DIGITAL ACCESS TO SCHOLARSHIP AT HARVARD
DASH.HARVARD.EDU

HARVARD LIBRARY
Office for Scholarly Communication

Communicable Memory and Lazy Barriers for Bulk Synchronous Parallelism in BSPk

Citation
Fahmy, Amr and Abdelsalam Heddaya. 1996. Communicable Memory and Lazy Barriers for Bulk Synchronous Parallelism in BSPk. Harvard Computer Science Group Technical Report TR-09-96.

Permanent link
http://nrs.harvard.edu/urn-3:HUL.InstRepos:24829615

Terms of Use
This article was downloaded from Harvard University's DASH repository, and is made available under the terms and conditions applicable to Other Posted Material, as set forth at http://nrs.harvard.edu/urn-3:HUL.InstRepos:dash.current.terms-of-use#LAA

Share Your Story

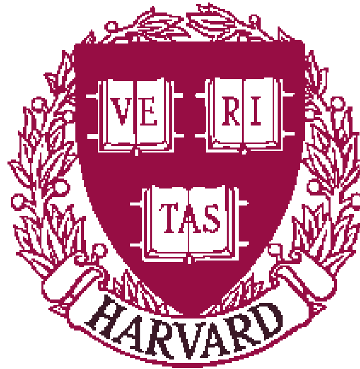
The Harvard community has made this article openly available.
Please share how this access benefits you. Submit a story .

Accessibility

Communicable Memory and Lazy Barriers for Bulk Synchronous Parallelism in BSPk

Amr Fahmy
Abdelsalam Heddaya

TR-09-96



Center for Research in Computing Technology
Harvard University
Cambridge, Massachusetts

Also available as BU-CS-96-012, Boston University Computer Science Department

Communicable Memory and Lazy Barriers for Bulk Synchronous Parallelism in BSPk

Amr Fahmy*
amr@das.harvard.edu
Aiken Computation Lab
Harvard University

Abdelsalam Heddaya*†
heddaya@cs.bu.edu
Computer Science Dept.
Boston University

September 20, 1996

Abstract

Communication and synchronization stand as the dual bottlenecks in the performance of parallel systems, and especially those that attempt to alleviate the programming burden by incurring overhead in these two domains. We formulate the notions of *communicable memory* and *lazy barriers* to help achieve efficient communication and synchronization. These concepts are developed in the context of BSPk, a toolkit library for programming networks of workstations—and other distributed memory architectures in general—based on the Bulk Synchronous Parallel (BSP) model. BSPk emphasizes efficiency in communication by minimizing local memory-to-memory copying, and in barrier synchronization by not forcing a process to wait unless it needs remote data. Both the message passing (MP) and distributed shared memory (DSM) programming styles are supported in BSPk. MP helps processes efficiently exchange short-lived unnamed data values, when the identity of either the sender or receiver is known to the other party. By contrast, DSM supports communication through between processes that may be mutually anonymous, so long as they can agree on variable names in which to store shared temporary or long-lived data.

*Research supported in part by NSF grant MCB-9527181.

†Some of this work was done while this author was on sabbatical leave at Harvard University's Aiken Computation Lab and Dept. of Biological Chemistry and Molecular Pharmacology.

1 Introduction

The economical programming of parallel machines is hampered by lack of consensus on a universal intermediate machine model that provides: (1) *efficiency* of implementation: (2) cost model *simplicity* and *accuracy*, (3) *architecture independence*, and (4) programming *convenience*. In an extended abstract of this paper [9], we describe the design of BSPk[1], a library and toolkit that we propose as a candidate that meets the above requirements. Here, we elaborate on our design rationale and include detailed implementation notes and BSPk programming examples. The bulk synchronous parallel (BSP) model [21] represents the conceptual foundation of BSPk, enabling the latter to serve as a host for direct programming, as a run-time system, and as a target for optimizing compilation of increasingly high level languages.

BSPk offers three independent contributions: First, we support zero-copy communication by taking over the management of dynamic memory, so as to provide the new low-level abstraction of *communicable memory*. Second, we develop the notion of *lazy barriers*—implemented as message counting logical barriers—and provide the means by which the program can declare its communication pattern, so that the barrier synchronization overhead can be reduced to zero when possible. Third, we elaborate and implement a BSP *programming model* that incorporates both message passing and distributed shared memory[2], which we view as complementary rather than mutually exclusive.

