Programming Research Group

QUESTIONS AND ANSWERS ABOUT BSP

D.B. Skillicorn
Department of Computing and Information Science
Queen's University, Kingston, Canada
skill@qucis.queensu.ca

Jonathan M.D. Hill and W.F. McColl
Computing Laboratory
University of Oxford
Oxford, U.K.
{Jonathan.Hill, Bill.McColl}@comlab.ox.ac.uk

Revised 11th November 1996

PRG-TR-15-96

Oxford University Computing Laboratory
Wolfson Building, Parks Road, Oxford OX1 3QD

Abstract

Bulk Synchronous Parallelism (BSP) is a parallel programming model that abstracts from low-level program structures in favour of *supersteps*. A superstep consists of a set of independent local computations, followed by a global communication phase and a barrier synchronisation. Structuring programs in this way enables their costs to be accurately determined from a few simple architectural parameters, namely the permeability of the communication network to uniformly-random traffic and the time to synchronise. Although permutation routing and barrier synchronisations are widely regarded as inherently expensive, this is not the case. As a result, the structure imposed by BSP comes for free in performance terms, while bringing considerable benefits from an application-building perspective. This paper answers the most common questions we are asked about BSP and justifies its claim to be a major step forward in parallel programming.

1 *Why is another model needed?*

In the 1980s a large number of different types of parallel architectures were developed. With hindsight we now see that this variety was both unnecessary and unhelpful. It stifled the commercial development of parallel applications software because, to achieve acceptable performance, all such software had to be tailored to the specific architectural properties of the machine.

Today the number of parallel computation models and languages probably greatly exceeds the number of different architectures with which parallel programmers had to contend ten years ago. Most are inadequate because they make it hard to achieve portability, hard to achieve performance, or both. Those based on message passing are inadequate because of the complexity of correctly creating paired communication actions (send and receive) in large and complex software. Such systems are prone to deadlock as a result. Furthermore, the performance of such programs is impossible to predict because of the interactions of large numbers of individual data transfers.

Some take the view that models based on shared memory are easier to program because they provide the abstraction of a single, shared address space and so a whole class of placement decisions are avoided. Moderately-parallel architectures capable of providing this abstraction can certainly be built, so they also believe that the modest parallelism they provide is enough to satisfy performance demands for the foreseeable future. We are dubious about both claims. While shared memory does reduce the need for placement, it creates a need to control simultaneous access to the same location. This requires either careful crafting of programs, in the PRAM style, or expensive lock management. Implementing shared-memory abstractions requires a larger and larger fraction of the computer's resources to be devoted to communication and the maintenance of coherence. Worse still, the technology required to provide the abstraction is the least likely to be of a commodity nature, and hence even more expensive.

The Bulk Synchronous Parallel (BSP) model [35] provides software developers with an attractive escape route from the world of architecture-dependent parallel software. The emergence of the model has coincided with the convergence of commercial parallel machine designs to a standard architectural form with which it is very compatible. These developments have been enthusiastically welcomed by a rapidly-growing community of software engineers who produce scalable and portable parallel applications. However, while the parallel-applications community has welcomed the approach, there is still a surprising degree of skepticism amongst

parts of the computer science research community. Many people seem to regard some of the claims made in support of the BSP approach as "too good to be true".

The only sensible way to evaluate an architecture-independent model of parallel computation such as BSP is to consider it in terms of *all* of its properties, that is (a) its usefulness as a basis for the design and analysis of algorithms, (b) its applicability across the whole range of general-purpose architectures and its ability to provide efficient, scalable performance on them, and (c) its support for the design of fully-portable programs with analytically-predictable performance. To focus on only one of these at a time, is simply to replace the zoo of parallel architectures in the 1980s by a new zoo of parallel models in the 1990s. It seems likely that this viewpoint on the nature and role of models will gain more and more support as we move from the straightforward world of parallel algorithms to the much more complex world of parallel software systems.

2 *What is Bulk Synchronous Parallelism?*

Bulk Synchronous Parallelism is a style of parallel programming developed for general-purpose parallelism, that is parallelism across all application areas and a wide range of architectures [25]. Its goals are more ambitious than most parallel-programming systems which are aimed at particular kinds of applications, or work well only on particular classes of parallel architectures [26].

BSP's most fundamental properties are that:

- *It is simple to write.* BSP programs look much the same as sequential programs. Only a bare minimum of extra information needs to be supplied to describe the use of parallelism.
- *It is independent of target architectures.* Unlike many parallel programming systems, BSP is designed to be architecture-independent, so that programs run unchanged when they are moved from one architecture to another. Thus BSP programs are portable in a strong sense.
- *The performance of a program on a given architecture is predictable.* The execution time of a BSP program can be computed from the text of the program and a few simple parameters of the target architecture. This makes design possible, since the effect of a decision on performance can be determined at the time it is made.

BSP achieves these properties by raising the level of abstraction at which programs are written and implementation decisions made. Rather than considering individual processes and individual communication actions, BSP considers computation and communication at the level of the entire program and executing computer. Determining the *bulk* properties of a program, and the *bulk* ability of a particular computer to satisfy them makes it possible to design with new clarity.

One way in which BSP is able to achieve this abstraction is by renouncing locality as a performance optimisation. This simplifies many aspects of both program and implementation design, and in the end does not adversely affect performance for most application domains. There will always be some application domains for which locality is critical, for example low-level image processing, and for these BSP may not be the best choice.

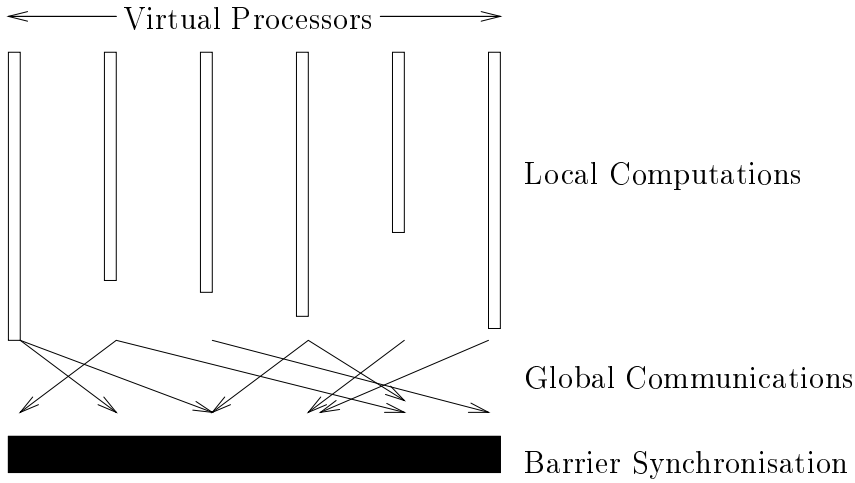

Figure 1: A Superstep

3 *What does the BSP programming style look like?*

BSP programs have both a vertical structure and a horizontal structure. The vertical structure arises from the progress of a computation through time. For BSP, this is a sequential composition of global *supersteps*, which conceptually occupy the full width of the executing architecture. Each superstep is further subdivided into three ordered phases consisting of:

- computation locally in each process, using only values stored in the memory of its processor;
- communication actions amongst the processes, involving movement of data between processors;
- a barrier synchronisation, which waits for all of the communication actions to complete, and which then makes the data that was moved available in the local memories of the destination processors.

The horizontal structure arises from concurrency, and consists of a fixed number of virtual processes. These processes are not regarded as having a particular linear order, and may be mapped to processors in any way. Thus locality plays no role in the placement of processes on processors. A superstep is shown in Figure 1.

We will use p to denote the virtual parallelism of a program, that is the number of processes it uses. If the target parallel computer has fewer processors than the virtual parallelism, an extension of Brent's theorem [5] can be used to transform a BSP program into a slimmer version.

4 *How does communication work?*

Most parallel programming systems handle communication, both conceptually and in terms of implementation, at the level of individual actions: memory-to-memory transfers, sends and receives, or active messages. However, this level is difficult to work with because there are many simultaneous communication actions in a parallel program, and their interactions are complex. This makes it hard to say much about the time any single communication action will take to complete.

Considering communication actions *en masse* both simplifies their treatment and makes it possible to bound the time it takes to deliver a whole set of data. BSP does this by considering all of the communication actions of a superstep as a unit. For the time being, imagine that all messages have a fixed size. During a superstep, each process has designated some set of outgoing messages and is expecting to receive some set of incoming messages. If the maximum number of incoming or outgoing messages per processor is h, then such a communication pattern is called an h-relation. The communication pattern in Figure 1 is a 2-relation. Because of the random placement of processes on processors, any structure on the set of messages in the abstract will almost certainly not be reflected as structure in the target architecture's communication topology. Thus the destination processor addresses of an h-relation are likely to approximate a sequence of *permutations* of processor identifiers.

The ability of communication network to deliver data is captured by a parameter, g, that measures the permeability of the network to continuous traffic addressed to uniformly-random destinations. Both the random placement of processes in processors, and techniques such as adaptive routing help to make the load generated by h-relations approximate the load generated by sequences of random permutations. Thus the applied load on the communication network has the kind of characteristics for which g is an appropriate measure. The parameter g is defined such that it takes time hg to deliver an h-relation. Subject to some small provisos, discussed later, hg *is* an accurate measure of communication performance over a large range of architectures. The value of g is normalised with respect to the clock rate of each architecture so that it is in the same units as the time for executing sequences of instructions.

Sending a message of length m clearly takes longer than sending a message of size 1. For reasons that will become clear later, BSP does not distinguish between a message of length m and m messages of length 1—the cost in either case is mhg. So messages of varying lengths may either be costed using the form mhg where h is the number of messages, or the message lengths can be folded into h, so that it becomes the number of units of data to be transferred.

The parameter g is related to the bisection bandwidth of the communication network but it is not equivalent. It also depends on other factors such as:

- the protocols used to interface with and within the communication network,
- the buffer management by both the processors and the communication network,
- the routing strategy used in the communication network, and
- the BSP runtime system.

So g is bounded below by the ratio of p to the bisection bandwidth, suitable normalised, but may be much larger. Only a very unusual network would have a bisection bandwidth that grew faster than p, so this means that g is a monotonically increasing function of p. The value

of g is, in practice, determined empirically for each parallel computer, by running suitable benchmarks. A BSP benchmarking protocol is given in Appendix B

Note that g is not the normalised single-word delivery time, but the single-word delivery time under continuous traffic conditions. This difference is subtle but crucial.

5 *Surely this isn't a very precise measure of how long communication takes? Don't hotspots and congestion make it very inaccurate?*

One of the most difficult problems of determining the performance of conventional messaging systems is precisely that congestion makes upper bounds hard to determine and quite pessimistic. BSP largely avoids this difficulty.

An apparently-balanced communication pattern may always generate hotspots in some region of the interconnection topology. BSP prevents this in several ways. First, the random allocation of processes to processors breaks up patterns arising from the problem domain. Second, the BSP runtime system uses routing techniques that avoid localised congestion. These include randomised routing [36], in which particular kinds of randomness are introduced into the choice of route for each communication action, and adaptive routing [4], in which data are diverted from their normal route in a controlled way to avoid congestion. If congestion occurs, as when an architecture has only a limited range of deterministic routing techniques for the BSP runtime system to choose from, this limitation on continuous message traffic is reflected in the measured value of g.

Notice also that the definition of an h-relation distinguishes the cost of a balanced communication pattern from one that is skewed. A communication pattern in which each processor sends a single message to some other (distinct) processor counts as a 1-relation. However, a communication pattern that transfers the same number of messages, but in the form of a broadcast from one processor to all of the others, counts as a p-relation. Hence, unbalanced communication, which is the most likely to cause congestion, is charged a higher cost. Thus the cost model does take into account congestion phenomena arising from the limits on each processor's capacity to send and receive data, and from the extra traffic that might occur on the communication links near a busy processor.

