Concurrent Computing – Architectures, Languages and Techniques
B. M. Cook (Ed.)
IOS Press, 1999

BSP Modelling of Two-Tiered Parallel Architectures

Jeremy M.R. MARTIN
Oxford Supercomputing Centre
Wolfson Building
Parks Road
Oxford OX1 3QD, UK

Alexandre V. TISKIN
Oxford University Computing Laboratory
Wolfson Building
Parks Road
Oxford OX1 3QD, UK

Abstract. In recent years there has been a trend towards using standard workstation components to construct parallel computers, due to the enormous costs involved in designing and manufacturing special-purpose hardware. In particular we can expect to see a large population of *SMP clusters* emerging in the next few years. These are local-area networks of workstations, each containing around four parallel processors with a single shared memory.

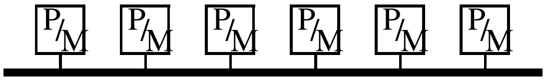
To use such machines effectively will be a major headache for programmers and compiler-writers. Here we consider how well-suited the BSP model might be for these two-tier architectures, and whether it would be useful to extend the model to allow for non-uniform communication behaviour.

1 Introduction

The structure of this paper is as follows. First we shall review the existing Bulk Synchronous Parallel (BSP) model of computation and illustrate its cost-prediction model with a suitable example. Then we shall consider how the BSP computer differs from emergent hierarchical supercomputing architectures. Next we define a two-tiered BSP computer, with a refined cost-prediction model. A representative selection of algorithms are parallelised using the new model, and their performance is analysed to see whether any significant gain in scalability is achieved with respect to the existing model.

The BSP computation model

The BSP computer[3] consists of a number of processor/memory pairs connected by a communications network. Each processor has fast access to local memory and *uniformly slow* access to remote memory.



The BSP programming model is a prominent example of the use of *remote memory transfer*. This is an alternative to message passing, in a distributed memory environment. Each process can directly write and read to the memory of a remote process. These actions are one-sided with no action by the remote process and hence there is no potential for deadlock.

Execution of a BSP program proceeds in *supersteps* separated by *global synchronisations*. A superstep consists of each process doing some calculation, using local data, *and/or* communicating some data by direct memory transfer. The global synchronisation event guarantees that all communication has completed before the commencement of the next superstep.

BSP programs are SPMD, which stands for *single program multiple data*. Each processor runs an identical program, but the programmer has access to the current process id (which is in the range 0 to $nprocs-1$, where $nprocs$ is the total number of processes) to allow different behaviour to be implemented at each node if required.

The main BSP commands are as follows:

bsp_begin, bsp_end define start and end of SPMD code

bsp_pid get local process id

bsp_nprocs get total number of threads

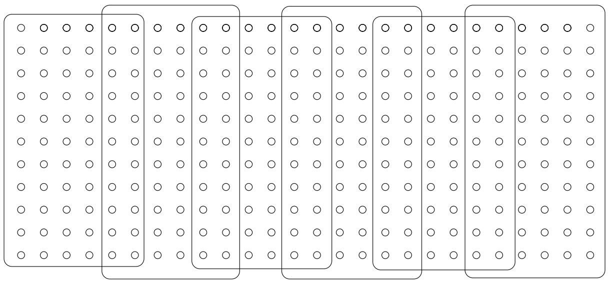
bsp_sync perform barrier synchronisation

bsp_put, bsp_get transfer data to/from other processes. In some implementations this may take place at any time during superstep so one must not use/change the data until after the next global synchronisation.

Let us consider a FORTRAN program for the numerical solution to Laplace's equation over a rectangular domain, with fixed values on the boundary, using the technique of Jacobi iteration[1]. The sequential code for a single iteration is as follows:

```
DO J = 1, JMAX
  DO I = 1, IMAX
    UNEW(I,J) = 0.25 * ( U(I-1,J) + U(I+1,J)
                       +   U(I,J-1) + U(I,J+1) )
  END DO
END DO
```

We could parallelise this using BSP by arranging the grid into overlapping strips, each to be worked on by a separate process.