Our approach in BSPk is consistent with the recent trend in the operating system and data communication research communities towards application level communication, synchronization, and resource management. We base our communicable memory on the design principle of *application level framing* (ALF) proposed in [5]. ALF stipulates that the application break up its communication units into packet frames that can be sent over the network without being copied, segmented or sequenced. This minimizes the CPU and memory overheads, and permits more flexible congestion control protocols. User-level communication systems that successfully employ similar ideas to achieve very low overhead include U-Net [24] and NX/Shrimp [2]. Both of these systems aim to support parallel applications that are able to present their data units for communication in a suitable form. BSPk does precisely that, and therefore is poised to take direct advantage of the efficiencies afforded by such systems as U-Net and NX/Shrimp. At the bottommost layer, an *ex-*

[1]BSPk, pronounced "bespeak," stands for bulk synchronous parallelism toolkit.

[2]Throughout this paper, we use *distributed shared memory* to mean that the memory is partitioned into a collection of remotely accessible local modules.

okernel [8] would export hardware resources directly to user-level communication subsystems, thus enabling them to realize their promised performance without jeopardizing protection. Hence, we view the layering from top to bottom as follows: parallel application, BSPk, user-level communication subsystem, exokernel.

There exists a number of programming systems that implement the BSP algorithmic model in ways that differ from ours in BSPk. These include: the Split-C programming language [7], the Oxford BSP library [19], and the Green BSP library [13]. Split-C and Oxford BSP support distributed shared memory, while Green BSP provides for message passing. An effort has recently been mounted to standardize on a library called BSP Worldwide [12], which supports both MP and DSM, but without integrating the underlying memory management support as BSPk does. BSPk differs from all of these systems in supporting zero-copy communication and message counting logical barriers. A more detailed discussion of each of these libraries can be found in Section 7. Also, an attempt towards defining operating system requirements for the support of the BSP model can be found in [14].

In the next section, we summarize the salient features of the BSP model, then give an overview of BSPk in Section 3. Sections 4 and 5 cover the concepts of communicable memory and lazy barriers, including methods of their implementation. In Section 6 we offer BSPk programming examples in the styles of message passing and distributed shared memory. Other systems that implement the BSP programming model are reviewed and contrasted with BSPk in Section 7, which is followed by a general discussion section.

2 The BSP Model

The bulk synchronous parallel (BSP) algorithmic model [21] forms the basis of the BSPk programming model. A BSP computation is structured as a sequence of *supersteps* each followed by a barrier synchronization. A superstep, in turn, is a sequence of local actions and remote communication requests, be they message sends and receives, or distributed shared memory (remote) fetches and stores. Within a single superstep, a BSP process (or thread) executes until it issues a barrier call, at which time it is suspended, and the physical processor switches its context to a ready thread (see Figure 1). When all of the local threads are suspended, the processor becomes idle until the network delivers to it all of the values sent to it, or that it requested.

Strictly speaking, multithreading is not required by the BSP model, however it offers a number of

Figure 1: Structure of a BSP computation in which each processor executes four BSP processes. A solid circle represents a barrier invocation, and a solid horizontal line represents its return.

advantages that are critical for achieving performance without sacrificing programming convenience. These benefits can be summarized as follows:

- Threads can trust each other to share context, and hence enjoy low context-switching overhead.
- The programmer can express the number of threads that is most suitable for the available concurrency in the program, yet the system can choose to run as many of these on a single node as is consistent with high efficiency or utilization.
- Multithreading enables the system to hide communication latency, without forcing the programmer to write code that executes instructions while awaiting the completion of asynchronous communication.
- Threads tend to have such light contexts that the cost of migrating them is low enough to make dynamic load balancing feasible at a fine grain.

BSP grew out of an earlier effort to validate the practicality of the PRAM algorithmic model [22], for which most of the existing parallel algorithms have been designed [1, 16], but whose realization on practical hardware suffers from serious performance handicaps.[3] While the spirited defense of

[3]The PRAM stipulates the existence of a constant cost shared random access memory, and that the parallel

the PRAM mounted by some theoreticians [23] deserves a fair experimental shake, the merits of BSP as a model do not hinge exclusively on its success in mediating efficiently between the PRAM and the hardware. BSP can be programmed directly, and can function as a target for compilation from a variety of higher level models other than the PRAM.

The quantitative aspects of BSP deserve brief mention, so as to permit the reader to judge the level of its simplicity in comparison to the simpler PRAM on the one hand, and to the more complex communication-topology-aware models [18] on the other. Two parameters capture the communication and synchronization costs involved in running BSP programs [21]. The *communication cost* g represents the computation-to-communication throughput ratio; g is measured by the mean interval between successive words delivered by the network to a processor, and is expressed in units of processor cycles per word. The *synchronization cost* L, denotes the number of cycles required to achieve barrier synchronization across all processors. Thus, BSP ignores the topology of the network in the sense that all nodes are considered equidistant, and disregards any other special purpose hardware that might exist in the machine, except to the extent that it influences the values of L and g. The competing LogP model [6] adds one more parameter—the minimum intersend interval—to the mix, which, in the authors' view, does not change the essence of the model. For an example of a BSP algorithm that demonstrates the power of the model, see the *provably optimally portable*[4] matrix multiplication one described in [4].