Experiments have shown that g is an accurate measure of the cost of moving large amounts of data on a wide range of existing parallel computers.

6 *Isn't it expensive to give up locality?*

Yes, there will always be application domains where exploiting locality is the key to achieving good performance. However, there are not as many of them as a naive analysis might suggest, for the following reason. Most performance-limited problems work with large amounts of data, and can therefore exploit large amounts of virtual parallelism. However, most existing parallel computers have only modest numbers of processors. When highly-parallel programs are mapped to much less parallel architectures, many virtual processes must be multiplexed onto each physical processor by the programmer. When this is done, almost all of the locality is lost, unless the communication network happens to match the structure of the problem domain very closely. Thus problems with apparently large amounts of locality tend not to have much locality when they actually execute.

7 Most parallel computers have a considerable cost associated with starting up communication. Doesn't this mean that the cost model is inaccurate for small messages, since g doesn't account for start-up costs?

The cost model can be inaccurate, but only in rather special circumstances. Recall that all of the communications in a superstep are regarded as taking place at the end of the superstep. This semantics makes it possible for implementations to wait until the end of the computation part of each superstep to begin the communication actions that have been requested. They can then package the data to be transferred into larger message units. The cost of starting up a data transfer is thus only paid once per destination per superstep and can be folded into the value of g.

However, if the **total** amount of communication in a superstep is small, then start-up effects may make a noticeable difference to the performance. We address this quantitatively later.

8 Aren't barrier synchronisations expensive? How are their costs accounted for?

Yes, barriers are often expensive on today's architectures and so they should be used as sparingly as possible. On the other hand, barriers are not nearly as inherently expensive as they are believed to be in high-performance computing folklore [17]. Future architecture developments may make them much cheaper.

The cost of a barrier synchronisation comes in two parts:

- The cost caused by the variation in the completion times of the computation steps that participate. There is not much that an implementation can do about this, but it does suggest that balance in the computation parts of a superstep is a good thing.
- The cost of reaching a globally-consistent state in all of the processors. This depends, of course, on the communication network, but also on whether or not special-purpose hardware is available for synchronising, and on the way in which interrupts are handled by processors.

For each architecture, the cost of a barrier synchronisation is captured by a parameter, l. The diameter of the communication network, or at least the length of the longest path that allows state to be moved from one processor to another clearly imposes a lower bound on l. However, it is also affected by many other factors, so that, in practice, an accurate value of l for each parallel architecture is obtained empirically.

Notice that barriers, although potentially costly, have a number of attractive features. There is no possibility of deadlock or livelock in a BSP program because barriers make circularities in data dependencies impossible. Hence there is no need for tools to detect and deal with them. Barriers also permit novel forms of fault tolerance.

9 How do these parameters allow the cost of programs to be determined?

The cost of a single superstep is the sum of three terms: the (maximum) cost of the local computations on each processor, the cost of the global communication of an h-relation, and

the cost of the barrier synchronisation at the end of the superstep. Thus the cost is given by

$$\text{cost of a superstep} = \mathop{\mathsf{MAX}}_{\text{processes}} w_i + \mathop{\mathsf{MAX}}_{\text{processes}} h_i\, g + l$$

where i ranges over processes, and w_i is the time for the local computation in process i. Often the maxima are assumed and BSP costs are expressed in the form $w + hg + l$. The cost of an entire BSP program is just the sum of the cost of each superstep. We call this the standard cost model.

To make this sum meaningful, and to allow comparisons between different parallel computers, the parameters w, g, and l are expressed in terms of the basic instruction execution rate of the target architecture. Since this will only vary by a constant factor across architectures, asymptotic complexities for programs are often given unless the constant factors are critically important. Note that we are assuming that the processors are homogeneous, although it is not hard to avoid that assumption by expressing performance factors in any common unit.

The existence of a cost model that is both tractable and accurate makes it possible to truly design BSP programs, that is to consciously and justifiably make choices between different implementations of a specification. For example, it is clear that the following strategies should be used to write efficient BSP programs:

- balance the computation in each superstep between processes, since w is a *maximum* over computation times, and the barrier synchronisation must wait for the slowest process;
- balance the communication between processes, since h is a *maximum* over fan-in and fan-out of data; and
- minimise the number of supersteps, since this determines the number of times l appears in the final cost.

The cost model also shows how to predict performance across target architectures. The values of p, w, and h for each superstep, and the number of supersteps can be determined by inspection of the program code, subject to the usual limits on determining the cost of sequential programs. Values of g, and l can then be inserted into the cost formula to estimate execution time before the program is executed. The cost model can be used

- as part of the design process for BSP programs;
- to predict the performance of programs ported to new parallel computers; and
- to guide buying decisions for parallel computers if the BSP program characteristics of typical workloads are known.

Other cost models for BSP have been proposed, incorporating finer detail. For example, communication and computation could conceivably be overlapped, giving a superstep cost of the form

$$\mathsf{MAX}(w, hg) + l$$

although this optimisation is not usually a good idea on today's architectures [16, 32]. It is also sometimes argued that the cost of an h-relation is limited by the time taken to send h messages and then receive h messages, so that the communication term should be of the form

$$(h_{in} + h_{out})g$$

All of these variations alter costs by no more than small constant factors, so we will continue to use the standard cost model in the interests of simplicity and clarity.

A more important omission from the standard cost model is any restriction on the amount of memory required at each processor. While the existing cost model encourages balance in communication and limited barrier synchronisation, it encourages profligate use of memory. An extension to the cost model to bound the memory associated with each processor is being investigated.

The cost model also makes it possible to use BSP to design algorithms, not just programs. Here the goal is to build solutions that are optimal with respect to total computation, total communication, and total number of supersteps *over the widest possible range of values of p.* Designing a particular program then becomes a matter of choosing among known algorithms for those that are optimal for the range of machine sizes envisaged for the application.

For example two BSP algorithms for matrix multiplication have been developed. The first, a block parallelization of the standard n^3 algorithm [26], has (asymptotic) BSP complexity

$$\text{Block MM cost } = n^3/p + (n^2/p^{1/2})g + p^{1/2}l$$

requiring memory at each processor of size n^2/p. This is optimal in time and memory requirement.

A more sophisticated algorithm due to McColl and Valiant [23] has BSP complexity

$$\text{Block and Broadcast MM cost } = n^3/p + (n^2/p^{2/3})g + l$$

requiring memory at each processor of size $n^2/p^{2/3}$. This is optimal in time, communication, and supersteps, but requires more memory at each processor. Therefore the choice between these two algorithms in an implementation may well depend on the relationship between the size of problem instances and the memory available on processors of the target architecture.

10 *Is BSP a programming discipline, or a programming language, or something else?*

BSP is a model of parallel computation. It is concerned with high-level structure of computations. Therefore it does not prescribe the way in which local computations are carried out, nor how communication actions are expressed. All existing BSP languages are imperative, but there is no intrinsic reason why this need be so.

BSP can be expressed in a wide variety of programming languages and systems. For example, BSP programs could be written using existing communication libraries such as PVM [9], MPI [27], or Cray's SHMEM. All that is required is that they provide non-blocking communication mechanisms and a way to implement barrier synchronisation. However, the values of g and l depend not only on the hardware performance of the target architecture

but also on the amount of software overhead required to achieve the necessary behaviour, so systems not designed with BSP in mind may not deliver good values of g and l.

The most common approach to BSP programming is SPMD imperative programming using Fortran or C, with BSP functionality provided by library calls. Two BSP libraries have been in use for some years: the Oxford BSP Library [28] and the Green BSP Library [11, 12]. A standard has recently been agreed for a library called BSPLib [13]. The BSPLib contains operations for delimiting supersteps, and two variants of communication, one based on direct-memory transfer, and the other on buffered message passing.

Other BSP languages have been developed. These include GPL [24], and Opal [21]. GPL is a first attempt to develop an MIMD language permitting synchronisation of subsets of executing processes. Opal is an object-based BSP language.

11 *How easy is it to program using the BSPLib library?*

The BSPLib library provides the operations shown in Table 1. There are operations to:

- set up a BSP program;
- discover properties of the environment in which each process is executing;
- participate in a barrier synchronisation;
- communicate, either directly into or out of a remote memory, or using a message queue;
- abort a computation from anywhere inside it; and
- communicate in a high-performance unbuffered mode.

The BSPLib library is freely available in both Fortran and C from `http://www.bsp-worldwide.org/implmnts/oxtool.htm`. A more complete description of the library can be found in Appendix A.

Another higher-level library provides specialised collective-communication operations. These are not considered as part of the core library, but they can be easily realised in terms of the core. These include operations for broadcast, scatter, gather, and total exchange.

12 *In what application domains has BSP been used?*

BSP has been used in a number of application areas, primarily in scientific computing. Much of this work has been done as part of contracts with Oxford Parallel (`http://www.comlab.ox.ac.uk/oxpara/`).

Computational fluid dynamics applications of BSP include: (a) an implementation of a BSP version of the OPlus library for solving 3D multigrid viscous flows, used for computation of flows around aircraft or complex parts of aircraft in a project with Rolls Royce [6], (b) a BSP version of FLOW3D, a computational fluid dynamics code, (c) oil reservoir modelling in the presence of discontinuities and anisotropies in a project with Schlumberger Geoquest Ltd.

Class	Operation	Meaning
Initialisation	`bsp_init`	Simulate dynamic processes
	`bsp_begin`	Start of SPMD code
	`bsp_end`	End of SPMD code
Enquiry	`bsp_pid`	Find my process id
	`bsp_nprocs`	Number of processes
	`bsp_time`	Local time
Synchronisation	`bsp_sync`	Barrier synchronisation
DRMA	`bsp_pushregister`	Make region globally visible
	`bsp_popregister`	Remove global visibility
	`bsp_put`	Push to remote memory
	`bsp_get`	Pull from remote memory
BSMP	`bsp_set_tag_size`	Choose tag size
	`bsp_send`	Send to remote queue
	`bsp_get_tag`	Match tag with message
	`bsp_move`	Fetch from queue
Halt	`bsp_abort`	One process halts all
High Performance	`bsp_hpput`	Unbuffered versions
	`bsp_hpget`	of communication
	`bsp_hpmove`	primitives

Table 1: Core BSP operations

Computational electromagnetics applications of BSP [30] include: (a) 3D modelling of electromagnetic interactions with complex bodies using unstructured 3D meshes, in a project with British Aerospace, (b) parallelisation of the TOSCA, SCALA, and ELEKTRA codes, and demonstrations on problems such as design of electric motors and permanent magnets for MRI imaging, (c) a parallel implementation of a time domain electromagnetic code ParEMC3d with absorbing boundary conditions, (d) parallelisation of the EMMA-T2 code for calculating electromagnetic properties of microstrips, wires and cables, and antennae [33].

There is also work involving parallelising the MERLIN code in a project with Lloyds Register of Shipping and Ford Motor Company. BSP has also been applied to plasma simulation at Rensselaer Polytechnic Institute in New York [31].

13 *What do BSP programs look like?*

Most BSP programs for real problems are large and it is impractical to include their source here. Instead we include some small example programs to show how the BSPLib interface can be used. We illustrate some different possibilities using the standard parallel prefix or scan operation: given $x_0, \ldots, x_{p-1}$ (with x_i stored on process i), compute $x_0 + \cdots + x_i$ on each process i.

All sums: version 1.