Each iteration involves a computation phase, then a communication phase, and finally a global synchronisation.

```
ME = BSP_PID()
NPROCS = BSP_NPROCS()
...
DO J = 1, JMAX / NPROCS
  DO I = 1, IMAX
  UNEW(I,J) = 0.25 * ( U(I-1,J) + U(I+1,J)
```

```
                    +   U(I,J-1) + U(I,J+1) )
  END DO
END DO
IF (ME .GT. 0) THEN
  CALL BSP_PUT... ! update process to the left
END IF
IF (ME .LT. NPROCS - 1)
  CALL BSP_PUT... ! update process to the right
END IF
CALL BSP_SYNC()
```

BSP cost modelling

Perhaps the most important feature of BSP is its cost-prediction model, which makes it relatively easy to evaluate the potential efficiency of an algorithm prior to implementation. In this model the parallel computer is reduced to four constants (s, p, l, g) where
$s =$ processor speed (Mflops)
$p =$ number of processors
$l = \dfrac{\text{latency/synchronisation time}}{\text{time for 1 floating point op}}$
$g = \dfrac{\text{time to get/send 1 fp. value}}{\text{time to do 1 floating point op.}}$
The cost of a single superstep is then

$$n_o + n_c g + l$$

and the predicted execution time is given by

$$s^{-1}(n_o + n_c g + l)$$

where
n_o = max number of f.p. operations performed by any process
n_c = max number of real values communicated by any process
The BSP cost of the whole task is just the sum of the individual supersteps.

In the case of the above Jacobi iteration example we evaluate the BSP cost function as follows. Let us assume that the grid has dimension I in each direction. Then the amount of data points to be updated by each processor is $\frac{I^2}{p}$. Each update requires four arithmetic operations so we have:

$$n_o = \frac{4I^2}{p}$$

The most communication that any processor has to do is to output two complete columns to neighbours (and simultaneously to input two new columns from the same parties). So we have:

$$n_c = 2I$$

This leads to an overall cost function, for a single iteration, of

$$\frac{4I^2}{p} + 2Ig + l$$

Note that we could reduce the cost by partitioning the data grid into squares rather than strips. In that case the cost would be

$$\frac{4I^2}{p} + \frac{4Ig}{\sqrt{p}} + l$$

Using the Oxford implementation of BSP[2], parameters (s, p, l, g) have been measured for a wide variety of architectures. These may be used to predict the likely performance of a BSP algorithm prior to execution (or even program construction). Certain algorithms can be immediately consigned to the waste-bin, perhaps avoiding months of futile effort.

Here are some examples of BSP parameters for some particular architectures, based on the Oxford BSPlib benchmarks[4].

Machine	s	p	l	g
Origin 2000	101	4	1789	10.24
		32	39057	66.7
Cray T3E	46.7	4	357	1.77
		16	751	1.66
Pentium NOW	61	4	139981	1128.5
10Mbit shared ether		8	826054	2436.3
Pentium II NOW	88	4	27583	39.6
100Mbit switched ether		8	38788	38.7

2 Extending the model

The BSP model has been very successful as a reliable means of producing truly portable parallel software. A catalogue of efficient BSP algorithms, covering a wide spectrum of computational problems, has been constructed by researchers in Oxford and the rest of the world.

However real hardware may differ from the BSP model in certain important respects.

1. Local memory access times are variable – they are affected by the usual hierarchy of memory, cache and registers.

2. Access times to remote memory may vary because of non-uniformity within the interconnection network.

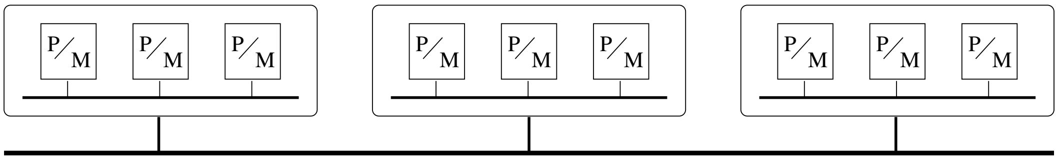
Difference 1 may be absorbed into the BSP model, if required, by allowing the value of s, the processor speed, to become application-specific. It may be measured for a particular program by benchmarking on a single processor.

Difference 2 may also be concealed using random placement of processes to processors and random message-routing, which involves diverting all messages via some arbitrary intermediate location. However, to do this removes any possibility of improving performance by exploiting fast local communications.