BSP suffers from the apparent problem of having to incur the full cost of barrier synchronization, even when weaker ordering would be acceptable by the application. The impact of this shortcoming is worsened when L is large, hence the possible need for specialized synchronization hardware. BSPk is intended to demonstrate that L can be made very small—as small as zero—in certain very common cases, and that the worst case value of L, even for all-software barriers, can be kept within a small multiple of the corresponding value of L for hardware synchronizers.

3 Overview of BSPk

At the core of BSPk lies a small set of memory and synchronization management primitives, in terms of which both message passing and remote memory access operations are defined. Table 1 lists all the important components of the BSPk interface. The BSPk memory allocation and

program's instructions execute in lock-step synchrony.

[4]An algorithm is optimally portable if it can run at constant efficiency—or utilization—that remains independent of the number of processors p, over the widest range of values for the model parameters g and L.

handling primitives provide the *communicable memory* (comem) abstraction which appears to the user program as contiguous regions of dynamic heap memory, yet is internally represented so as to permit *application level framing* (ALF) [5], and hence zero-copy communication when sent across the network. This is achievable because the user computes in the same communicable memory that serves as the communication buffer. Comem pointers identify a single element in a comem region, and can be dereferenced and incremented using the BSPk primitives shown in Table 1.

Our design aims to enable the *pipelined* execution of BSP program supersteps whenever possible, so as to reduce or eliminate the cost of the barrier synchronizations required at the end of every superstep. Towards this goal, we develop *lazy barriers*, which employ message counting to guarantee zero-overhead barriers for programs that can predeclare the number of messages to be received in each superstep. For programs that cannot do so, BSPk will compute the number of messages that each process (thread) should expect to receive in each superstep.

The BSPk superstep offers programming convenience akin to that of critical sections and atomic transactions; the programmer can safely ignore concurrency during the superstep. This is achieved by tying the concurrency semantics of the communication primitives to barrier synchronization: they all appear to take effect at the barrier. We refer the reader to Section 6 for detailed BSPK programming examples illustrating MP and DSM communication styles.

In BSPk, both the message passing (MP) *send* and *receive* operations, and the distributed shared memory (DSM) *copy*[5] primitive, integrate seamlessley with communicable memory, and with the barrier synchronization provided by the *sync* call (see Table 1). All communication appears to take place at superstep boundaries. MP *receive* operations in a given superstep return messages that were sent in the previous superstep.[6] DSM *copy* calls also take effect at the point of the subsequent *sync*, with the constraint that copy operations that fetch data from remote comem regions to local ones must logically follow those that store data. Thus, fetch operations whose results are needed in a superstep i, must be issued in superstep $i-1$, and they are guaranteed to observe the effects of all relevant store operations issued before superstep i. In effect, the programmer can write code as if there is no concurrency during each superstep. Moreover, MP and DSM communication functions operate directly on comem regions, at once simplifying programming, and enabling efficient implementation.

[5]To perform a *fetch*, the program requests a copy from a remote comem region into a local one; and conversely to *store*.

[6]Any unclaimed messages that were sent but never received are destroyed.

Table 1: Interface functions of BSPk.

Class	Primitive	Description
Cummunicable memory	bspk_malloc(VarName, VarIdx, N, S)	Allocates, in local memory, a comem region to store N elements, each of size S bytes. Registers the comem region under the given name and index. Returns a comem pointer that is globally "dereferencable" via the DSM function bspk_copy, and locally dereferencable using bspk_elemP.
	bspk_lookup(VarName, VarIdx)	Returns a (possibly remote) comem pointer to a comem object registered under the given name and index.
	bspk_free(LocalComemPtr)	Free the given local comem region.
Comem pointers	bskp_elemP(LocalComemPtr)	Returns an ordinary pointer to the comem element pointed to by LocalComemPtr. *No pointer arithmetic is allowed on the returned pointer; the programmer must use* bspk_incr *for this purpose.*
	bspk_incr(LocalComemPtr, Amount)	Returns a copy of LocalComemPtr, incremented by Amount.
Synchronization	bspk_sync( )	Barrier synchronization; defines the boundary between two supersteps, and guarantees the logical ordering of MP and DSM memory operations performed in different supersteps.
	bspk_expect(NumMsgs) bspk_will_send(NumMsgsArr)	Announce the number of messages expected to be received from all other processes, or to be sent to every other process, in the current superstep. This should *not* include lookups, but should account for send, recv, and copy operations of which the local process is either the source or the destination. *In each BSPk superstep, if a thread issues one of these calls, it should be the same type of call invoked by other threads.*
Message passing (MP)	bspk_send(Dest, LocalComemPtr, N)	Send N elements from local comem region to process Dest, starting from the comem element pointed to by LocalComemPtr.
	bspk_recv() bspk_sender_id(LocalComemPtr)	bspk_recv returns a pointer to a freshly allocated local comem region containing a copy of one that was sent in the previous superstep. Returns NULL_COMEM_PTR if all such messages have been received. The sender can be identified via bspk_sender_id.
Distributed shared memory (DSM)	bspk_copy(DestComemPtr, SrcComemPtr, N)	Copy N comem elements from source comem region to destination, starting from the given element positions. The data becomes visible only in the next superstep. *At least one of the two comem regions must be local.*

