starts to attain speedups for $n \geqslant 2^{18}$, which corresponds to 4 MB of data; for $p = 1$, this is eight times the size of an L2 cache, and for $p = 4$, this is twice the size. The Intel computation starts getting speedups for $p = 2$ at 16 MB of data, and for $p = 4$ at 32 MB. Note that the Intel processor does not have an L3 cache and will communicate through main memory for $n > 2^{18}$. The AMD does have an L3 cache, but its capacity is exceeded for the same n, so it will communicate through main memory as well.

Starting at $n = 2^{22}$ (64-MB) for the AMD processor, or $n = 2^{23}$ (128 MB) for the Intel processor, speedups become superlinear. This is explained by better cache use caused by data locality: the first superstep of the parallel FFT algorithm performs n/p unordered FFTs on contiguous blocks of size p, in parallel; that is, each processor performs n/p^2 unordered FFTs of small size, and thus performs extremely data-local computations. The second superstep performs p unordered generalised FFTs of size n/p; these are less data-local and performance will drop if a vector of size n/p no longer fits into local cache. This drop is in comparison with the first superstep, however, and does not seem to hinder overall scalability.[**]

[**]See the results of $n \geqslant 2^{21}$ for the Intel architecture and $n \geqslant 2^{19}$ for the AMD; both exceed the L2 cache for the second superstep.

DOI: 10.1002/cpe

Table X. Measured speedups of MulticoreBSP FFT on two shared-memory architectures, the Intel Q6600 and the AMD 945e. The speedup is calculated as the time taken for $p = 1$ divided by the time taken for the parallel run, and these timings are taken averaged over 30 FFT runs. The input vector size varies between 8 kB (for $n = 2^9$) to 512 MB (for $n = 2^{25}$).

	Intel Q6600		AMD 945e	
$\log_2 n$	$p = 2$	$p = 4$	$p = 2$	$p = 4$
9	**1.1**	0.9	0.8	0.6
10	**1.1**	0.9	0.6	0.7
11	0.3	0.2	0.7	0.8
12	0.4	0.5	**1.2**	1.1
13	0.5	0.3	0.9	0.9
14	0.7	0.4	1.0	**1.2**
15	0.7	0.4	0.9	1.0
16	0.4	0.2	0.8	0.8
17	0.5	0.3	0.9	1.0
18	0.6	0.4	1.0	**1.2**
19	1.0	0.6	1.3	**1.5**
20	**1.5**	0.7	1.7	**2.2**
21	**1.7**	1.5	1.9	**2.7**
22	1.8	**1.9**	2.3	**3.2**
23	2.3	**2.4**	2.6	**3.6**
24	2.2	**2.2**	2.9	**4.9**
25	2.6	**3.2**	2.9	**5.0**

Table XI reports speedups attained on the Sun architecture. Experiments start at a vector size $n = 2^{13}$ because the larger number of available threads would otherwise make more than one redistribution necessary. For this n and $p = 64$, the only reported slowdown (of a factor 0.8) for the FFT on the UltraSPARC T2 occurs. Whereas for the AMD and Intel processors no speedups could be obtained for the shorter input vectors, the Sun platform starts at a speedup of 5.3, with the lowest maximum speedup being 2.9 for $n = 2^{15}$. As the problem size increases, the best speedups are attained for increasingly larger p. Superlinear speedups are attained as well, but, in contrast, only for $p \leqslant 8$ and sufficiently large n; and in all cases, a larger (but sublinear) speedup can be obtained by using a larger p.

4.4. Dense LU decomposition

The BSP LU algorithm performs an LU decomposition of an $n \times n$ dense matrix A. It is a straightforward parallelisation of the well-known LU decomposition algorithm with partial pivoting, for

Table XI. Measured speedups of MulticoreBSP FFT on the Sun UltraSPARC T2, obtained in the same way as for Table X. The input vector size varies between 128 kB (for $n = 2^{13}$) to 512 MB (for $n = 2^{25}$).