In an SMP cluster, the cost of communication between separate SMP units may far outweigh the cost of local communication, involving only processors within the same unit. So, although we can make such a beast behave like a BSP computer by the above technique, we may not wish to do so. Let us consider an extension to the BSP model, which allows for two levels of synchronisation.

The BSP2 computer

A BSP2 computer consists of a number of BSP units, connected by a communication network. Each BSP unit, in turn, consists of a fixed number of processor/memory pairs, connected by a local network. Each processor has fast access to its local memory and uniformly slow access to memory belonging to other processors within the same network. Each BSP unit has *uniformly slower* access to the memory within other BSP units.



Execution of a BSP2 program proceeds in *super-supersteps*, separated by global synchronisations. On each super-superstep each BSP unit performs a complete BSP computation and/or communicates some data with other BSP units.

BSP2 cost model

The BSP2 computer is modelled by seven parameters (s, p, q, l, L, g, G), where
$s =$ processor speed (Mflops)

$p =$ number of processors in each BSP unit

$q =$ number of BSP units (total processor count: $p \times q$)

$$l = \frac{\text{latency/synchronisation time for BSP unit}}{\text{time for 1 floating point op}}$$

$$L = \frac{\text{latency/synchronisation time for all BSP units}}{\text{time for 1 floating point op}}$$

$$g = \frac{\text{time to get/send 1 fp. value within a unit}}{\text{time to do 1 floating point op.}}$$

$$G = \frac{\text{time to get/send 1 fp. value between two units}}{\text{time to do 1 floating point op.}}$$

The cost of a single super-superstep is $C_l + L + n_c G$
where
C_l = max cost of BSP computation performed by any unit
n_c = max number of real values communicated by all the processors within a BSP unit to/from other units.

To realise programs in this extended model, additional BSP2 primitives would need to be provided, e.g.

bsp_uid get BSP unit id

bsp_units get total number of BSP units, q

bsp_usync synchronise within a unit

bsp_uput, bsp_uget transfer data to/from processors within the same unit.

So we are now in a position to develop BSP2 algorithms and to calculate their performance times. For the sake of comparison, we shall need some means of predicting how fast ordinary BSP programs would run on the BSP2 computer too. This involves working out BSP parameters (s, p, l, g) for the BSP2 machine (s, p, q, l, L, g, G).

Clearly the s-parameter, representing processor speed, will be unchanged. The total processor count will be pq. Overall synchronisation of the system will require a local synchonisation within each unit to be followed by global synchronisation, and therefore will have cost $l + L$. The sole difficultly arises in deciding upon a value for g.

Due to the random placement of processes on processors (in BSP not BSP2) we can assume that communication will be remote rather than local. The cost of sending a single real value from one processor to a processor in another unit must incorporate the local data-transmission cost of g plus the cost of transmission between units. The rate at which the p individual processors, within a a unit, can transmit data to other units should be p times slower than the rate G at which the unit as a whole can transmit data. These considerations lead to a g-value of $g + pG$. Thus the ordinary BSP parameters for the BSP2 machine (s, p, q, l, L, g, G) are $(s, pq, l + L, g + pG)$.

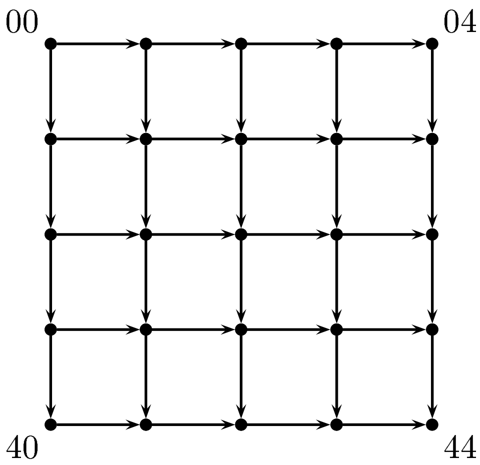
3 BSP2 parallelisation and cost-analysis of representative algorithms

In this section we work out the BSP2 cost for the natural BSP2 parallelisation of certain important algorithms. In each case this is compared with the cost of running the best-known BSP algorithm on the BSP2 machine. From now on, for the purpose of clarity, we shall omit certain small constants from cost expressions.