The reason we support both MP and DSM is that we believe they complement each other. For example, explicit message passing is useful in efficiently transfering temporary unnamed values between processes that are known to each other (actually, it suffices for either the sender or the receiver(s) to identify the other). In this case a straightforward MP *send* eliminates the potential extra delay in DSM accesses when they need to go through a third party because it happens to have to own the memory through which communication is taking place. By contrast, distributed shared memory supports access to long-lived named data structures. DSM may also be used to hold temporary values whose producer(s) and consumer(s) can agree on a name to associate with the data, despite mutual anonymity. This suggests that both MP and DSM may profitably be mixed even in the same program.

4 Communicable Memory

The BSPk memory allocation and handling primitives provide the *communicable memory* abstraction which appears to the user program as contiguous heap memory, yet is internally represented so as to permit application level framing (ALF) [5], and hence zero-copy communication when sent across the network. This can be achieved because the user computes in the same communicable memory that serves as the communication buffer. Comem is applicable not only to BSPk, but also to any other parallel system that must move data between different memory modules over the network, *i.e.*, almost all systems.

The responsibility for allocating buffers for messages or copies of remote shared memory regions, rests with the library's `bspk_malloc` function (see Table 1), which reserves enough contiguous space for both user-visible message contents and header information private to the library. As a result of this, BSPk messages and DSM objects are variable sized comem regions, that appear to the programmer as contiguous regions, but that are implemented as collections of packet frames sized to match the characteristics of the network. Variable size in comem regions helps reduce false-sharing because each comem region can be used to store one logical object. Another advantage of variable size is portability, since the natural packet size of the system can be used to transmit data, while the application sees the size that is appropriate for it.

BSPk programs are able to maintain a collection of comem objects that can represent arbitrarily distributed pointer data structures. The `bspk_copy` operation allows access to remote comem regions. By making comem pointers usable as is from any node in the system, copying can be carried

out without any need for pointer modifications of transmitted pointer structures. Thus, comem lays the ground work for supporting efficient communication of complicated data structures [10].

Implementing Comem

An example implementation of a comem is shown in Figures 2 and 3. A comem region consists of a set of constant size packet frames, and an index array whose length is determined at the time of invocation of `bspk_malloc`. It is the packet frames that make comem regions efficiently *communicable*, because their size and structure are chosen to fit the particular network interface hardware and communication protocol implementation. If these lower levels of the system are designed to deal with packet frames that are prepared in the application address space—the application in this case being the the BSPk library—then sending a packet onto the wire requires no unnecessary memory-to-memory copying of packet contents.[7] At the same time, the index array enables pointer arithmetic to be performed quickly, in time that is independent of the number of packet frames that make up a given comem region.

The BSPk programmer treats the comem pointer as if it were pointing to a contiguous sequence of elements, whose size is stored in the comem pointer structure to aid in performing pointer arithmetic. The comem abstraction comes at the cost of the longer time it takes to dereference a comem pointer, compared to dereferencing an ordinary memory pointer.

Two different dynamic memory pools would be used to implement the BSPk memory allocator: one for the constant size packet frames, and another, much smaller one, for index arrays. Allocating and deallocating memory from the first pool can be done very simpley and quickly. Because the index arrays are so much smaller, in proportion to the packet frames to which they point, it should be possible to restrict the possible sizes that can be allocated for index arrays, so as to guarantee fast allocation of index arrays as well.

5 Lazy Barriers

The BSP model stipulates that computation proceed in *supersteps* explicitly denoted in the program text. This seems to incur a synchronization penalty for every global communication phase. We view this as purely an *ordering* requirement on communication steps, not as a real-time synchronization requirement. Therefore, all of our communication primitives are defined to respect *only* the ordering

[7] Examples of such communication subsystems include U-Net [24] and NX/Shrimp [2].

```
typedef struct {
        char[24] GlobVarName;   /* Optional */
        int      GlobVarInx;    /* Optional */
        int      NumElems;
        int      ElemSize;
        int      OwnerPID;
        int      SenderPID;     /* Used only in receive buffers. */
        void *   BasePtr;
        int      Offset; }      comemPtrType;
```

Figure 2: An example implementation of the comem pointer type.