$\log_2 n$	$p = 2$	$p = 4$	$p = 8$	$p = 16$	$p = 32$	$p = 64$
13	1.9	3.5	**5.3**	4.6	2.1	0.8
14	1.4	2.4	3.9	**4.2**	1.8	1.2
15	1.6	1.6	2.8	2.4	**2.9**	1.3
16	1.4	2.0	2.6	**4.0**	2.8	1.7
17	1.3	2.2	3.1	3.4	**3.7**	2.7
18	1.4	2.0	**3.4**	3.0	3.0	2.4
19	1.8	2.6	3.6	**5.4**	3.3	3.0
20	2.2	3.2	4.7	**6.8**	5.8	4.2
21	2.5	3.8	5.8	8.6	**9.0**	6.8
22	2.7	5.2	6.4	10.8	**13.5**	12.6
23	2.9	6.2	8.3	11.1	18.5	**22.0**
24	2.9	5.9	10.5	13.6	15.4	**23.9**
25	2.9	5.9	9.8	13.0	16.4	**16.7**

DOI: 10.1002/cpe

example presented by Golub and van Loan [23, Algorithm 3.4.1]. Note that this algorithm is not based on level-3 BLAS, and that a better-performing algorithm may be obtained by using those; optimisation or parallelisation for high performance computing then typically requires hand-tuned [24] or auto-tuned [25] software. The parallel algorithm used is described in detail by Bisseling [5, Chapter 2]; again, we proceed with only a short description. We assume that the matrix is cyclically distributed in two dimensions over $p = p_y p_x$ processes; that is, the ith row of A is local to a process with $s_i = i \mod p_y$, and the jth column of A is local to a process with $t_j = j \mod p_x$. Taking for example the map $s = s_i p_x + t_j$, each combination of row and column properly corresponds to one unique process. For each stage k from 0 to $n-1$, first, a pivot element in the kth column is searched for, in parallel. When identified, the corresponding row is swapped with the kth row by using the get primitives. During this procedure, it is possible that $2p_x$ processes work and $(p_y - 2)p_x$ others idly stand by, thus negatively impacting the load balance. On the other hand, when the two swapped rows are local to the same processes, communication is local only, which may be preferable even if it causes load imbalance. These considerations do not contradict when p_y is taken to be 1; then, there is no load imbalance and there are only local computations during pivoting. When swapping is complete, all local elements of the lower-right $(n-k) \times (n-k+1)$ submatrix can be updated in parallel by caching (using direct-get primitives) the relevant ranges of the kth row and the kth column, including the pivot value. After updating, the algorithm increments k and continues with the next stage.

The LU algorithm differs from SpMV and FFT mainly in that the number of supersteps as well as the amount of communication are linearly related to n instead of being constant; the overhead of the algorithm is thus expected to be higher. Another difference is that local matrix entries are reused in the calculation, instead of only the components of input or output vectors; this makes it easier to attain good speedups. In our experiments, A is taken as a random matrix, and 30 decompositions are timed, and the average time taken is reported. In our implementation, the matrix size must be divisible by both p_y and p_x; we tested the sizes $n = 120, 600, 1200$.

For the smallest size tested, $n = 120$, no speedup was attained on the Intel and AMD architectures, whereas the UltraSPARC T2 architecture gained a maximum factor 1.4 for $p_y = 3$ and $p_x = 1$ and did not get slowdowns as long as $p_y p_x < 8$; these results are not included in tabular form. Speedups for $n = 600, 1200$ are reported in Tables XII and XIII. For the Intel processor, no speedup is attained for $n = 600$. The AMD processor performs somewhat better with a factor 1.1 for $2 \cdot 1$ processors. As expected, the results improve for yet larger n; both these architectures perform up to a factor 1.4 better, again by using $2 \cdot 1$ processors. The Sun platform performs somewhat better, reporting speedups up to 3.0 for $n = 600$ and 6.7 for $n = 1200$, using $2 \cdot 3$, respectively, $6 \cdot 2$

Table XII. Measured speedups for the MulticoreBSP LU algorithm on the Intel Q6600 and the AMD 945e processors. The algorithm has been run for square dense matrices of size 600 and 1200. The matrices are distributed over processes both row-wise and column-wise; the number of processes p_x used in the column direction is displayed horizontally in the four parts of the table, and the number of processes p_y used in the row direction is displayed vertically. The total number of processes $p = p_x p_y$ exceeds the available number of cores (4) in half the cases presented; those speedups are printed in italic. The largest speedup for each of the four parts is printed in boldface.

		Intel Q6600				AMD 945e			
600×600 :		1	2	3	4	1	2	3	4
	1	1.0	0.7	0.6	0.5	1.0	0.7	0.5	0.3
	2	0.8	0.5	*0.3*	*0.3*	**1.1**	0.4	*0.2*	*0.2*
	3	0.7	*0.4*	*0.3*	*0.2*	0.6	*0.3*	*0.2*	*0.2*
	4	0.4	*0.3*	*0.2*	*0.2*	0.4	*0.2*	*0.2*	*0.2*
1200×1200 :									
	1	1.0	1.2	1.0	0.9	1.0	1.1	0.8	0.5
	2	**1.4**	1.2	*0.9*	*0.8*	**1.4**	0.7	*0.4*	*0.4*
	3	1.2	*1.0*	*0.8*	*0.6*	1.2	*0.5*	*0.4*	*0.3*
	4	1.3	*0.9*	*0.7*	*0.6*	0.8	*0.5*	*0.3*	*0.3*

DOI: 10.1002/cpe

Table XIII. Similar to Table XII, but for the Sun UltraSPARC T2 architecture.