The function `bsp_allsums1` calculates the partial sums of p integers stored on p processors. The algorithm uses the logarithmic technique that performs $\lceil \log p \rceil$ supersteps, such that

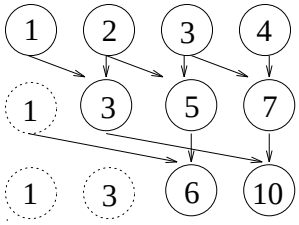

Figure 2: All sums using the logarithmic technique

during the k^{th} superstep, the processes in the range $2^{k-1} \leq i < p$ each combine their local partial sums with process $i - 2^{k-1}$. Figure 2 shows the steps involved in summing the values `bsp_pid()+1` using 4 processors.

```
int bsp_allsums1(int x) {
  int i, left, right;
  bsp_pushregister(&left,sizeof(int));
  bsp_sync();

  right = x;
  for(i=1;i<bsp_nprocs();i*=2) {
    if (bsp_pid()+i < bsp_nprocs())
      bsp_put(bsp_pid()+i,&right,&left,0,sizeof(int));
    bsp_sync();
    if (bsp_pid()>=i) right = left + right;
  }
  bsp_popregister(&left);
  return right;
}
```

When `bsp_put(bsp_pid()+i,&right,&left,0,sizeof(int))` is executed on process `bsp_pid()`, then a single integer `right` is copied into the memory of processor `bsp_pid()+i` at the address `&left+0` (where `left` is a previously-registered data structure).

The procedure `bsp_pushregister` allows all processors to declare that the variable `left` is willing to have data put into it during a DRMA operation. The reason that registration is required is that each processor's copy of the data structure `left` is not necessarily stored at the same address. Registration therefore creates a correspondence between data structures that have the "same name" on different processors.

The cost of the algorithm is $\lceil \log p \rceil (1 + g + l) + l$ as there are $\lceil \log p \rceil + 1$ supersteps (including one for registration); during each superstep a local addition is performed (which costs 1 flop), and at most one message of size 1 word enters and exits each process.

All sums: version 2.

An alternative implementation of the prefix sums function can be achieved in a single superstep by using a temporary data structure containing up to p integers. Each process i puts the data to be summed into the i^{th} element of the temporary array on processes j (where $0 \leq j \leq i$). After all communications have been completed, a local sum is then performed on the accumulated data. The cost of the algorithm is $p + pg + 2l$.

```
int bsp_allsums2(int x) {
  int i, result,*array = calloc(bsp_nprocs(),sizeof(int));
  if (array==NULL)
    bsp_abort("Unable to allocate %d element array",bsp_nprocs());
  bsp_pushregister(array,bsp_nprocs()*sizeof(int));
  bsp_sync();

  for(i=bsp_pid();i<bsp_nprocs();i++)
    bsp_put(i,&x,array,bsp_pid()*sizeof(int),sizeof(int));
  bsp_sync();

  result = array[0];
  for(i=1;i<=bsp_pid();i++) result += array[i];
  free(array);
  bsp_popregister(array);
  return result;
}
```

The first algorithm performs a logarithmic number of additions and supersteps, while the second algorithm performs a linear number of additions but a constant number of supersteps. If the operation being performed at each iteration of the algorithm were changed from addition to another, more-costly, associative operator, then BSP cost analysis provides a simple mechanism for determining which is the better implementation.

All sums on an array.

Either of the routines defined above can be used to sum n values held in n/p blocks distributed among p processors. The algorithm proceeds in four phases:

1. The running sum of each n/p block of integers is computed locally on each processor.
2. As the last element of each n/p block contains the sum of each n/p-element segment, then either of the two simple algorithms can be used to calculate the running sums of the last element in each block (call this `last`).
3. Each processor gets the value of `last` from its left neighbouring processor (we call this `lefts_last`).
4. Adding `lefts_last` to each of the locally-summed n/p elements produces the desired effect of the running sums of all n elements.

```
void bsp_allsums(int *array, int n_over_p) {
  int i, last, lefts_last;
  bsp_pushregister(&last,sizeof(int));

  for (i=1;i<n_over_p;i++)
    array[i] += array[i-1];

  last = bsp_allsums2(array[n_over_p-1]);

  if (bsp_pid()==0) lefts_last=0;
  else
```

```
    bsp_get(bsp_pid()-1,&last,0,&lefts_last,sizeof(int));
  bsp_sync();

  for(i=0;i<n_over_p;i++)
    array[i] += lefts_last;

  bsp_popregister(&last);
}

void main() {
  int i,j,n_over_p,*xs;
  bsp_begin(bsp_nprocs());

  n_over_p  = 100;
  xs = calloc(n_over_p,sizeof(int));
  for (i=0;i<n_over_p;i++) xs[i]=1;
  bsp_allsums(xs,n_over_p);

  for(i=0;i<bsp_nprocs();i++) {
    if (bsp_pid()==i) {
      printf("On process %d: ",bsp_pid());
      for(j=0;j<n_over_p;j++) printf("%d ",xs[j]);
      printf("\n");
      fflush(stdout);
    }
    bsp_sync();
  }

  bsp_end();
}
```

14 *What are typical values of g and l for common parallel computers?*

Values of the BSP cost model parameters are shown in Table 2. The values of the g and l parameters are normalised by the instruction rate of each processor (to aid comparisons between machines, raw rates are also given in microseconds). Because this instruction rate depends heavily upon the kind of computations being done, the average of two different measured values are used:

$\lfloor s \rfloor$ measures the cost of an inner product, where $\mathcal{O}(n)$ operations are performed on a data structure of size n. The value of n is chosen to be far greater than the cache size on each processor. This benchmark therefore gives a lower-bound megaflop rate for the processor as each arithmetic operation induces a cache miss.

$\lceil s \rceil$ measures the cost of a dense matrix multiplication, where $\mathcal{O}(n^3)$ operations are performed on a data structures of size n^2. Because a large percentage of the computation can be kept in cache, this benchmark gives an upper-bound megaflop rate for the processor.

As we have already mentioned, good BSP algorithm design is often based around balanced patterns of communication. We illustrate the communication capacity g using two balanced

communications. The first is a particularly easy 1-relation, a cyclic shift of data between neighbouring processors. This benchmark provides an upper-bound rate for communication.

Parallel computers have far greater difficulty in achieving scalable communication for patterns of communication that move lots of data to many destinations. As an extreme example, we consider the p-relation generated by a total exchange among the processors. No scalable architecture can provide p^2 dedicated wires because it is too expensive. So sparser interconnections are used. For example, the Cray T3D uses a 3D Torus, while the IBM SP2 uses a hierarchy of 8-node fully-connected crossbar switches. The value of g for a total exchange therefore provides a good measure of the lower-bound rate of communication of an architecture.

Not very surprisingly, the two values of g, derived directly from a 1-relation, and from the pg cost of a p-relation total exchange can be quite different. This might mean that the 1-relation performance of the network is not very good (for example, a ring takes time proportional to p to deliver both a 1-relation and a p-relation), but usually means that the network's effective capacity is not as large as the per-link bandwidth would suggest. When cost modelling algorithms, it is advisable to use the value of g produced by the total exchange benchmark.

When $p = 1$, g represents the memory speed of the processor, taking into account any buffering of communication that may occur in the implementation of BSPLib . The efficiency of the communication network can also be roughly estimated by comparing the cost of g for one processor with g for $p > 1$. This gives a ratio of inter-processor communication to memory speed, which is 9 for the IBM SP2 (8 nodes) with switch communication, 19 for the SGI Power Challenge (4 nodes), and 8 for the Cray T3D (256 nodes).

Machine	Mflops			p	l		g (local)		g (total exch.)		$n_{1/2}$
	$\lfloor s \rfloor$	$\lceil s \rceil$	s		flops	μs	flop/word	μs/word	flop/word	μs/word	words
SGI PowerChallenge	53	94	74	1	226	3.1	0.5	0.007	0.5	0.007	80
				2	1132	15.3	9.8	0.13	10.2	0.14	12
				3	1496	20.2	8.9	0.12	9.5	0.13	12
				4	1902	25.7	9.8	0.13	9.3	0.13	12
Cray T3D	5	19	12	1	68	5.6	0.3	0.02	0.3	0.02	94
				2	164	13.5	0.7	0.06	1.0	0.08	71
				4	168	13.9	0.7	0.06	0.8	0.65	66
				8	175	14.4	0.8	0.07	0.8	0.65	59
				9	383	31.7	0.9	0.07	1.2	0.10	39
				16	181	14.9	0.9	0.07	1.0	0.08	61
				25	486	40.2	1.1	0.09	1.5	0.13	26
				32	201	16.6	1.1	0.09	1.4	0.12	28
				64	148	12.3	1.0	0.09	1.7	0.14	27
				128	301	24.9	1.1	0.09	1.8	0.15	20
				256	387	32.1	1.2	0.11	2.4	0.19	15
IBM SP2 (switch)	25	27	26	1	244	9.4	1.3	0.05	1.3	0.05	7
				2	1903	73.2	6.3	0.24	7.8	0.30	6
				4	3583	137.8	6.4	0.25	8.0	0.31	7
				8	5412	208.2	6.9	0.27	11.4	0.43	6

continued on next page ...

... continued from previous page

Machine	Mflops			p	l		g (local)		g (total exch.)		$n_{1/2}$
	$\lfloor s \rfloor$	$\lceil s \rceil$	s		flops	μs	flop/word	μs/word	flop/word	μs/word	words
Multiprocessor Sun	3.8	16.4	10.1	1	24	2.4	0.4	0.04	0.4	0.04	7
				2	54	5.3	3.0	0.29	3.4	0.34	7
				3	74	7.4	2.9	0.29	4.1	0.41	8
				4	118	11.7	3.3	0.32	4.1	0.41	11
Hitachi SR2001	2.3	8.5	5.4	1	31	5.6	0.2	0.05	0.2	0.05	16
				2	1165	216.1	2.6	0.50	3.0	0.54	8
				4	2299	426.1	2.8	0.53	3.0	0.56	8
				8	3844	712.1	3.0	0.54	3.1	0.59	8
				16	4638	911.4	3.0	0.55	3.0	0.55	8
				32	6906	1321.7	4.7	0.90	4.9	0.92	6
Convex Exemplar			10.5	1	60	5.8	0.16	0.02			18
				2	21373	2035	8.3	0.8			6
				4	64457	6138	9.2	0.9			7
				8	194476	18521	11.3	1.2			9
Digital Alpha Farm			10.1	1	29	2.9	0.3	0.03			17
				2	17202	1703.1	81.1	8.0			4
				3	34356	3401.6	83.0	8.2			4
				4	47109	4664.3	81.3	8.1			4
Parsytec GC			19.3	1	98	5.1	1.0	0.05	1.0	0.05	16
				2	6309	325	109	5.6	113	5.9	3
				4	23538	1219	190	9.9	143	7.4	3
				8	29080	1506	252	13.1	254	13.2	3
				16	224977	11600	253	13.1	342	17.7	3
				32	130527	6700	272	14.1	658	34.1	3
IBM SP2 (ethernet)	25	27	26	1	241	9.3	1.3	0.05	1.3	0.05	8
				2	18759	721.5	182.1	7.0	183.6	7.1	3
				4	39025	1500.9	388.2	14.9	628.2	24.2	5
				8	88795	3415.2	1246.6	47.3	1224.1	47.1	2

Table 2: BSP machine parameters. (1) All values for g are for communications of 32-bit words; (2) benchmarks were performed at the `-O3` optimisation level; (3) the Cray T3D, SGI PowerChallenge, IBM SP2, Parsytec GC, and Hitachi SR2001 used native implementations of the toolset; (4) the toolset used on the multiprocessor Sun was built using generic System V shared-memory facilities; (5) the Digital Alpha Farm consists of a cluster of Alpha workstations connected via FDDI and a giga-switch. The toolset implementation was built on top of a generic version of MPI (`mpich`).

Appendix B shows how these figures were obtained. The meaning of $n_{1/2}$ is explained in Section 16.

15 *How can the BSPLib be implemented efficiently on today's architectures?*

The semantics of the BSPLib operations reflects the high-level view of BSP in which computation and communication do not overlap. The Oxford implementation of BSPLib keeps these two phases separate also. Thus while the semantics of calls to `put` and `get` permits them to begin executing concurrently with the local process's computation, the performance advantages of postponing them turn out to be larger than of exploiting the potential overlap

[16]. This approach contradicts current practice in communication libraries, where overlapping computation and communication is considered a good thing, even though it can create at best a factor of two improvement. Of course, treating communication at the level of single messages provides no obvious opportunity to improve performance by postponing communication.