Figure 3: A possible memory layout of a comem region.

of communication events, relative to the boundaries of the supersteps. To achieve this, we employ a message counting scheme that enables data messages to be used to trigger the beginning of new supersteps, in the pipelined fashion as that of the data messages. When this works well, the ordering requirement of bulk synchrony is satisfied without any waiting incurred beyond that needed simply to deliver the data messages. In other words, barrier synchrony can cost nothing[8] under certain circumstances.

Message Counting and Declaration of Communication Pattern

If the number of messages r_i that process i has to receive during the current superstep, is known *a priori*, no waiting is necessary beyond that which is required to receive the data messages. In many parallel algorithms this number is known, an example, among many others, is the BSP algorithm for matrix multiplication [4]. Using this observation, BSPk supports three primitives to achieve the effect of barriers.

At the end of each superstep `bspk_sync()` is called. The behavior of this call depends on the completion of the communication steps pertaining to the process on which it was called; if all the messages that a process is to receive have arrived and if all the messages that a process has to send are out, the process is allowed to proceed to the next superstep *without* waiting for the rest of the processes to finish the superstep.

Two more primitives are used to make available to BSPk the communication pattern of the processes, and thus r_i, for the current superstep. The call `bspk_expect(m)` informs BSPk to expect m messages during the current superstep. This primitive is used when the communication pattern of the superstep is known. Note that the cost of achieving the effect of barrier synchronization is zero in this case. However, in some algorithms, the communication pattern is not known *a priori* and it must be computed. The function call `bspk_will_send(v)`, where `v` is a vector whose length is the number of processes, informs BSPk that the process will send `v[j]` messages to process `j` during the current superstep. Using this information, BSPk calculates r_i for all i by computing the element-wise sum of the all the vectors declared by all processes in their `bspk_will_send(v)` call. This is done in $\log p$ steps where p is the number of processes. Subsequently, BSPk broadcasts the sum in another $\log p$ steps. The Oxford BSP library [15] achieves the effect of barriers by performing a similar computation on every superstep, whether it is needed or not, by calculating the number of messages that each process will receive during the current superstep using a hypercube-embedded

[8] Except for the cost of sequencing data messages to identify the superstep in which they were sent.

tree.

6 Example BSPk Programs

The following examples can be viewed as program templates. In them, we include all the detail that is relevant to BSPk. It is possible, indeed recommended, that a single program contain both MP and DSM primitives. As we mentioned earlier, MP and DSM communication suit different purposes, and these can easily exist simultaneously in the same program.

The two examples given abide by the following programming rules:

1. Memory for communication should always be allocated via BSPk.

2. User program should call `bspk_free` to recycle memory used in a communication primitive, but not before the subsequent `bspk_sync`.

3. Invoke `bspk_copy` to fetch remote data only when necessary, because it forces the destination process to wait for its own `bspk_sync` and for reception of all its messages, before responding to the fetch request.

BSPk programming, be it in the MP or the DSM style (or both intermixed), requires the dereferencing of comem pointers that are local. This is done via the BSPk function `bspk_elemP`. Pointer arithmetic can be performed using `bspk_incr`.

6.1 Message Passing

A generic message passing program template is illustrated in Figure 4. The shown superstep first frees the comem regions that were already sent as messages in the previous superstep, then reels in all the messages that need to be received in the current superstep. Every new message is allocated implicitly by `bspk_recv`, and a pointer to it is what's returned to the program. The core of the superstep is the loop that is statement (4) in the figure. Each iteration allocates a fresh comem region in which to compute a new data value, then send it. Just before finally invoking the barrier, at the point when the received messages are no longer useful, their comem regions are freed.

The large number of calls to `bspk_free` to deallocate received messages is necessary so as to allow them to be reused by `bspk_recv` in the next superstep. The same is not necessarily true, however, for comem regions used to compute and send data; those can be reused directly by the program, so long as their sizes are appropriate. Because the size of fundamental unit of memory

```
Pi:     ...
  (0) bspk_sync();
  (1) for (j = 0; j < NumSent; j++)
      (1.1) bspk_free(OutMsg[j]);    /* Free previously sent msg's. */
  (2) NumRcvd = −1;
  (3) do    {    /* Pull in all pending msgs. */
      (3.1) NumRcvd = NumRcvd + 1;
      (3.2) InMsg[NumRcvd] = bspk_recv();
      (3.3) } until (InMsg[NumRcvd] == NULL);
  (4) for (NumSent = 0; [More work to do]; NumSent++) {
      (4.1) OutMsg[NumSent] = bspk_malloc("", 0, NumElems, ElemSize);
      (4.2) [Compute contents of OutMsg[NumSent], using received messages in array InMsg[*].]
      (4.3) bspk_send(DestProc, OutMsg[NumSent]); }
  (5) for (j = 0; j < NumRcvd ; j++)
      (5.1) bspk_free(InMsg[j]);    /* Free msgs received and no longer needed. */
  (6) bspk_sync();
      ...
```