600×600 :		1	2	3	4	5	6	8
	1	1.0	1.8	2.3	2.6	2.8	2.2	2.1
	2	1.8	2.8	**3.0**	2.9	2.6	2.0	1.8
	3	2.4	3.0	2.9	2.4	2.2	1.6	1.4
	4	2.7	2.9	2.4	2.0	1.8	1.4	1.2
	5	2.9	2.6	2.2	1.9	1.6	1.2	1.0
	6	3.0	2.4	2.0	1.7	1.4	1.0	0.8
	8	2.9	2.2	1.6	1.3	1.0	0.8	0.7
1200×1200 :								
	1	1.0	2.2	3.3	3.9	4.5	5.0	5.4
	2	2.1	4.3	5.4	5.9	6.1	6.1	5.7
	3	3.0	5.6	6.1	6.4	6.2	5.6	5.1
	4	4.1	6.3	6.3	6.2	5.7	5.4	4.7
	5	4.9	6.6	6.4	5.9	5.3	4.9	4.1
	6	5.5	**6.7**	6.1	5.6	5.0	4.3	3.7
	8	6.3	6.2	5.4	4.6	4.1	3.6	2.8

processors. Larger n will likely yield larger speedups using more processors on this architecture. We observe that all architectures seem to prefer $p_y > p_x$, which is surprising because this causes load imbalance, as discussed earlier.

5. CONCLUSIONS

We have demonstrated that the BSP [2] programming paradigm can be efficiently used for shared-memory parallel programming. A proof-of-concept BSP library, MulticoreBSP, targeted towards shared-memory architectures, has been implemented and is freely available. It differs from prior work in that it exploits the shared-memory architecture solely through a *direct-get* method, otherwise assuming processes only have local memory and communicate only through distributed-memory primitives: the standard *get*, *put* and *send* methods. An SpMV algorithm has been described and implemented using this library, similar to the algorithm introduced in BSPedupack [5], except that the number of supersteps has been reduced by use of the direct-get. Experiments on both the predictability and scalability of this algorithm have been performed. Predictions of the running time leave much to be desired; predictions of the speedup are more accurate.

Actual speedup measurements have been carried out on three different architectures, two quad-core machines (Intel and AMD) and one highly-threaded machine (Sun). Results of the SpMV multiplication show modest overall speedup, with the higher speedups reserved for the larger matrices, showing a speedup of 3.34 for the wikipedia-2005 matrix with $p = 4$, and superlinear speedup of 2.33 for $p = 2$ on the same matrix, both on the AMD architecture. Speedups for SpMV multiplication on the Sun UltraSPARC T2 processor are very good when compared with those of the other architectures for $p \leqslant 4$. For larger p, performance increases even more, with a largest measured speedup of 9.31 for $p = 32$ threads, on the Stanford-Berkeley link matrix; surprising, because of all link matrices, this one is the hardest to partition [18,21]. On the other hand, difficulty in partitioning implies that there is no apparent data locality to be found; thus irregular data accesses are expected, which is precisely when the UltraSPARC T2 architecture performs well. For the larger test matrices, the performance drops, gaining only good speedups for small p. By the same reasoning, it appears that for these matrices there is enough data locality to cause the processor core to saturate on a lower number of threads.

Other example applications have been implemented and experimented with, as well. They are the FFT and dense LU decomposition. Both are easier to parallelise: most data elements are used several times, instead of only once as is the case with SpMV multiplication. Good speedup, even superlinear speedup, is attained for the FFT: up to 6.2 for $p = 4$ on the Sun UltraSPARC T2, 5.0

DOI: 10.1002/cpe

for $p = 4$ on the AMD 945e, and 2.6 for $p = 2$ on the Intel Q6600. The maximum speedup for the Intel is 3.2 out of 4, and that for the UltraSPARC T2 hovers around a factor 24, which still seems to increase for larger n. The LU decomposition performs less well on the AMD and Intel architectures; we think this is caused by the lack of optimisation of the algorithm. During each iteration of the algorithm, threads move along the pivot column unchecked while updating the local submatrices, which generally do not fit into local caches. This may yield much data movement for each thread in shared caches or main memory, while inside a computation superstep; this points to another large difference between distributed-memory and shared-memory BSP. Solutions lie in employing existing sequential optimisation techniques to limit unnecessary data movement, which should lead to speedups when using multiple threads, and not just in a constant factor gain. Supporting this, on the UltraSPARC T2 where data locality is less important, the speedups are much more pronounced, going up to a factor 6.7. Also for smaller problems, the UltraSPARC T2 attains a much better speedup than that of the more traditional Intel and AMD platforms. Latency hiding thus gives ample opportunity for speedups on highly unstructured problems, but it seems hard to predict the size of such speedups within the BSP model used here.