We have found that postponing communication until the end of local computation creates major performance-enhancement opportunities. Combining all of the messages between each processor pair means that transmission startup costs are paid only once per superstep, instead of once per message (although it does require more memory for buffering). The freedom to reorder transmissions to different processors means that patterns guaranteed to avoid congestion can be set up in software, rather than requiring expensive hardware solutions operating during the data transfers. This is important since, although congestion inside the network is not as significant a problem as it once was, it is common at the processor-network interface. The performance gains of delaying communication are so large that even the high-performance versions of the `put` and `get` operations, which are designed so that computation and communication can be overlapped without buffering, postpone transmissions until the end of the computation phase of each superstep.

The general structure of an Oxford implementation of BSPLib is that all `put` and `get` operations initiated in a superstep are delayed until the end of the superstep, and optimisations whose effect is to minimise both the absolute value of g and its variance are applied to the entire h-relation.

Regardless of the type of parallel architecture, the ability to reorder messages before transmission is crucial to creating a consistent bulk-communication behaviour without increasing the value of g. Two mechanisms used are:

- randomly ordering the messages to reduce the likelihood of troublesome patterns, and
- using a latin square to schedule transmissions in a guaranteed contention-free way.

Which of these mechanisms is to be preferred is architecture-dependent.

Recall that a latin square is a $p \times p$ square in which each of the values from 1 to p appears p times, with no repetition in any row or column. Such a square can be used as a schedule for the routing of the h-relation, using row i as the schedule for processor i, with the contents of the row regarded as the destinations for each communication time step.

The use of such mechanisms has a major effect on performance. For example, consider a total exchange algorithm shown in Figure 3 where each processor i has data x_i of size n that is to be exchanged with every other processor. After the communication, each processor will contain a data structure of size np containing all of the x_j, where $1 \leq j < p$. The BSP cost of the algorithm is $png + l$ because p messages enter and exit each processor. However, a naive implementation may have each processor send a message to processor 0 on the first time step, to processor 1 on the second, and so on. This causes p messages to contend at process 0, then p to contend at process 1, and so on. The cost of this communication will be $\mathcal{O}(p^2)$ rather than the linear cost predicted by the BSP cost formula $png + l$. An alternative ordering that does not cause contention is for processors to send their data in the order $mod(i+j, p)$; where $1 \leq j < p$, and i is the processor identifier, using a simple latin square. The expected linear (in p) cost can then be achieved.

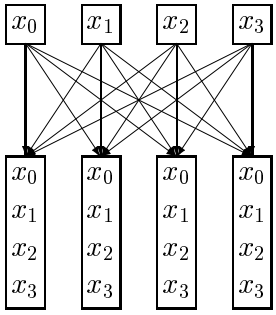

Figure 3: Total exchange between four processors

Procs	immediate transmission		BSPLib delaying and reordering	
	contention	latin square	contention	latin square
2	.168	.157	.157	.157
4	.392	.194	.191	.191
8	.461	.239	.228	.229
16	.598	.289	.344	.345
32	.784	.413	.465	.456
64	.903	.529	.548	.546
128	.961	.575	.599	.599

Table 3: The effects of node contention on the Cray T3D. Entries in the table are in seconds for routing a 4,000,000-relation. e.g., for 128 processors, 15625 integers per process.

Table 3 shows the results of an implementation that routes total exchanges. The first two columns show what happens when the programmer writes `puts` in the order that causes maximum contention and then in the latin square order that avoids it, as above. Here the runtime system is neither combining nor reordering, but transmits data as soon as the `put` is executed. The final two columns show what happens for the same two user programs when BSPLib delays messages and reorders them. As expected, reordering makes a significant difference; and the library reordering induces the improved performance regardless of the textual form of the program. Reordering makes the implementation consistent with the model, without a large sacrifice of efficiency.

The precise details of handling communication and building barriers differs depending on the specifics of target architectures:

Distributed-memory machines with remote-memory access (Cray T3D). A barrier synchronisation is performed to ensure that each process has finished its local computation. Once all the processors have passed the barrier, one-sided memory accesses are used to route messages into the memories of the remote processors. The communication phase of a superstep is completed by performing a further barrier synchronisation.

Distributed-memory machines with message-passing (IBM SP2,Hitachi SR2001, Alpha Farm, Parsytec GC). On architectures that provide native non-blocking send and blocking receive message-passing primitives, the h-relation is routed through the communica-

tion network in three phases:

1. a total exchange is performed, exchanging information about the number, sizes, and destination addresses of messages. This total exchange is considered to be the barrier synchronisation for the superstep.

2. `gets` are translated into `puts` and the data they refer to is buffered at the source processor.

3. after the total exchange, each processor knows how many messages, from every other process, it is expecting. Each process therefore knows when the communication phase of the superstep is complete by counting the incoming messages. Communication is performed by interleaving the outgoing and incoming messages, so that minimum buffering requirements are placed on the underlying message-passing system.

Shared-memory architectures (SGI Power Challenge, Sun, Convex Exemplar). The implementation on shared-memory architectures combines features from both of the implementations above. The information about the number and size of messages to be sent between each processor pair is constructed in a region of shared memory by each call to `put` and `get`. After the computation phase, a barrier synchronisation takes place to ensure that this information is frozen. Because the message information is in shared memory, an implicit total exchange can be considered to have occurred at this point. The actual exchange of data is performed in a message-passing style. First messages are copied into buffers associated with each process in shared memory. These buffers are then inspected by the remote process, and their contents copied into the remote processor's memories. Using a contention-limiting order for messages, the number of message passing buffers associated with each process can be minimised. Finally, the message information region is cleared and a further barrier synchronisation takes place to allow renewed access to it.

16 *How much effect does message size have on the value of g?*

As we have already seen, the way in which BSPLib delays communication until the end of each superstep and then combines messages into the largest possible units reduces the importance of message size. The cost model makes no distinction between the cost of a process sending h messages of size one or a single message of size h; both communications have an h-relation cost of hg. However, a superstep in which very little *total* communication occurs may still deviate from the cost model because of the effects of startup costs for message transmission.

Miller refined the standard cost model [29] using a technique of Hockney [20] to model the effect of message granularity on communication cost. In the refined model, g is defined as a function of the message size x:

$$g(x) = \left(\frac{n_{1/2}}{x} + 1\right) g_\infty \tag{1}$$

where g_∞ is the asymptotic communication cost for very large messages (g reported in Table 2 is g_∞) and $n_{1/2}$ is the size of message that produces half the optimal bandwidth of the machine so $g(n_{1/2}) = 2g_\infty$.

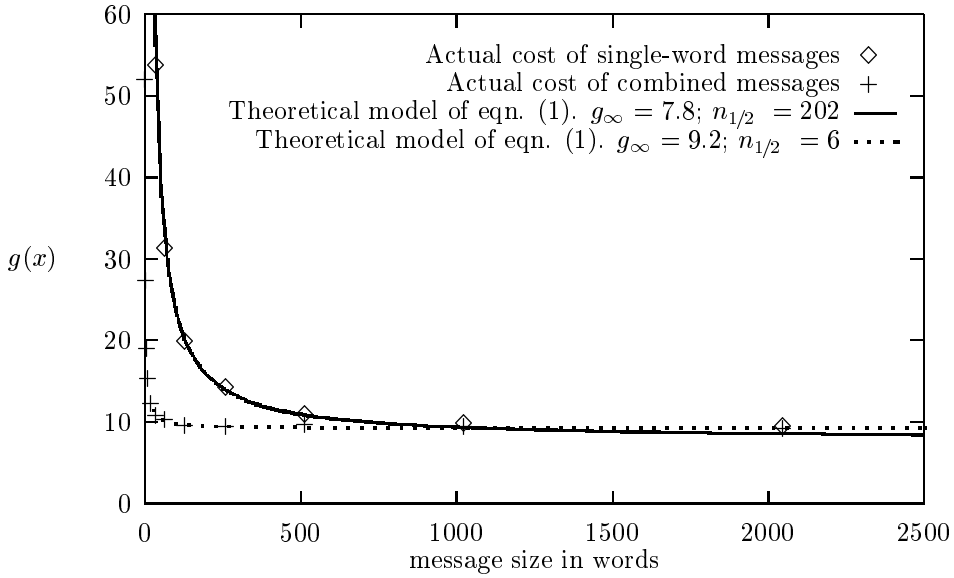

Figure 4: Fitting experimental values of $g(x)$ flops/word to Equation (1) using an 8-processor IBM SP2 with switch communication. The messages are communicated using one-sided *put* communication where a process puts data into another processor's memory. The top curve represents single-word messages and the bottom curve uses a message-combining scheme.

The value of $n_{1/2}$ in Equation (1) is determined experimentally for each machine configuration by fitting a curve to actual values of $g(x)$. Figure 4 shows the actual values of $g(x)$ on an 8-processor IBM SP2. Because messages are combined in each superstep, the value of $n_{1/2}$ is effectively reduced to 6 words. For comparison purposes, the effect of naively communicating messages separately is shown by the data points labeled "actual cost of single-word messages" in the figure. Fitting a curve to this data gives $n_{1/2} = 202$ words.

The $n_{1/2}$ parameter can be used to discover the minimum message size for which the standard cost model is within a given percentage of the more-detailed cost model. For the standard model to be within y% accuracy of the cost attributed by the model that includes message granularity, then:

$$\left(\frac{100+y}{100}\right) h_0 g_\infty = h_0 g(h_0) = \left(\frac{n_{1/2}}{h_0} + 1\right) h_0 g_\infty \tag{2}$$

where h_0 words is Valiant's parameter [35] that measures the minimum size of h-relation to achieve $n_{1/2}$ throughput. Thus the percentage error in the communication cost $h_0 g_\infty$ is

$$y \;=\; \left(\frac{100 n_{1/2}}{h_0}\right) \%. \tag{3}$$

So on the IBM SP2 with switch communication the error in the standard BSP model for communicating h_0=60 32-bit words is 10%. Moreover, as would be expected, as the size of h-relation increases, the error in the standard BSP model decreases.

17 *What tools are available to help with building and tuning BSP programs?*

The *intensional properties* of a parallel program (i.e., how it computes a result) can often be hard to understand. The BSP model goes some way towards alleviated this problem if cost analysis is used to guide program development. Unfortunately, in large-scale problems, cost analysis is rarely used at the time as program development. The role of current BSP tools [18] is to aid programmers in understanding the intensional properties of their programs by graphically providing profiling and cost information. The tools analyse the actual communication properties of a program, or analyse the predicted performance of the code assuming the parallel machine the program was run upon acts like a real BSP computer (i.e., a scalable machine with constant l and g that routes h-relations in time $hg + l$).

A central problem with any parallel-profiling system is the effective visualisation of large amounts of profiling data. In contrast to conventional parallel-profiling tools, which highlight the patterns of communication between individual sender-receiver pairs in a message passing system, the BSP approach significantly simplifies visualisation because all of the communications that occur in a superstep can be visualised as a single monolithic unit.