Figure 4: A BSPk message passing program sketch for a superstep of process P_i. Step 3.2 (framed) represents the application's work in the shown superstep.

allocation by BSPk is actually constant throughout an entire execution, a fast and simple memory allocator should be easy to build.

6.2 Distributed Shared Memory

Consider the problem of iteratively updating an array, where the new value of each element is a function of the four elements directly above, below to the left and to the right of the element. This is a common pattern of computation such as in the solution of partial differential equations, for example.

A parallel implementation of this problem may allocate different slices of the array to different processes such that each process communicates with the two processes to its north and to its south. A program sketch that demonstrates the use of BSPk's DSM primitives appears in Figure 5. Initially, comem is allocated for the parts of the array that need to be transmitted, ToNorth and ToSouth, and for values that are needed from other processes, FromNorth and FromSouth, see

```
Pi:     ...
  (0) FromNorth = bspk_malloc("FromNorth", MyId, NumElems, ElemSize);
  (1) FromSouth = bspk_malloc("FromSouth", MyId, NumElems, ElemSize);
  (2) ToNorth = bspk_malloc("ToNorth", MyId, NumElems, ElemSize);
  (3) ToSouth = bspk_malloc("ToSouth", MyId, NumElems, ElemSize);
  (4) MyArray = malloc(ArraySize);
  (5) NorthBufName = bspk_lookup("FromSouth", north);
  (6) SouthBufName = bspk_lookup("FromNorth", south);
      ...
  (7) bspk_sync();
  (8) bspk_expect(numNeighbors);
  (9) [Compute data needed by north.]
  (10) bspk_copy("NorthBufName", ToNorth, NumElems);
  (11) [Compute data needed by south.]
  (12) bspk_copy("SouthBufName", ToSouth, NumElems);
  (13) [Compute contents of MyArray, using FromNorth and FromSouth updated in last superstep.]
  (14) bspk_sync();
      ...
```

Figure 5: A BSPk DSM program sketch demonstrating the use of communicable memory. Steps 0-6 constitute an initialization phase.

Figure 6. Each process then, has to look up by name the buffers thus allocated in the remote processes' memories where it will store the updated values. This is carried out using the `bspk_lookup` primitive. Note that the allocation of memory may not have happened before `bspk_lookup` is called but is guaranteed to have been completed at the end of the superstep.

This example also serves to illustrate the use of `bspk_expect` when the number of messages to be received is known *a priori*; in this case, two for each process that is allocated an interior slice of the array and one otherwise. The barrier synchronization for this example has zero cost.

7 Other BSP Systems

The Bulk Synchronous Parallel computing model (BSP) is currently supported by Split-C [7], by the Oxford BSP library [19], and by the Green BSP library [13]. An effort has been mounted to standardize on a library called BSP Worldwide [12]. We discuss each of these in turn in this section.

Figure 6: Slice of the array for the DSM example allocated to a single process, shaded areas represent comem regions.

Systems that support high performance user-level communication include U-Net [24] and NX [2].

The main problems we found with other BSP systems concern superfluous restrictions and inefficiencies in the use of memory for communication, and extra waiting for synchronization even when the communication pattern is known to the application program in advance. Thus, we remove several restrictions on communication, while simultaneously designing our interface so as to enable higher performance communication and synchronization. In particular, BSPk differs from all the systems mentioned below in supporting comem and lazy barriers, in ensuring that a DSM *fetch* reads from the last superstep preceding the one in which the data will be used, and in accepting directives from the user program regarding the number of messages to expect in a given superstep. BSPk places no extraneous restrictions on conflicting concurrent DSM operations, nor does it incur in-memory copying costs for communication of distributed dynamic data structures.