The algorithms we shall consider are as follows:

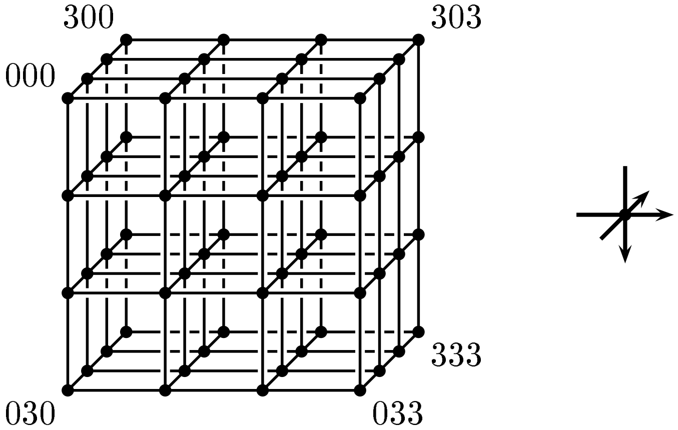
Diamond DAG: a computation with a dependence pattern shaped as an $n \times n$ regular square grid.

The cost function for the natural BSP parallelisation of this algorithm is $\dfrac{n^2}{p} + ng + pl$.



Cube DAG: a computation with a dependence pattern shaped as an $n \times n \times n$ cube grid.

The BSP cost is $\dfrac{n^3}{p} + \dfrac{n^2}{\sqrt{p}}g + \sqrt{p}l$.



Matrix-vector multiplication: assuming that the matrix data are predistributed across the processors, the BSP cost is

$$\frac{n^2}{p} + \frac{n}{\sqrt{p}} g + l$$

Matrix-matrix multiplication: BSP cost $\frac{n^3}{p} + \frac{n^2}{p^{\frac{2}{3}}} g + l$.

Parallel sort by regular sampling: BSP cost $\frac{n \log n}{p} + \frac{n}{p} g + l$.

For full details of these algorithms, and how they are parallelised using BSP, please see [6] and [7].

The following table lists the BSP2 cost of each of the above algorithms against the cost of running BSP code on the BSP2 machine.

Algorithm	BSP2 cost	BSP on BSP2 machine
Diamond DAG	$\frac{n^2}{pq} + n(g+G) + pql + qL$	$\frac{n^2}{pq} + n(g+pG) + pq(l+L)$
Cube DAG	$\frac{n^3}{pq} + n^2\left(\frac{g}{\sqrt{pq}} + \frac{G}{\sqrt{q}}\right) + \sqrt{pq}l + \sqrt{q}L$	$\frac{n^3}{pq} + n^2\left(\frac{g}{\sqrt{pq}} + \frac{\sqrt{p}G}{\sqrt{q}}\right) + \sqrt{pq}(l+L)$
Matrix $\times$ vector	$\frac{n^2}{pq} + n\left(\frac{g}{\sqrt{pq}} + \frac{G}{\sqrt{q}}\right) + l + L$	$\frac{n^2}{pq} + n\left(\frac{g}{\sqrt{pq}} + \frac{\sqrt{p}G}{\sqrt{q}}\right) + l + L$
Matrix $\times$ matrix	$\frac{n^3}{pq} + n^2\left(\frac{g}{(pq)^{\frac{2}{3}}} + \frac{G}{q^{\frac{2}{3}}}\right) + l + L$	$\frac{n^3}{pq} + n^2\left(\frac{g}{(pq)^{\frac{2}{3}}} + \frac{p^{\frac{1}{3}}G}{q^{\frac{2}{3}}}\right) + l + L$
PSRS	$\frac{n \log n}{pq} + n\left(\frac{g}{pq} + \frac{G}{q}\right) + l + L$	$\frac{n \log n}{pq} + n\left(\frac{g}{pq} + \frac{G}{q}\right) + l + L$

When these cost functions are analysed in detail they do not make any serious case for using BSP2. If we assume that l and g are small, compared with L and G respectively, then the best speedup factor is achieved for the Diamond DAG algorithm – speedup by a factor of p in the communications. However this is a *compute-bound* algorithm – that is to say that the dominant term in the BSP cost function is the computation element. So, there is no particular benefit in speeding up the communications by, say, a factor of 10.