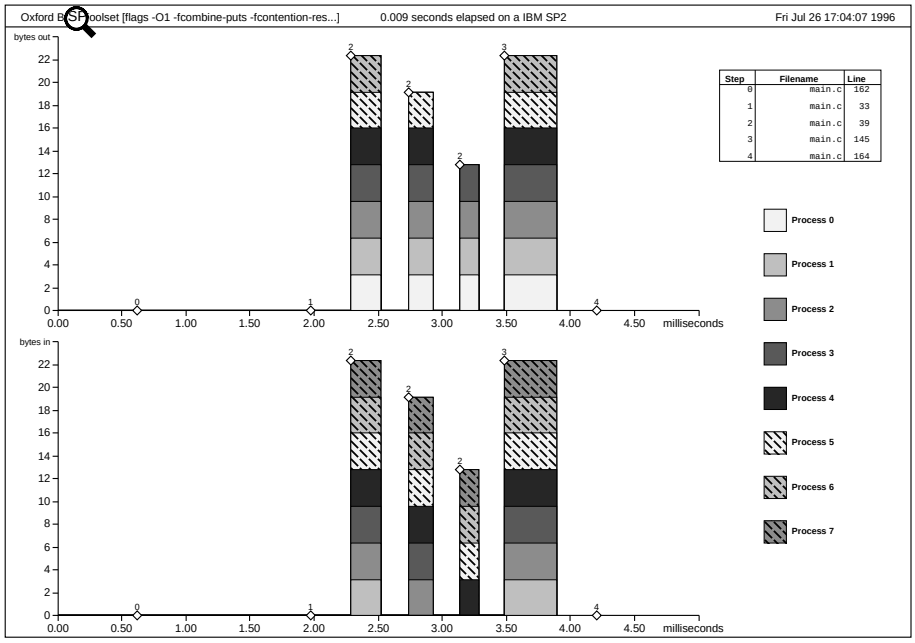

Figure 5: All sums of 32,000 elements using the logarithmic technique on an 8-processor IBM SP2

Figure 5 is an example of the results from a BSP profiling tool running on the IBM SP2. It shows a *communication* profile for the parallel prefix algorithm (with $n > p$) developed on page 12.

The top and bottom graphs in Figure 5 show, on the y axis, the volume of data moved, and on the x axis, the elapsed time. Each pair of vertically-aligned bars in the two graphs represents the total communication during a superstep. Within each communication bar is a series of bands. The height of each band represents the amount of data communicated by a particular process, identified by the band's shade. The sum of all the bands (the height of the bar) represents the total amount of communication during a superstep. The width represents

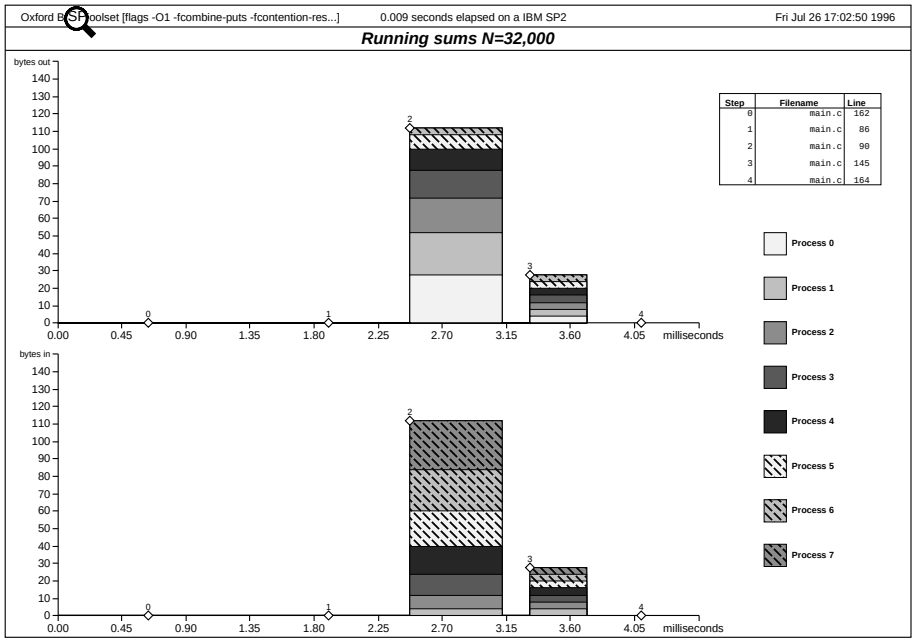

Figure 6: All sums of $32,000$ elements using total exchange on an 8-processor IBM SP2

the elapsed time spent in *both* communication and barrier synchronisation. The label found at the top left-hand corner of each bar can be used in conjunction with the legend in the right of the graph to identify the end of each superstep (i.e., the call to `bsp_sync`) in the user's code. The white space in the figure represents the local computation time of each superstep.

In Figure 5, the start and end of the running sums is identified by the points labelled 0 and 4. The white space in the graphs between supersteps 0 and 1 shows the computation of the running sums executed locally in each process on a block of size n/p. The first superstep, which is hidden by the label 1 at this scale, shows the synchronisation that arises due to registration in the function `bsp_allsums1`. The three successively-smaller bars represent the logarithmic number of communication phases of the parallel prefix technique. Contrasting the sizes of the communication bars in Figure 5 with the schematic diagram of Figure 2 graphically shows the diminishing numbers of processors involved in communication as the parallel prefix algorithm proceeds. Contrasting this method of running sums with the total-exchange-based algorithm in Figure 6 shows that although the number of synchronisations within the algorithm is reduced from $\lceil \log p \rceil$ to 1, the time spent in the total exchange of `bsp_allsums2` is approximately the same as the algorithm based upon the logarithmic technique. This is due to the larger amount of data transferred i.e., 1.51 milliseconds spent in summing p values in p processes using the parallel prefix technique, compared to 1.42 milliseconds when the total exchange is used.

Figures 7 and 8 show profiles of the same two algorithms running on a 32-processor Cray T3D, with the same data-set size as the IBM SP2. Although the T3D has a lower value for the barrier synchronisation latency than the IBM SP2 (see Table 2), reducing the number of supersteps from $\lceil \log 32 \rceil = 6$ supersteps to 1 has a marked effect on the efficiency. The version `bsp_allsums1` (i.e., logarithmic) takes 1.39 milliseconds compared to 0.91 milliseconds

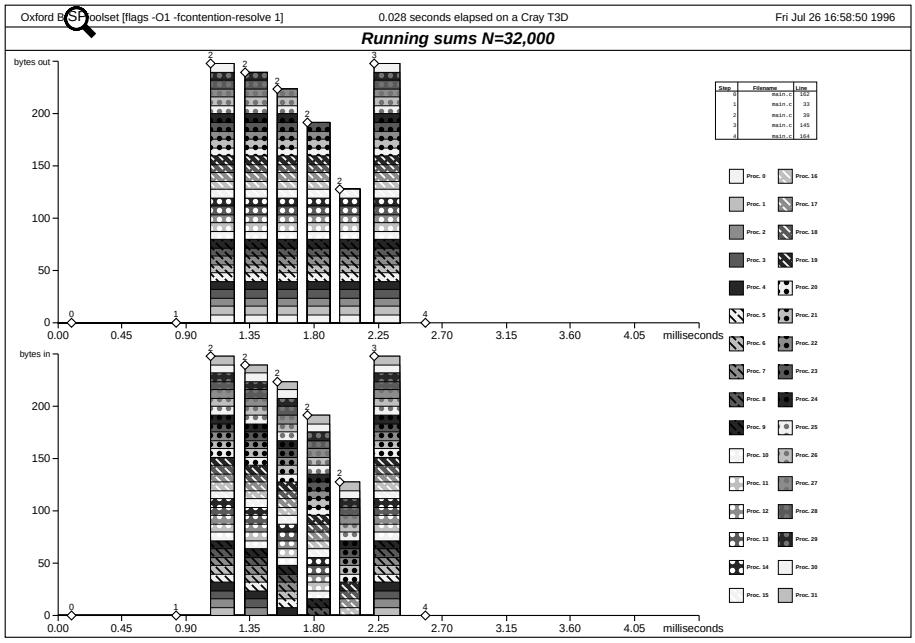

Figure 7: All sums of 32,000 elements using the logarithmic technique on a 32-processor Cray T3D

for `bsp_allsums2` (i.e., total exchange).

These data show that, for today's parallel computers, it is often better to reduce the number of supersteps, even at the expense of requiring more communication.

18 *How does BSPLib compare with other communication systems such as PVM or MPI?*

In recent years, the PVM message-passing library [1, 2, 10] has been widely implemented and widely used. In that respect, the goal of source code portability in parallel computing has already been achieved by PVM. What then, are the advantages of BSP programming, if any, over a message-passing framework such as PVM? On shared-memory architectures and on modern distributed-memory architectures with powerful global communications, message-passing models such as PVM are likely to be less efficient than the BSP model, where communication and synchronisation are decoupled. This will be especially true on those modern distributed-memory architectures that have hardware support for direct remote-memory access (or one-sided communications). PVM and all other message-passing systems based on pairwise, rather than barrier, synchronisation also suffer from having no simple analytic cost model for performance prediction, and no simple means of examining the global state of a computation for debugging.

MPI [14] has been proposed as a new standard for those who want to write portable message-passing programs in Fortran and C. At the level of point-to-point communications (send, receive etc.), MPI is similar to PVM, and the same comparisons apply. The MPI standard is very general and appears to be very complex relative to the BSP model. However, one could use some carefully-chosen combination of the various non-blocking communication

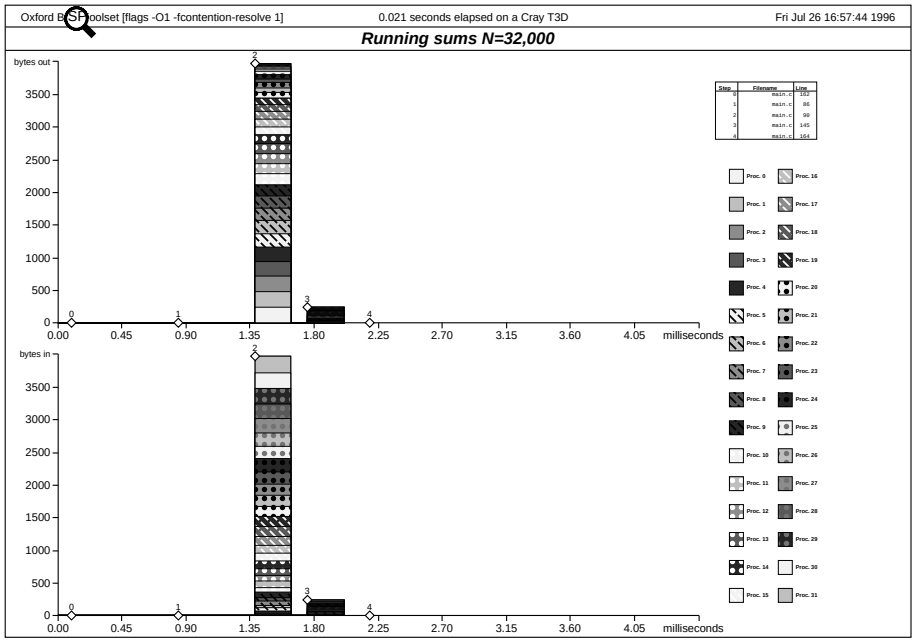

Figure 8: All sums of $32,000$ elements using a total exchange on a 32-processor Cray T3D

primitives available in MPI, together with its barrier synchronisation primitive, to produce an MPI-based BSP programming model. At the higher level of collective communications, MPI provides support for various specialised communication patterns which arise frequently in message-passing programs. These include broadcast, scatter, gather, total exchange, reduction, and scan. These standard communication patterns are also provided for BSP in a higher-level library. There has been one attempt to compare BSP performance with MPI [31] on a network of workstations. The results show that performance differences are very small, of the order of a few percent.

Compared to PVM and MPI, the BSP approach offers (a) a simple programming discipline (based on supersteps) that makes it easier to determine the correctness of programs, (b) a cost model for performance analysis and prediction which is simpler and compositional, and (c) more efficient implementations on many machines.

19 *How is BSP related to the LogP model?*

LogP [7] differs from BSP in three ways:

- It uses a form of message passing based on pairwise synchronisation.
- It adds an extra parameter representing the *overhead* involved in sending a message. This has the same general purpose as the $n_{1/2}$ parameter in BSP, except that it applies to *every* communication, whereas the BSP parameter can be ignored except for a few unusual programs.
- It defines g in local terms. The g parameter in BSP is regarded as capturing the throughput of an architecture when every processor inserts a message (to a uniformly-

distributed address) on every step. It takes no account of the actual capacity of the network, and does not distinguish between delays in the network itself and those caused by inability to actually enter the network (blocking back at the sending processor). In contrast, LogP regards the network as having finite capacity, and therefore treats g as the minimal permissible *gap* between message sends from a single process. This amounts to the same thing in the end, that is g in both cases is the reciprocal of the available per-processor network bandwidth, but BSP takes a global view of the meaning of g, while LogP takes a more local view.