Split-C. In supporting BSP style programming, Split-C focuses primarily on static DSM with no support for zero-copy communication, bulk-synchronous message passing, or low-overhead barriers. Split-C provides blocking *read/write* operations, asynchronous or locally bulk-synchronized *put/get* primitives, and a globally bulk-synchronous *store* (without a corresponding *fetch*). BSP style programs can be written in Split-C by using *get* and *store*, and marking each superstep boundary by a a pair of *sync* and *all-store sync* operations. The *sync* ensures that all *gets* are done, and the *all-store sync* waits for all the *stores* issued by all the processes to be acknowledged. Even though global pointers are allowed to point to C heap objects, Split-C's *get*, *put* and *store* operations all cause potentially unnecessary in-memory copying of data. BSPk doesn't include a *put* operation, which differs from *store* in that *put* is synchronized locally, while *store* is synchronized globally. Split-C

offers the *put* operation, in addition to *store*, because the destination process is not constrained to run in bulk-synchronous fashion, and the responsibility for synchronizing with the completion of the *put* must rest with someone.

Oxford BSP. This library also supports DSM communication, but not MP. Oxford BSP's *store* and *fetch* primitives both require copying of data from communication buffers belonging to the library to buffers managed by the user program. Conflicting operations (*e.g.*, two *stores*, or a *store* and a *fetch* involving the same memory location) are unnecessarily prohibited from being issued in the same superstep. As a result, a *fetch* must read data that was stored, not in the superstep immediately preceding the superstep in which the data is used, but from a superstep twice removed. In BSPk we relax these restrictions, and set the semantics of the operations so that the behavior is correct in such cases. On a network of workstations, Oxford BSP is implemented on top of TCP, which can be quite inefficient on LANs, given that TCP's message retransmission and congestion control are designed for continuous data streams over heavily shared WANs. TCP tends to be quite conservative and deferential in requiring positive acknowledgements, and in drastically reducing its transmission rate when faced with congestion. In summary, BSPk differs from Oxford BSP in the following major respects: (1) BSPk supports dynamically allocated distributed shared memory, (2) BSPk supersteps are purely sequential programs, from the point of view of the programmer, (3) BSPk's *copy* operations obtain values that could have been updated in the previous superstep, (4) BSPk's communication requires no unnecessary copying, (5) like BSP-WW (see below), BSPk integrates message passing with DSM.

Green BSP. This library supports bulk-synchronous MP of fixed-size packets, in contrast to Split-C and Oxford BSP's emphasis on DSM. We borrow the semantics of our *receive* operation from the Green BSP library, but relax the fixed-size message restriction, and enable messages to be sent without copying, as explained above.

BSP Worldwide. This is a nascent effort to propose a standard BSP library interface, based on the experience through Oxford BSP, and Green BSP libraries, and some of the applications developed using them [12]. The above comparison between BSPk and these two libraries applies to BSP-WW, except for BSP-WW's support for both MP and DSM.

8 Discussion

In order to implement BSPk, and realize the performance gains that its design promises, we need a user-level communication system such as U-Net [24]. This system will permit the efficient implementation of both comem and message counting barriers. Another important consideration has to do with process control utilities, and I/O. BSPk provides neither, which means that we must implement BSPk within an environment that does.

A number of algorithms and applications have been developed directly in the BSP model [3, 11, 20]. However, we view BSPk more as a general purpose foundation, on top of which it is possible to design and implement support for *global distributed data structures* [10], uniform address-space shared memory [17], multi-threading, and PRAM programming [23]. Each of these abstractions has demonstrable benefits that can be reaped only if BSPk proves in practice to be as efficient a testbed for their support as we hope.

One of the apparent limitations of BSPk has to do with its demand for barrier synchronizations, since the communication operations do not become visible until the next barrier is cleared. We feel that only programming experience will illuminate the extent to which this is a constraint, given its counter-balancing advantage in allowing the programmer to ignore concurrency completely in-between barriers.

Acknowledgements. Thanks to Tom Cheatham, Dan Stefanescu, Les Valiant, Bob Walton, and the rest of the BSP group at Harvard for many useful discussions and comments.

References

[1] Selim G. Akl. *Parallel sorting algorithms.* Academic Press, Orlando, Florida, 1985.

[2] R.D. Alpert, C. Dubnicki, E.W. Felten, and K. Li. Design and implementation of NX message passing using Shrimp virtual memory mapped communication. In *Proc. of the 1996 International Conference on Parallel Processing, Bloomingdale, Illinois*, Aug. 12–16 1996.

[3] R.H. Bisseling and W.F. McColl. Scientific computing on bulk synchronous parallel architectures (short version). In B. Pehrson and I. Simon, editors, *Proc. 13th IFIP World Computer Congress (Volume 1)*. Elsevier, 1994. Full paper available as technical report 836, Dept. of Math, Univ. of Utrecht, Holland.

[4] T.E. Cheatham, A. Fahmy, D.C. Stefanescu, and L.G. Valiant. Bulk synchronous parallel computing: A paradigm for transportable software. In *Proc. 28th Hawaii International Conference on System Sciences, Maui, Hawaii*, Jan. 1995.