In general, we have shown that the BSP model, library and algorithms can be adapted for shared-memory architectures, and that these algorithms can work well, as tested on a variety of architectures. Similarities, but also some of the large differences have been discussed, leading to the observation that sequential optimisation is of increasing importance in shared-memory computing; gains in data movement result in less contention between memory accesses of many threads, yielding better scalability. Nevertheless, such optimisations will benefit distributed-memory computing as well. Hence, the same efficient and scalable BSP algorithms can be employed for both distributed-memory and shared-memory architectures.

5.1. Future work

While the MulticoreBSP library shows that the BSP model is valid for current shared-memory systems, the Java implementation is not viable for high-performance computing. Although it attains modest speedups and is proper for concept testing and educational use, the algorithms are seen to perform slower than expected. Construction of a similar, more competitive library in for example the C++ programming language should be worthwhile. Such a library can build forth on existing technologies such as POSIX threads and may be augmented with the Message Passing Interface, or distributed-memory BSP; combining both can result in a BSP library which is able to switch as required between shared-memory and distributed-memory implementations. Moreover, by incorporating the Multi-BSP model [1] instead of the flat BSP model, such a library can automatically distribute a BSP algorithm over a hybrid distributed-memory and shared-memory system, thereby completely eliminating the need for explicit hybrid parallel programming.

Whereas predictability here only has been investigated for the SpMV algorithm we find, like many others, that processor speed is not always properly measured in flops per second. Instead, timings based on irregular data accesses may be of more importance and may yield better predictions of BSP algorithms working on sparse problems. Such benchmarks may be integrated into the BSP benchmarking application, which would also benefit from an update to the Multi-BSP model; message size can be varied so that properties of the different levels of the memory hierarchy can be measured. Ideally, however, this also requires information on the exact memory size of each such memory level.

Regarding the SpMV multiplication specifically, interleaving the querying of the process index map π^s_r for locality of output vector elements results in a constant overhead during execution of the SpMV multiplication. An alternative is to buffer the remote elements locally, thus removing this overhead, which results in a more efficient kernel. After execution of this kernel, the local buffer is read out, and non-local elements can again be sent out using the send primitive. The resulting *fully buffered* SpMV multiplication kernel enables the use of external multiplication code, which may perform additional optimisations [18, 21, 26, 27]. To attain better scalability, further optimisation surely is required. Applying partitioning to minimise communication between computing cores is not enough, as data access patterns of the input vector are not improved while bandwidth becomes

DOI: 10.1002/cpe

more limited as more cores are involved in the computation. Future work should be directed towards combining communication minimisation with methods to enhance cache use, for example by permuting of the local input matrix representations [18, 21], by adapting the sparse matrix storage scheme [27–29] or both.

ACKNOWLEDGEMENTS

We thank the Center for Computing and Communication of RWTH Aachen University for providing access to their Sun UltraSPARC T2 cluster, and for helping in our experiments. We also extend our special thanks to Ruud van der Pas from Oracle Corporation, who offered extensive feedback on this text, as well as valuable insights on the UltraSPARC T2 architecture and its performance during the experiments described here.