Over the last few years experience in developing software using the LogP model has shown that to analyse the correctness and efficiency of LogP programs it is often necessary, or at least convenient, to use barriers. Also major improvements in network hardware and in communications software have greatly reduced the overhead associated with sending messages. In early multiprocessors, this overhead could be substantial, since a single processor handled both the application and its communication. Manufacturers have learned that this is a bad idea, and most newer multiprocessors provide either a dedicated processor to handle message traffic at each node or direct remote-memory access. In this new scenario, the only overhead for the application processor in sending or receiving a message is the time to move it from user address space to a system buffer. This is likely to be small and relatively machine-independent, and may even disappear as communication processors gain access to user address space directly, so the importance of the overhead parameter in the long term seems negligible.

Given that LogP + barriers – overhead = BSP, the above points would suggest that the LogP model does not improve upon BSP in any significant way. However, it is natural to ask whether or not the more "flexible" LogP model enables a designer to produce a more efficient algorithm or program for some particular problem, at the expense of a more complex style of programming. Recent results show that this is not the case. In [3] it is shown that the BSP and LogP models can efficiently simulate one another, and that there is therefore no loss of performance in using the more-structured BSP programming style.

20 *How is BSP related to the PRAM model?*

The BSP model can be regarded as a generalisation of the PRAM model which permits the frequency of barrier synchronisation, and hence the demands on the routing network, to be controlled. If a BSP architecture has a very small value of g , e.g. $g = 1$, then it can be regarded as a PRAM and we can use hashing to automatically achieve efficient memory management. The value of l determines the degree of parallel slackness required to achieve optimal efficiency. The case $l = g = 1$ corresponds to the idealised PRAM, where no parallel slackness is required.

21 *How is BSP related to data parallelism?*

Data parallelism is an important niche within the field of scalable parallel computing. A number of interesting programming languages and elegant theories have been developed in support of the data-parallel style of programming, see e.g. [34]. High Performance Fortran [22] is a good example of a practical data-parallel language. Data parallelism is particularly appropriate for problems in which locality is crucial.

The BSP approach in principle offers a more flexible and general style of programming than is provided by data parallelism. However, the current SPMD language implemented by BSPLib is very much like a large-grain data parallel language, in which locality is not considered and programmers have a great deal of control over partitioning of functionality. In any case, the two approaches are not incompatible in any fundamental way. For some applications, the flexibility provided by the BSP approach may not be required and the more limited data-parallel style may offer a more attractive and productive setting for parallel software development, since it frees the programmer from having to provide an explicit specification of the various processor scheduling, communication and memory management aspects of the parallel computation. In such a situation, the BSP cost model can still play an important role in terms of providing an analytic framework for performance prediction of the data-parallel program.

22 *Can BSP handle synchronisation among a subset of the processes?*

Synchronising a subset of executing processes is a complex issue because the ability of an architecture to synchronise is not necessarily a bulk property in the sense that its processing power and communication resources are. Certain architecture provide a special hardware mechanism for barrier synchronisation across all of the processors. For example the Cray T3D provides an add-and-broadcast tree, and work at Purdue [8] has created generic, fast, and cheap barrier synchronisation hardware for a wide range of architectures. Sharing this single synchronisation resource among several concurrent subsets that may wish to use it at any time seems difficult. We are currently exploring this issue.

Architectures in which barrier synchronisation is implemented in software do not have any difficulty in implementing barriers for subsets of the processors. The remaining difficulty here is a language design one—it is not yet clear what an MIMD, subset-synchronising language should be like if it is to retain the characteristics of BSP.

23 *Can BSP be used on vector, pipelined, or VLIW architectures?*

Nothing about BSP presupposes how the sequential parts of the computation, that is the processes within each processor, are computed. Thus architectures in which the processor uses a specialised technique to improve performance might make it harder to determine the value of w for a particular program, but they do not otherwise affect the BSP operation or cost modelling. The purpose of normalising g with respect to processor speed is to enable terms of the form hg to be compared to computation times so that the balance between computation and communication in a program is obvious. Architectures that issue multiple instructions per cycle might require a more sophisticated normalisation to keep these quantities comparable in useful ways.

24 *BSP doesn't seem to model either input/output or memory hierarchy?*

Both of these properties can be modelled as part of the cost of executing the computation part of a superstep. Modelling the latency of deep storage hierarchies fits naturally into BSP's approach to the latency of communication, and investigations of extensions to the BSP cost model applicable to databases are underway.

25 *Does BSP have a formal semantics?*

Several formal semantics for BSP have been developed. The paper [15] shows how these may be used to give algebraic laws for developing BSP programs. BSP is used as a semantics case study in a forthcoming book [19].

26 *Will BSP influence the design of architectures for the next generation of parallel computers?*

The contribution of BSP to architecture design is that it clarifies those factors that are most important for performance on problems without locality. It suggests that the critical properties of an architecture are:

- high permeability of the communication system, that is the ability to move arbitrary patterns of data quickly, and
- the ability to reach a consistent global state quickly by barrier synchronisation.

More subtly, it also suggests that predictability of communication delivery across a wide range of communication patterns is more important than extremely high performance for some special communication patterns, and low performance for others. In other words, low variance is more significant than low mean.

The two parameters l and g capture, in a direct way, how well an architecture achieves these two major performance properties. Details of exactly which topology to use, what routing technology, and what congestion control scheme are all subsumed in the single consideration of total throughput.

When the BSP model was first considered, it was often felt to be necessarily inefficient because of its use of permutation routing. After a while, it came to be appreciated that permutation routing is not necessarily expensive, and architectures that do it well were developed. Then the BSP model was considered inefficient because of its requirement for barrier synchronisation. It is now understood that barriers need not be expensive, and architectures that handle them well are being developed. It may be that total exchange is the next primitive to be made central to BSP and the same arguments about inefficiency may well be made. New communication technologies, such as ATM, repay foreknowledge of communication patterns, and it may be that total exchange will turn out to be a reasonable standard building block for parallel architectures as well.

27 *How can I find out more about BSP?*

Development of BSP is coordinated by *BSP Worldwide*, an organisation of researchers and users. Information about it can be found at the web site `http://www.bsp-worldwide.org/`. A standard for the BSPLib has been agreed. BSP Worldwide organises semiannual workshops on BSP. Other general papers about BSP are [23, 35].

There are groups of BSP researchers at:

- Oxford — `http://www.comlab.ox.ac.uk/oucl/groups/bsp`

- Harvard — `http://das-www.harvard.edu/cs/research/bsp.html`
- Utrecht — `http://www.math.ruu.nl/people/bisseling.html`
- Carleton — `http://www.scs.carleton.ca/~palepu/BSP.html`
- Central Florida — `http://longwood.cs.ucf.edu/csdept/faculty/goudreau.html`

as well as individuals working on BSP at a number of other universities.

Acknowledgements. We appreciate the helpful comments made on earlier drafts of this paper by David Burgess, Gaétan Hains, Jifeng He, Quentin Miller, Heiko Schröder, Bolek Szymanski, and Alexandre Tiskin.

D B Skillicorn was supported in part by EPSRC Research Grant GR/K63740 "A Unified Framework for Parallel Programming".

J M D Hill and W F McColl were supported in part by EPSRC Research Grant GR/K40765 "A BSP Programming Environment"

A The BSPLib Library

This Appendix provides slightly more detail about the current major BSP system, the BSPLib . We describe C interfaces to the library, but a Fortran version is also available.

Initialisation

Processes are created in a BSPLib program by the operations `bsp_begin` and `bsp_end`. There can only be one instance of a `bsp_begin`/`bsp_end` pair within a program, although there are two different ways to start a BSPLib program: If `bsp_begin` and `bsp_end` are the first and last statements in a program, then the entire BSPLib computation is SPMD.

In an alternative mode a single process starts execution and determines the number of parallel processes required for the calculation. It then spawns the required number of processes using `bsp_begin`. Execution of the spawned processes then continue in an SPMD manner, until `bsp_end` is encountered by all the processes. At that point, all processes except process zero are terminated, and process zero is left to continue the execution of the rest of the program sequentially. One problem with providing this mode is that some parallel machines available today, for example almost all distributed-memory machines, e.g. IBM SP2, Cray T3D, Meiko CS-2, Parsytec GC, Hitachi SR2001, do not provide dynamic process creation. Therefore we *simulate* dynamic spawning using an operation `bsp_init` which takes as its argument a procedure name. The procedure named in `bsp_init` must contain `bsp_begin` and `bsp_end` as its first and last statements.

The interface for these library operations is

```
void bsp_init(void (*startproc)(void), int argc, char **argv);
void bsp_begin(int maxprocs);
void bsp_end()
```

`maxprocs` is the number of processes requested by the user.
`startproc` is the name of a procedure that contains `bsp_begin` and `bsp_end` as its first and last statements.
`argc` and `argv` are command line size and arguments.

Enquiry

There are also operations to determine total number of processes and for each process to identify which it is. The interface for these operations is:

```
int bsp_nprocs();
int bsp_pid();
```

If the function `bsp_nprocs` is called before `bsp_begin`, then it returns the number of processors which are available. If it is called after `bsp_begin` it returns n, the actual number of processes allocated to the program, where $1 \leq n \leq$ `maxprocs`, and `maxprocs` is the number of processes requested in `bsp_begin`. Each of the n processes created by `bsp_begin` has a unique associated value m in the range $0 \leq m \leq n-1$. The function `bsp_pid` returns the associated value of the process executing the function call.

Synchronisation

A BSPLib calculation consists of a sequence of supersteps. The end of one superstep and the start of the next is identified by a call to the library procedure `bsp_sync` with interface

```
void bsp_sync();
```

DRMA

There are two ways of communicating between processes: one using direct remote-memory access (DRMA), and the other using a BSP version of message passing.

The DRMA communication operations are defined for stack- and heap-allocated data structures as well as for static data. This is achieved by allowing a process to reference only certain *registered* areas of a remote memory. In a registration procedure, processes use the operation `bsp_pushregister` to announce the address of the start of a local area which is available for global remote use. This makes it possible to execute BSP programs using heterogeneous processor architectures. Registration takes effect at the next barrier synchronisation.

```
void bsp_pushregister (void *region, int nbytes);
void bsp_popregister (void *region);
```

`region` is the starting address of the region to be registered or unregistered. The name `region` must be the same for all logically-related calls to `bsp_pushregister` or `bsp_popregister`, and implementations may check that this is true.
`nbytes` is the size of the region (used for range checking).

Each processor maintains a stack of registration slots. Logically-related calls to `bsp_pushregister` in different processes (the ith call in each process is related to the ith call in all of the others) associate a variable name and the addresses to which it is mapped in each process with the next available slot. Each `bsp_popregister` invalidates the slot at the top of the stack and hence the association of a variable name with its addresses in different processors. The argument is logically unnecessary but may be used by an implementation to check that the user's action and intent match.

The intent of registration is to make it simple to refer to variables in other processes without requiring their locations to be explicitly known. A reference to a registered name in a put or get is translated to the address corresponding to the remote variable with the same name. Here is an example:

Process 0

```
int x;
bsp_pushregister(&x, sizeof(int));
bsp_sync();
x := 3;
bsp_put(1, &x, &x, 0, sizeof(int));
bsp_sync();
```

Process 1

```
int y;
bsp_pushregister(&x, sizeof(int));
bsp_sync();
bsp_sync();
```

Process 0 and Process 1 register `x` in the first slot. When Process 0 executes a put, using `x` as the destination region name, this is mapped to the region whose address is associated with the first slot in Process 1. Therefore, the variable `x` in Process 1 has the value 3 placed in it as the result of the put.