The only algorithm we have considered that is not compute-bound is PSRS. In this case there is no benefit at all in BSP2 parallelisation anyway. In general, irregular problems cannot benefit even from single-level BSP data locality, therefore the second level of data locality is of little use.

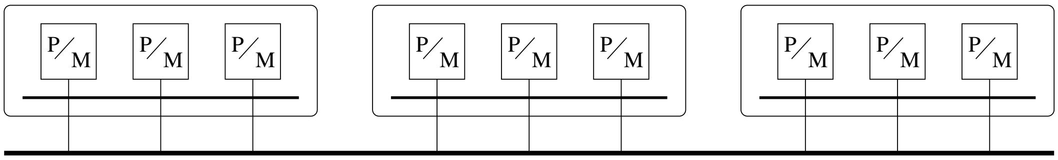
4 Conclusions

The BSP model has proved a trusty tool in the discipline of parallel programming for producing reliable and portable codes, with predictable efficiency. Now, however, additional complexity in the hardware, forces a review of the model.

Here we have considered how to extend the BSP model hierarchically in an attempt to exploit fast local communications. However, none of the algorithms we have analysed shows any significant benefit from this approach, even when local communication is several orders of magnitude faster that remote communication. It seems that the scalability of each of the wide range of algorithms considered is fundamentally constrained by the slowest communication paths in the hardware, and the presence of fast local links may be of little benefit.

The failure of the BSP2 model to provide any major performance gain provides a strong case for using the existing, flat BSP model to program hierarchical architectures, such as we have considered. Manufacturers' effort should be directed towards improving the global communication factors, in order to bring them as close as possible to those achieved locally.

One important design decision we took, when defining the BSP2 computer, was to assume that all communication between BSP units is serialised through single points of contact. The alternative would have been to allow each processor within a unit parallel access to the outer world.



This model would require an adjustment to how n_c, the communication component of a super-superstep, is calculated. Now it would be defined as the maximum amount of global communication performed by any one processor in the system. We have analysed all the algorithms from section 4 using this slight variant of the model, and we have found that it is essentially equivalent to the BSP2 computer (with parameter G replaced with G/p). So there is no extra information that is likely to be revealed.

One approach to programming SMP clusters, that has been already been investigated[5], is to use a combination of shared memory programming (by compiler directives) at the level of each SMP unit and explicit message passing (with MPI) between SMPs. This mixed-mode style of coding would require the programmer to be proficient in both forms of parallelisation. The use of a single consistent model, such as BSP, should be appealing to programmers when contrasted with this hybrid approach considered by others. The existence of a metric for performance prediction is also a major bonus.

Acknowledgements

We have enjoyed the considerable benefit of discussions with Bill McColl and Jon Hill while preparing this paper.

References

[1] Gene Golub and James M. Ortega *Scientific Computing: An Introduction with Parallel Computing*, Academic Press, Inc 1993.

[2] Jonathan M. D. Hill, Bill McColl, Dan C. Stefanescu, Mark W. Goudreau, Kevin Lang, Satish B.Rao, Torsten Suel, Thanasis Tsantilas, and Rob Bisseling *BSPlib: The BSP Programming Library*, Parallel Computing, to appear

[3] W. F. McColl *Scalable Computing*, In J van Leeuwen, editor, *Computer Science Today: Recent Trends and Developments*, Lecture Notes in Computer Science 1000, pages 41-61. Springer-Verlag 1996

[4] BSP Machine Parameters, see
URL: `http://www.BSP-Worldwide.org/implmnts/oxtool.htm`

[5] Stef Salvini, Brian T. Smith and John Greenfield *Towards Mixed Mode Parallelism on the New Model F50-based IBM SP System*, Technical Report AHPCC98-003, Albuquerque High Performance Computing Centre 1998.

[6] A. Tiskin *The bulk-synchronous parallel random access machine*, Theoretical Computer Science,196, 1–2, pp. 109–130 Elsevier 1998

[7] A. Tiskin *The design and analysis of bulk-synchronous parallel algorithms*, Oxford University D.Phil thesis, to appear 1999