REFERENCES

1. Valiant LG. A bridging model for multi-core computing. In *Algorithms - ESA 2008, Lecture Notes in Computer Science*, Vol. 5193. Springer: Berlin, 2008; 13–28.
2. Valiant LG. A bridging model for parallel computation. *Communications of the ACM* 1990; **33**(8):103–111.
3. Hill JMD, McColl B, *et al*. BSPlib: the BSP programming library. *Parallel Computing* 1998; **24**(14):1947–1980.
4. Bonorden O, Juurlink B, *et al*. The Paderborn University BSP (PUB) library. *Parallel Computing* 2003; **29**(2): 187–207. DOI: 10.1016/S0167-8191(02)00218-1.
5. Bisseling RH. *Parallel Scientific Computation: A Structured Approach using BSP and MPI*. Oxford University Press: Oxford, UK, 2004.
6. Dagum L, Menon R. OpenMP: an industry standard API for shared-memory programming. *Computational Science and Engineering* 1998; **5**(1):46–55.
7. Chapman B, Jost G, *et al. Using OpenMP: Portable Shared Memory Parallel Programming*, Scientific and Engineering Computation Series, The MIT Press: Cambridge, MA, 2007.
8. Lecomber D. An object-oriented programming model for BSP computations. *Proceedings of the PPECC Workshop on Parallel and Distributed Computing*, 1994.
9. Dehne F, Fabri A, *et al*. Scalable parallel computational geometry for coarse grained multicomputers. *International Journal on Computational Geometry and Applications* 1996; **6**(3):379–400.
10. Chan A, Dehne F. CGMgraph/CGMlib: implementing and testing CGM graph algorithms on PC clusters and shared memory machines. *International Journal of High Performance Computing Applications* 2005; **19**:81–97.
11. Yelick K, Semenzato L, *et al*. Titanium: a high-performance Java dialect. *Concurrency: Practice and Experience* 1998; **10**(11–13):825–836. DOI: 10.1002/(SICI)1096-9128(199809/11)10:11/13h825::AID-CPE383i3.0.CO;2-H.
12. Kaminsky A. Parallel Java: A unified API for shared memory and cluster parallel programming in 100% Java. In *International Parallel and Distributed Processing Symposium*. IEEE Press: Long Beach, CA, USA, 2007; 1–8.
13. Hinsen K. High-level parallel software development with Python and BSP. *Parallel Processing Letters* 2003; **13**(3):473–484.
14. Tiskin A. The bulk-synchronous parallel random access machine. *Theoretical Computer Science* 1998; **196** (1–2):109–130. DOI: 10.1016/S0304-3975(97)00197-7.
15. Hou Q, Zhou K, *et al*. BSGP: bulk-synchronous GPU programming. *ACM Transactions on Graphics* August 2008; **27**(3):19.1–19.12.
16. Vastenhouw B, Bisseling RH. A two-dimensional data distribution method for parallel sparse matrix–vector multiplication. *SIAM Review* 2005; **47**(1):67–95.
17. Devine KD, Boman EG, *et al.* Parallel hypergraph partitioning for scientific computing. In *International Parallel and Distributed Processing Symposium*. IEEE Press: Long Beach, CA, USA, 2006; 102.
18. Yzelman AN, Bisseling RH. Cache-oblivious sparse matrix–vector multiplication by using sparse matrix partitioning methods. *SIAM Journal on Scientific Computing* 2009; **31**(4):3128–3154.
19. Toledo S. Improving the memory-system performance of sparse-matrix vector multiplication. *IBM Journal of Research and Development* 1997; **41**(6):711–725.
20. Williams S, Oliker L, *et al*. Optimization of sparse matrix-vector multiplication on emerging multicore platforms. *Parallel Computing* 2009; **35**(3):178–194. DOI: 10.1016/j.parco.2008.12.006.
21. Yzelman AN, Bisseling RH. Two-dimensional cache-oblivious sparse matrix–vector multiplication, 2011. Preprint.
22. Frigo M, Johnson SG. FFTW: An adaptive software architecture for the FFT. In *Proceedings IEEE International Conference on Acoustics, Speech, and Signal Processing*, Vol. 3. IEEE Press: Los Alamitos, CA, 1998; 1381–1384.
23. Golub GH, Van Loan CF. *Matrix Computations*, 3rd ed., Johns Hopkins Studies in the Mathematical Sciences. The Johns Hopkins University Press: Baltimore, MD, 1996.
24. Goto K, Milfeld K. GotoBLAS2. http://www.tacc.utexas.edu/tacc-projects/gotoblas2, [February 2011].
25. Whaley RC, Petitet A, *et al*. Automated empirical optimizations of software and the ATLAS project. *Parallel Computing* 2001; **27**(1–2):3–35.
26. Vuduc R, Demmel JW, *et al*. OSKI: a library of automatically tuned sparse matrix kernels. *Journal of Physics: Conference Series* 2005; **16**:521–530.

27. Yzelman AN, Bisseling RH. A cache-oblivious sparse matrix–vector multiplication scheme based on the Hilbert curve. In *Progress in Industrial Mathematics at ECMI 2010*. Springer: Berlin, 2011. in press.
28. Buluç A, Fineman JT, *et al*. 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*. ACM: New York, NY, USA, 2009; 233–244. http://doi.acm.org/10.1145/1583991.1584053.
29. Martone M, Filippone S, *et al*. Utilizing recursive storage in sparse matrix–vector multiplication – preliminary considerations. In *Proceedings of the ISCA 25th International Conference on Computers and Their Applications (CATA)*, Philip T (ed.). ISCA: Hawaii, USA, 2010; 300–305.

DOI: 10.1002/cpe