The same, or overlapping, regions may be registered in more than one slot. Because the slots form a stack, processes must unregister regions in the reverse order to that in which they were registered.

The operation `bsp_put` pushes locally-held data into a registered remote-memory area on a target process, without the active participation of the target process. The operation `bsp_get` reaches into the registered local memory of another process to copy data values held there into a data structure in its own local memory. All gets are executed before all puts at the end of a superstep in line with the semantics that communications do not take effect locally until the end of a superstep. Their interfaces are

```
void bsp_[hp]put(
              int pid,
              const void *src,
              void *dst,
              int   offset,
              int   nbytes);
```

`pid` is the identifier of the process where data is to be stored.
`src` is the location of the first byte to be transferred by the put operation. The calculation of `src` is performed on the process that initiates the put.
`dst` is the is the base address of the area where data is to be stored. It must be a previously-registered data area.
`offset` is the displacement in bytes from `dst` to which `src` will copy. The calculation of `offset` is performed by the process that initiates the put.
`nbytes` is the number of bytes to be transferred from `src` into `dst`. It is assumed that `src` and `dst` are addresses of data structures that are at least `nbytes` in size.

```
void bsp_[hp]get(
            int   pid,
            const void *src,
            int   offset,
            void *dst,
            int   nbytes);
```

`pid` is the identifier of the process from which data is to be obtained.
`src` is the base address of the area from which data will be obtained. `src` must be a previously-registered data structure.
`offset` is an offset from `src`. The calculation of `offset` is performed by the process that initiates the get.
`dst` is the location of the first byte where the data obtained is to be placed. The calculation of `dst` is performed by the process that initiates the get.
`nbytes` is the number of bytes to be transferred from `src` into `dst`. It is assumed that `src` and `dst` are addresses of data structures that are at least `nbytes` in size.

The semantics adopted for BSPLib `bsp_put` communication is *buffered-locally/buffered-remotely*. When a `put` is executed, the data to be transferred is copied out of user address space immediately. The executing process is free to alter the contents of those locations after return from the call to `put`. While the semantics is clean and safety is maximized, puts may unduly tax the memory resources of an implementation, thus preventing large transports of data. Consequently, BSPLib also provides a *high-performance put* operation `bsp_hpput` whose semantics is *unbuffered-locally/unbuffered-remotely*. The use of this operation requires care, as correct data delivery is only guaranteed if neither communication nor local/remote computations modify either the source or the destination areas during a superstep. The main advantage of this operation is its economical use of memory. It is therefore particularly useful for applications which repeatedly transfer large data sets.

The `bsp_get` and `bsp_hpget` operations reach into the local memory of another process and copy previously-registered remote data held there into a data structure in the local memory of the process that initiated them.

BSMP

Bulk synchronous remote-memory access is a convenient style of programming for BSP computations that can be statically analysed in a straightforward way. It is less convenient for computations in which the volumes of data being communicated are irregular and data-dependent, or where the computation to be performed in a superstep depends on the quantity and form of data received at its start. A more appropriate style of programming in such cases is bulk-synchronous message passing (BSMP).

In BSMP, a non-blocking send operation delivers messages to a system buffer associated with the destination process. The message is guaranteed to be in the destination buffer at the beginning of the subsequent superstep, and can be accessed by the destination process only during that superstep. A collection of messages sent to the same process has no implied ordering at the receiving end. However, since messages may be tagged, the programmer can identify them by their tag.

In BSPLib , bulk-synchronous message passing is based on the idea of two-part messages, a fixed-length part carrying tagging information that will help the receiver to interpret the message, and a variable-length part containing the main data payload. We will call the fixed-length portion the *tag* and the variable-length portion the *payload*. In C programs, either part could be a complicated structure. The length of the tag is required to be fixed during any particular superstep, but may vary between supersteps. The buffering mode of the BSMP operations is *buffered-locally/buffered-remotely*.

The procedure to set tag size must be called collectively by all processes. Moreover, in any superstep where `bsp_set_tag_size` is called, it must be called before sending any messages.

```
void bsp_set_tag_size (int *tag_bytes);
```

`tag_bytes`, on entry to the procedure, specifies the size of the fixed-length portion of every message from the current superstep until it is updated; the default tag size is zero. On return from the procedure, `tag_bytes` is changed to reflect the *previous* value of the tag size to allow for its use inside procedures.

The tag size of incoming messages is prescribed by the outgoing tag size of the previous step.

The `bsp_send` operation is used to send a message that consists of a tag and a payload to a specified destination process. The destination process will be able to access the message during the subsequent superstep. Its interface is

```
void bsp_send(int pid,
              const void *tag,
              const void *payload,
              int payload_bytes);
```

`pid` is the identifier of the process where data is to be sent.
`tag` is a token that can be used to identify the message. Its size is determined by the value specified in `bsp_set_size_tag`.

`payload` is the location of the first byte of the payload to be communicated.
`payload_bytes` is the size of the payload.

It copies both the tag and the payload of the message out of user space into the system before returning. The `tag` and `payload` inputs may be changed by the user immediately after the `bsp_send`.

To receive a message, the operations `bsp_get_tag` and `bsp_move` are used. The operation `bsp_get_tag` returns the tag of the first message in the buffer. The operation `bsp_move` copies the payload of the first message in the buffer into `payload`, and removes that message from the buffer. Its interface is

```
void bsp_get_tag(int *status,
                 void *tag);
```

`status` returns -1 if the system buffer is empty. Otherwise it returns the length of the payload of the first message in the buffer. This length can be used to allocate an appropriately-sized data structure for copying the payload using `bsp_move`.
`tag` is unchanged if the system buffer is empty. Otherwise it is assigned the tag of the first message in the buffer.

```
void bsp_move(void *payload,
              int reception_nbytes);
```

`payload` is an address to which the message payload will be copied. The buffer is then advanced to the next message.
`reception_nbytes` specifies the size of the reception area where the `payload` will be copied into. At most `reception_nbytes` will be copied into `payload`.

```
int bsp_hpmove(void **tag_ptr_buf, void **payload_ptr_buf);
```

`bsp_hpmove` is a function which returns -1, if the system buffer is empty. Otherwise it returns the length of the payload of the first message in the buffer and (a) places a pointer to the tag in `tag_ptr_buf`; (b) places a pointer to the payload in `payload_ptr_buf`; and (c) conceptually removes the message (by advancing a pointer representing the head of the buffer).

Note that `bsp_move` flushes the corresponding message from the buffer, while `bsp_get_tag` does not. This allows a program to get the tag of a message (as well as the payload size in bytes) before obtaining the payload of the message. It does, however, require that even if a program only uses the fixed-length tag of incoming messages the program must call `bsp_move` to get successive message tags.

`bsp_get_tag` can be called repeatedly and will always return the same tag until a call to `bsp_move`.

Halt

The function `bsp_abort` can be used to print an error message followed by a halt of the entire BSPLib program. The routine is designed *not to* require a barrier synchronisation of all processes. A single process can therefore halt the entire BSPLib program.

```
void bsp_abort(char* format,...);
```

> `format` is a C-style format string as used by `printf`. Any other arguments are interpreted in the same way as the variable number of arguments to printf.

The function `bsp_time` provides access to a high-precision timer—the accuracy of the timer is implementation-specific. The function is a local operation of each process, and can be issued at any point after `bsp_begin`. The result of the timer is the time in seconds since `bsp_begin`. The semantics of `bsp_time` is as though there were `bsp_nprocs` timers, one per process. BSPLib does *not impose any synchronisation requirements between the timers in each process.*

```
double bsp_time();
```

B Benchmarking

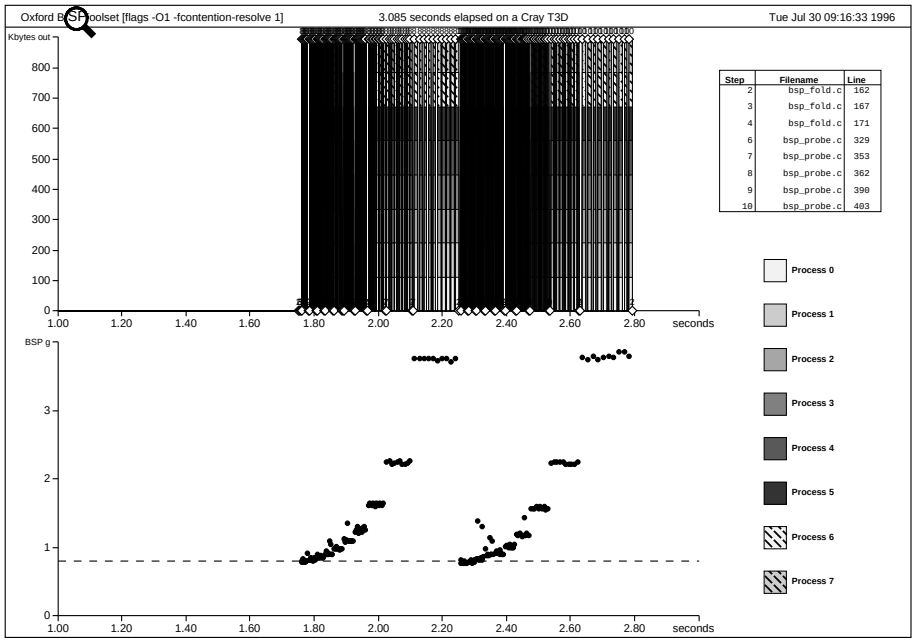

Figure 9: Cyclic shift, followed by total exchange, on an 8-processor Cray T3D

The BSP parameter l measures the minimum time for all processors to barrier synchronise. It is benchmarked by repeatedly over-sampling barrier synchronisation, whilst measuring

the wall-clock time of the synchronisations. Repeated barrier synchronisation produces a pessimistic value for l as it models the case where the computation part of each superstep completes in each processor at the same moment. This produces most contention in whatever resources are used for synchronising.

Two values for the BSP parameter g are calculated. The first is the value of g experienced when routing a local communication (a cyclic shift), and the second a global communication using a total exchange. As well as calculating the value of g, the benchmark also calculates the value for $n_{1/2}$ used in Equation 1. This is done by routing a fixed-sized h-relation (a over-sampling of 10 iterations is performed for each h-relation) using first a single message of size h; then two messages of size $h/2$; through to $h/4$ messages of size 4 words. Figures 9, 10, and 11 show communication profiles [18] for the benchmark program running on the Cray T3D and IBM SP2. Each figure contains two graphs. The upper graph contains a breakdown of the communication patterns that arise in each superstep of the benchmark. As the benchmark repeatedly routes the same h-relation, albeit with a different mix of message sizes each time, the bars in upper graph are all the same size. The lower graph shows the actual value of g attained on a superstep-by-superstep basis, calculated from the execution time of the superstep.