[5] D.D. Clark and D.L. Tennenhouse. Architectural considerations for a new generation of protocols. In *Proc. ACM SICGOMM'90*, pages 200–208, Sep. 1990. Published as special issue of Computer Communication Review, vol. 20, number 4.

[6] D. Culler, R. Karp, D. Patterson, A. Sahay, K.E. Schauser, E. Santos, R. Subramonian, and T. von Eicken. LogP: Towards a realistic model of parallel computation. In *Proc. 4th ACM SIGPLAN Symp. on Principles and Practice of Parallel Programming, San Diego*, May 1993.

[7] D.E. Culler, A. Dusseau, S.C. Goldstein, A. Krishnamurthy, S. Lumetta, T. von Eicken, and K. Yelick. Introduction to Split-C: Version 1.0. Technical report, Univ. of California, Berkeley, EECS, Computer Science Division, April 1993.

[8] D.R. Engler, M.F. Kaashoek, and J. O'Toole Jr. Exokernel: An operating system architecture for application-level resource management. In *Proc. 15th ACM Symp. on Operating System Principles, Copper Mountain, Colorado*, Dec. 1995.

[9] A. Fahmy and A. Heddaya. BSPk: Low overhead communication constructs and logical barriers for Bulk Synchronous Parallel programming. *Bulletin of the IEEE Technical Committee on Operating Systems and Application Environments (TCOS)*, 8(2):27–32, Summer 1996. (Extended abstract).

[10] A.F. Fahmy and R.A. Wagner. On the distribution and transportation of data structures in parallel and distributed systems. Technical Report TR-27-95, Harvard University, December 1995.

[11] A.V. Gerbessiotis and L.G. Valiant. Direct bulk-synchronous parallel algorithms. *Journal of Parallel and Distributed Computing*, 22, 1994.

[12] J.W. Goudreau, J.M.D. Hill, K. Lang, B. McColl, S.B. Rao, D.C. Stefanescu, T. Suel, and T. Tsantilas. A proposal for the BSP Worldwide standard library (preliminary version). Technical report, Oxford Parallel Group, Oxford Univ., April 1996. Available as URL `http://www.bsp-worldwide.org/standard/stand2.htm`.

[13] M.W. Goudreau, K. Lang, S.B. Rao, and T. Tsantilas. The Green BSP library. Technical Report CS-TR-95-11, Dept. of Computer Science, Univ. of Central Florida, June 1995.

[14] A. Heddaya and A.F. Fahmy. OS support for portable bulk synchronous parallel programs. Technical Report BU-CS-94-013, Boston Univ., Computer Science Dept., Dec. 1994.

[15] Jon Hill. The Oxford BSP toolset and profiling system. Source code available through `http://www.comlab.ox.ac.uk/oucl/oxpara/`, Aug. 1996.

[16] J. JàJà. *An Introduction to Parallel Algorithms.* Addison-Wesley, Reading, Massachusetts, 1992.

[17] Richard P. LaRowe and Carla Schlatter Ellis. Experimental comparison of memory management policies for NUMA multiprocessors. *ACM Trans. on Computer Systems*, 9(4):319–363, Nov. 1991.

[18] F. Thomas Leighton. *Introduction to Parallel Algorithms and Architectures: Arrays, Trees, Hypercubes*, volume I. Morgan Kaufmann, San Mateo, California, 1992.

[19] R. Miller. A library for bulk synchronous parallel programming. In *Proc. British Comp. Soc. Parallel Processing Specialist Group Workshop on General Purpose Parallel Computing*, Dec. 22 1993.

[20] M. Nibhanupudi, C. Norton, and B. Szymanski. Plasma simulation on networks of workstations using the bulk synchronous parallel model. In *Proc. International Conference on Parallel and Distributed Processing Techniques and Applications, Athens, GA*, Nov. 1995.

[21] L.G. Valiant. A bridging model for parallel computation. *Comm. ACM*, 33(8):103–111, Aug. 1990.

[22] L.G. Valiant. General purpose parallel architectures. In Jan van Leeuwen, editor, *Handbook of Theoretical Computer Science*, volume I. Elsevier & MIT Press, Amsterdam, New York and Cambridge (Mass.), 1990.

[23] U. Vishkin. A case for the PRAM as a standard programmer's model. In F. Meyer auf der Heide, B. Monien, and A.L. Rosenberg, editors, *Parallel Architectures and their Efficient Use*, pages 11–19. Springer-Verlag, Nov. 11-13 1992.

[24] T. von Eicken, A. Basu, V. Buch, and W. Vogels. U-Net: a user-level network interface for parallel and distributed computing. In *Proc. 15th ACM Symp. on Operating System Principles, Copper Mountain, Colorado*, Dec. 1995.