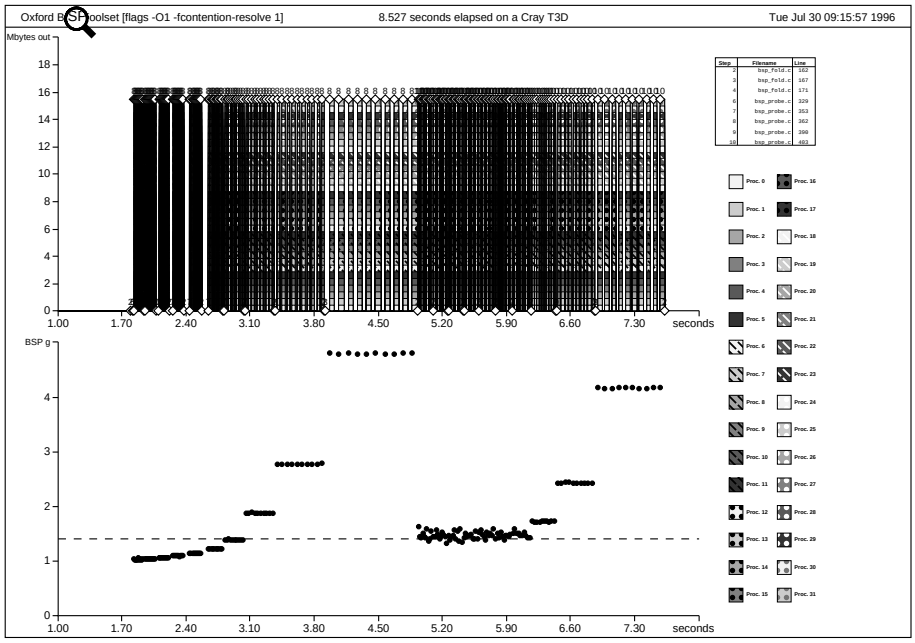

Figure 10: Cyclic shift, followed by total exchange, on an 32-processor Cray T3D

The first exponential curve in Figure 9 shows the value of g during the local-communication phase (cyclic shift) of the benchmark. Notice how the curve is a good match of Equation 1 which uses the $n_{1/2}$ parameter to account for the extra cost of communicating small messages. The second curve in Figure 9 shows the value of g when routing a series of total exchanges. The same size of h-relation, and mix of message sizes are used in this benchmark as in the local communication benchmark. This ensures that the two benchmarks have the same total theoretical cost, and *should* therefore take the same time to run. The left-hand side of each curve shows the value of g_∞ of the communication device calculated by the benchmark

program, whereas the dotted line in the graph shows the value of g_∞ from Table 2.

It should be noted that the implementation of BSPLib on the Cray, *does not* use the optimisation that combines small messages together (although it does use the contention-limiting optimisation). There is little need for this optimisation on the T3D as it is a close fit to a "BSP computer" with constant, scalable, and *predictable* values for l and g. This is borne out when a larger number of processors are used in the benchmark, as can be seen from Figure 10.

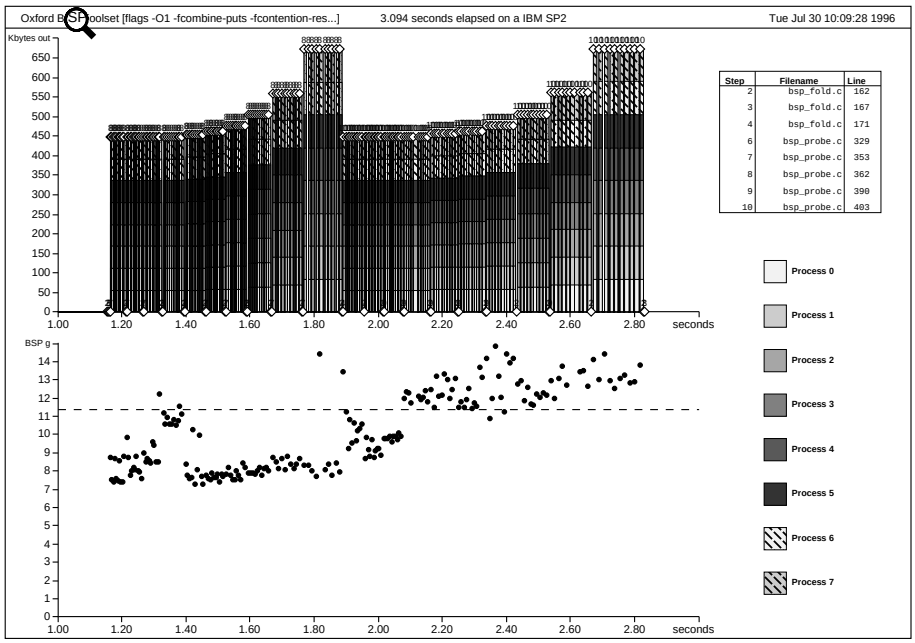

Figure 11: Cyclic shift followed by total exchange on an 8-processor IBM SP2

Figure 11 shows the same benchmark running on a eight-processor IBM SP2. Unlike the Cray, the value of g is more unpredictable. However, although g has a value which is three times larger than that of the Cray, the SP2 has a per-node computation rate twice that of the T3D, so the absolute values of g are closely matched on the two machines. From the upper graph of Figure 11 it can be seen that the amount of data communicated gradually grows, even though the benchmark routes a fixed size h-relation. The reason for this difference is that, on the SP2, small messages are combined. For the combining to work, information concerning the size and destination of the individual communications are sent with the combined individual communications, so that the destination process can unpack the data correctly. Therefore, the total size of data sent may triple due to the extra unpacking information. Nevertheless, this difference is an implementation issue, and is not reflected in the values of g reported in table 2, as the benchmark calculates a value for g for a fixed size of data communicated.

References

[1] A Beguelin, J Dongarra, A Geist, R Manchek, and V Sunderam. Recent enhancements to PVM. *International Journal of Supercomputing Applications and High Performance*

Computing, 95.

[2] Adam Beguelin, Jack Dongarra, Al Geist, Robert Manchek, and Vaidy Sunderam. A users' guide to PVM parallel virtual machine. Technical Report CS-91-136, University of Tennessee, July 1991.

[3] G. Bilardi, K.T. Herley, A. Pietracaprina, G. Pucci, and P. Spirakis. BSP vs LogP. In *Proceedings of the 8th Annual Symposium on Parallel Algorithms and Architectures*, pages 25–32, June 1996.

[4] Boppana and Chalasani. A comparison of adaptive wormhole routing algorithms. *CANEWS: ACM SIGARCH Computer Architecture News*, 21, 1993.

[5] R.P. Brent. The parallel evaluation of general arithmetic expressions. *Journal of the ACM*, 21, No.2:201–206, April 1974.

[6] P.I. Crumpton and M.B. Giles. Multigrid aircraft computations using the OPlus parallel library. In *Parallel Computational Fluid Dynamics: Implementation and Results using Parallel Computers. Proceedings Parallel CFD '95*, pages 339–346, Pasadena, CA, USA, June 1995. Elsevier/North-Holland.

[7] D. E. Culler, R. M. Karp, D. A. Patterson, A. Sahay, K. E. Schauser, E. Santos, R. Subramonian, and T. von Eicken. Logp: Towards a realistic model of parallel computation. In *Fourth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, San Diego, CA, May 1993.

[8] H.G. Dietz, T. Muhammad, J.B. Sponaugle, and T. Mattox. PAPERS: Purdue's adapter for parallel execution and rapid synchronization. Technical Report TR-EE-94-11, Purdue School of Electrical Engineering, March 1994.

[9] Al Geist, Adam Beguelin, Jack Dongarra, Weicheng Jiang, Robert Manchek, and Vaidy Sunderam. *PVM 3 Users Guide and Reference Manual.* Oak Ridge National Laboratory, Oak Ridge, Tennessee 37831, May 1994.

[10] G. A. Geist. PVM3: Beyond network computing. In J. Volkert, editor, *Parallel Computation*, Lecture Notes in Computer Science 734, pages 194–203. Springer, 1993.

[11] M. Goudreau, K. Lang, S. Rao, T. Suel, and T. Tsantilas. Towards efficiency and portability: Programming the BS model. In *Proceedings of the 8th Annual Symposium on Parallel Algorithms and Architectures*, pages 1–12, June 1996.

[12] Mark W. Goudreau, Kevin Lang, Satish B. Rao, and Thanasis Tsantilas. The Green BSP Library. Technical Report 95–11, University of Central Florida, August, 1995.

[13] M.W. Goudreau, J.M.D. Hill, K. Lang, W.F. McColl, S.D. Rao, D.C. Stefanescu, T. Suel, and T. Tsantilas. A proposal for a BSP Worldwide standard. BSP Worldwide, `http://www.bsp-worldwide.org/`, April 1996.

[14] W. Gropp, E. Lusk, and A. Skjellum. *Using MPI: Portable Parallel Programming.* MIT Press, Cambridge MA, 1994.

[15] J. He, Q. Miller, and L. Chen. Algebraic laws for BSP programming. In *Proceedings of Europar '96*. Springer-Verlag Lecture Notes in Computer Science, to appear 1996.

[16] J.M.D. Hill and D.B. Skillicorn. Lessons learned from implementing BSP. Technical Report TR-96-21, Oxford University Computing Laboratory, November 1996.

[17] J.M.D. Hill and D.B. Skillicorn. Practical barrier synchronisation. Technical Report TR-96-16, Oxford University Computing Laboratory, August 1996.

[18] Jonathan M. D. Hill, Paul I. Crumpton, and David A. Burgess. Theory, practice, and a tool for BSP performance prediction. In *EuroPar'96*, number 1124 in Lecture Notes in Computer Science, pages 697–705. Springer-Verlag, August 1996.

[19] C.A.R. Hoare and J. He. *Unified Theories of Programming.* Prentice-Hall International, to appear 1996.

[20] R.W. Hockney. Performance parameters and benchmarking of supercomputers. *Parallel Computing*, 17:1111–1130, 1991.

[21] Simon Knee. Program development and performance prediction on BSP machines using Opal. Technical Report PRG-TR-18-94, Oxford University Computing Laboratory, August 1994.

[22] C.H. Koelbel, D.B. Loveman, R.S. Schreiber, G.L. Steele Jr., and M.E. Zosel. *The High Performance Fortran Handbook.* MIT Press, Cambridge MA, 1994.

[23] W. F. McColl. Scalable computing. In J. van Leeuwen, editor, *Computer Science Today: Recent Trends and Developments*, volume 1000 of *Lecture Notes in Computer Science*, pages 46–61. Springer-Verlag, 1995.

[24] W. F. McColl and Q. Miller. The GPL language: Reference manual. Technical report, ESPRIT GEPPCOM Project, Oxford university Computing Laboratory, October 1995.

[25] W.F. McColl. General purpose parallel computing. In A.M. Gibbons and P. Spirakis, editors, *Lectures on Parallel Computation*, Cambridge International Series on Parallel Computation, pages 337–391. Cambridge University Press, Cambridge, 1993.

[26] W.F. McColl. Special purpose parallel computing. In A.M. Gibbons and P. Spirakis, editors, *Lectures on Parallel Computation*, Cambridge International Series on Parallel Computation, pages 261–336. Cambridge University Press, Cambridge, 1993.

[27] Message Passing Interface Forum. MPI: A message passing interface. In *Proc. Supercomputing '93*, pages 878–883. IEEE Computer Society, 1993.

[28] Richard Miller. A library for Bulk Synchronous Parallel programming. In *Proceedings of the BCS Parallel Processing Specialist Group workshop on General Purpose Parallel Computing*, pages 100–108, December 1993.

[29] Richard Miller. *Two approaches to architecture-independent parallel computation.* D.Phil thesis, Oxford University Computing Laboratory, Wolfson Building, Parks Road, Oxford OX1 3QD, 1994.

[30] P.B. Monk, A.K. Parrott, and P.J. Wesson. A parallel finite element method for electromagnetic scattering. *COMPEL*, 13, Supp.A:237–242, 1994.

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

[32] M.J. Quinn and P.J. Hatcher. On the utility of communication-computation overlap in data-parallel programs. *J. Parallel and Distributed Computing*, 33:197–204, 1996.

[33] J Reed, K Parrott, and T Lanfear. Portability, predictability and performance for parallel computing: BSP in practice. *Concurrency: Practice and Experience*, to appear.

[34] D.B. Skillicorn. *Foundations of Parallel Programming.* Cambridge Series in Parallel Computation. Cambridge University Press, 1994.

[35] Leslie G. Valiant. A bridging model for parallel computation. *Communications of the ACM*, 33(8):103–111, August 1990.

[36] L.G. Valiant. General purpose parallel architectures. In J. van Leeuwen, editor, *Handbook of Theoretical Computer Science, Vol. A.* Elsevier Science Publishers and MIT Press, 1990.